Semantic role-based access control

Alexander William Macfie

Faculty of Science and Technology

Semantic Role-Based Access Control

Alexander William Macfie

A thesis submitted in partial fulfilment of the requirements of the University of Westminster for the degree of Doctor of Philosophy

July 2014

Ontological Role-Based Access Control

Alexander William Macfie
University of Westminster, London, UK
July 2014

Abstract: In this thesis we propose two semantic ontological role-based access control (RBAC) reasoning processes. These processes infer user authorisations according to a set of role permission and denial assignments, together with user role assignments. The first process, SO-RBAC (Semantic Ontological Role-Based Access Control) uses OWL-DL to store the ontology, and SWRL to perform reasoning. It is based mainly on RBAC models previously described using Prolog. This demonstrates the feasibility of writing an RBAC model in OWL and performing reasoning inside it, but is still tied closely to descriptive logic concepts, and does not effectively exploit OWL features such as the class hierarchy. To fully exploit the capabilities of OWL, it was necessary to enhance the SO-RBAC model by programming it in OWL-Full. The resulting OWL-Full model, ESO-RBAC (Enhanced Semantic Ontological Role-Based Access Control), uses Jena for performing reasoning, and allows an object-oriented definition of roles and of data items. The definitions of roles as classes, and users as members of classes representing roles, allows user-role assignments to be defined in a way that is natural to OWL. All information relevant to determining authorisations is stored in the ontology. The resulting RBAC model is more flexible than models based on predicate logic and relational database systems.

There are three motivations for this research. First, we found that relational database systems do not implement all of the features of RBAC that we modelled in Prolog. Furthermore, implementations of RBAC in database management systems is always vendor-specific, so the user is dependent on a particular vendor's procedures when granting permissions and denials. Second, Prolog and relational database systems cannot naturally represent hierarchical data, which is the backbone of any semantic representation of RBAC models. An RBAC model should be able to infer user authorisations from a hierarchy of both roles and data types, that is, determine permission or denial from not just the type of role (which may include sub-roles), but also the type of data (which may include sub-types). Third, OWL reasoner-enabled ontologies allow us to describe and manipulate the semantics of RBAC differently, and consequently to address the previous two problems efficiently.

The contribution of this thesis is twofold. First, we propose semantic ontological reasoning processes, which are domain and implementation independent, and can be run from any distributed computing environment. This can be developed through integrated development environments such as NetBeans and using OWL APIs. Second, we have pioneered a way of exploiting OWL and its reasoners for the purpose of defining and manipulating the semantics of RBAC. Therefore, we automatically infer OWL concepts according to a specific stage that we define in our proposed reasoning processes. OWL ontologies are not static vocabularies of terms and constraints that define the semantics of RBAC. They are repositories of concepts that allow *ad-hoc* inference, with the ultimate goal in RBAC of granting permissions and denials.

Table of Contents

Index of Figures

Index of Tables

Index of Code Snippets

Index of Outputs

Index of Texts

Index of Formulae

List of Abbreviations

Access Control Models

DAC	Discretionary Access Control
MAC	Mandatory Access Control
RBAC	Role-Based Access Control
DRBAC	Dynamic RBAC
GRBAC	Generalized RBAC
TRBAC	Temporal RBAC
GTRBAC	Generalized TRBAC
OBAC	Ontology-Based Access Control
ABAC	Attribute-Based Access Control
SO-RBAC	Semantic and Ontology-based Role-Based Access Control
ESO-RBAC	Enhanced Semantic and Ontology-based Role-Based Access Control
TBAC	Task-Based Access Control
TMAC	Team-Based Access Control
ARBAC97	Administrative RBAC 1997
OrBAC	Organization Based Access Control
SAC	Semantic Access Control
SACE	Semantic Access Control Enabler
OASIS	Open Architecture for Securely Interworking Services
MOSQUITO	Mobile Workers' Secure Business Applications in Ubiquitous Environments
PERMIS	PrivilEge and Role Management Infrastructure Standards

Languages

OWL	Web Ontology Language
OWL-DL	OWL Description Logics
XML	Extensible Markup Language
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
SQWRL	Semantic Query-enhanced Web Rule Language
SPARQL	*SPARQL* Protocol and RDF Query Language
XACL	XML Access Control Language
XACML	XML Access Control Markup Language
DTD	Document Type Definition

RBAC Terminology

URA	User-Role Assignment
PRA	Role-Permission Assignment
SSD	Static separation of duties
DSD	Dynamic separation of duties

Database Terminology

DBMS	Database Management System
VPD	Virtual Private Databases
CIM	Common Information Model

SWRL Terminology

TBox	Terminology Box
ABox	Assertion Box

1 Introduction

An increasingly important issue in data and application security is the use of security models. This thesis focuses on data security models for medical data, although the same principles are equally applicable to other fields, such as banking. The information that medical databases contain is highly sensitive, holding personal data about individuals and needing a strict way of protecting the privacy of patient personal and medical data [1][2][3]. When referring to the use of electronic medical records, of utmost concern is the privacy and security of individual patient information in clinical databases [4]. Measures for protecting medical data are supported by law in the UK by the Data Protection Act 1998 in the UK [5], in the US by the Fair Health Information Practices Act of 1994 [2], and elsewhere.

Wiederhold *et al.* [6] proposed Trusted Interoperation of Healthcare Information (TIHI), a centralised solution for assigning a security officer the responsibility to manage the sharing of sensitive information. However, this approach may not be appropriate for healthcare environments, as the dynamic and *ad hoc* nature of sharing healthcare and medical data would place considerable burdens on the workload of such an officer [7]. Some other works on security requirements include security policies and policy models [8][9] that put forward the concepts of clinical governance and availability of clinical knowledge.

The traditional methods of database access control are Mandatory Access Control (MAC) and Discretionary Access Control (DAC) [10]. In DAC, the owner of the data determines who has access to it. [11] This is a very widely used security model, and is widely used in operating systems and relational databases, but is rather insecure and hard to maintain. MAC grants access according to a hierarchical control structure. It is commonly used in the military, but is generally considered to be too rigid for use in the corporate context. [12]

There has been much interest recently in the development of flexible Role-Based Access Control (RBAC) models, in which access to data depends on a user's role. In RBAC, permissions and users are both assigned to roles. RBAC systems can be divided into two types: static RBAC, in which the permissions assigned to users and roles do not change, and dynamic RBAC, in which permissions assigned to roles may change according to internal and external contexts. This is particularly useful in pervasive software applications, which are dependent on changeable context. In these situations, access control requirements are likely to change constantly [13].

RBAC models have been built in logic programming languages such as Prolog [14] for almost two decades, and have been implemented in database management systems such as Oracle, Postgres and MySQL. With the standardisation of Semantic Web Technology [15] and introduction of web languages based on predicate logic such as OWL (Web Ontology Language) [16] and SWRL [17], we have been able to build RBAC models that are database independent in their implementations and which can use natural inheritance available in SWRL in order to address hierarchical nature of RBAC models.

This thesis first examines the modelling of static and dynamic RBAC using predicate logic and its applicability to relational DBMS (Database Management Systems). The static RBAC elements are those of Barker & Stuckey [18] [19], and the dynamic elements are devised by Strembeck & Neumann [20][21]. The static and dynamic RBAC are implemented in Prolog as described by those authors, and then in the Oracle relational DBMS.

We then explore the possibility of developing an RBAC model developed in OWL, and creating a reasoning process with SWRL upon RBAC concepts codified in OWL. We have used our experiences of defining RBAC model in Prolog and converting its facts and rules into OWL modelling concepts and reasoning. This new ontological RBAC

model is called SO-RBAC (**S**emantic and **O**ntology-based **R**ole-**B**ased **A**ccess **C**ontrol). SO-RBAC uses OWL-DL, with reasoning performed by SWRL, and directly translates the static RBAC model from Prolog.

We then extend SO-RBAC to develop ESO-RBAC (**E**nhanced **S**emantic and **O**ntology-based **R**ole-**B**ased **A**ccess **C**ontrol). ESO-RBAC uses OWL-Full, with reasoning performed by Jena, and represents a novel method of modelling a more flexible RBAC by taking advantage of some of the native features of ontologies, such as class hierarchy.

Early iterations of the systems now known as SO-RBAC and ESO-RBAC were called **D**ynamic **O**ntology-based **R**ole-**B**ased **A**ccess **C**ontrol (DO-RBAC) [22][23].

In the ontological models, the reasoning process upon OWL concepts grants permissions or denials solely within OWL-enabled ontological environments. Therefore OWL RBAC implementation in any data centric environment, where RBAC is needed, will be managed by accessing OWL classes though the Protégé OWL-API [24]. Consequently our reasoning process is application and database independent and the process of reasoning, which results in either permissions or denials, is being done within OWL/SWRL enabled environment.

This thesis is organised as follows. Chapter 2 presents an overview of the access control models DAC, MAC and RBAC.

Chapter 3 examines the feasibility of implementing, in a relational database management system (DBMS), a dynamic RBAC model for a hospital database originally written in Prolog. The static RBAC model is first described, followed by its implementation in Oracle. The dynamic RBAC is then described in the same way. The dynamic RBAC is built upon the static RBAC model of Barker & Stuckey [18], and has dynamic constraints separately defined through their context constraints. Table 2 shows the roles of Hospital staff and the permissions assigned to them as defined in the static RBAC model. The static RBAC schema is reused from earlier research prototypes where we experimented with solutions for data sharing across the NHS [25]. The dynamic RBAC design is built upon the semantics of the static RBAC and extended by Strembeck & Neumann [21].

Chapter 4 describes some of the problems with implementation of RBAC in traditional relational database systems, and introduces the modelling of RBAC using the XML and the Semantic Web.

Chapter 5 presents SO-RBAC, the first of our proposed ontology-based RBAC models, which was translated from our traditional RBAC in Prolog as described in Section 3.2.1 into OWL-DL with reasoning performed using SWRL. It is important to note that the purpose of SO-RBAC was not to create a pure ontological RBAC model, but to demonstrate the feasibility of mapping an RBAC model based on Prolog facts and rules into an ontology. Therefore, the proposed SO-RBAC is not designed from 'scratch'. It is instead based on a set of existing Prolog facts and rules, which are translated into an ontological schema. Prolog facts are modelled as instances within OWL classes, or as properties of these classes. RBAC rules are modelled through domain and range constraints, is-a relationships and inheritance, or using SWRL rules. The model is described, and the results of implementation are shown.

Chapter 6 describes the Enhanced Semantic Ontology-based RBAC (ESO-RBAC). Most previous ontologies for access control have used OWL-DL. Although this is widely supported and easy to understand, it was found to be inflexible. ESO-RBAC uses OWL-Full so that classes, as well as instances, can be used as the Domain and Range of properties. This increases flexibility in defining properties, and allows the use of OWL's native class hierarchy in defining roles in an object-oriented fashion. Therefore, roles need to be defined as classes, not as instances. However,

some properties in the ontology take roles as their domains and/or ranges. Unlike OWL-DL, OWL-Full permits the use of classes as property parameters.

Chapter 7 evaluates the work in this thesis on modelling RBAC in Prolog, relational database management systems and ontologies. It contrasts OWL with description logic, and considers the advantages and disadvantages of each. the main advantages of OWL over predicate logic in modelling RBAC are as follows:

- the ability to use the ontological class and property hierarchies as part of the model, allowing a natural representation of hierarchical relationships and eliminating the need for certain computations;
- the ability to query static ontologies quickly without recomputation;
- independence of the querying layer from the ontology;
- OWL and reasoning languages are not vendor-specific.

However, the ontology needs to be rebuilt every time the data or permissions change, and the reasoning process is slow and the OWL files are large. Issues are also identified with the ability using currently available tools to perform reasoning, particularly for the ESO-RBAC model using OWL-FULL.

Further development of ESO-RBAC is also discussed in Chapter 7, in order to refine the dynamic RBAC model by, for instance, introducing a hierarchy of context constraints.

2 The Domain: Access Control Models

2.1 Database Security and Access Control

Database security is mainly concerned with the availability, integrity and confidentiality of data stored and shared within structured data repositories [1]. We understand availability to mean that authorized access to a database is never denied, and integrity to mean that database rules (such as integrity constraints and rules defining who is authorized to access the database) are not breached. Confidentiality refers to the protection of data about individuals. Security in general has three stages [11]:

1. authentication (are you who you claim to be?),
2. access control (protection from unauthorized access requests), and
3. audit (checking for security breaches as they happen or after they have occurred).

We are concerned with stage 2, access control. Authorizations and access control mechanisms reduce the risk of confidentiality, integrity and availability of data being breached, and consequently contribute towards secure mechanisms for data sharing and collaborations in database applications. Access control involves analyzing and checking each access query against resources [26]. It requires access control rules to define the basis upon which access is granted or denied, as well as procedures to check requests for access against the rules.

The traditional methods of database access control are Mandatory Access Control (MAC) and Discretionary Access Control (DAC) [10].

In DAC, the owner of the data determines who has access to it. [11] DAC is a very widely used access control mechanism. For example, Unix file system permissions and the SQL GRANT/REVOKE model [27] are based on DAC. DAC is simple to apply and understand, but has two problems in a large corporate context. DAC is *not very secure*. In many corporate settings it is inappropriate for any individual user to "own" an object, thus enabling him possibly to revoke access to it for any other individual in the organization. DAC is also *difficult to maintain* in an organization with many users. The same permissions that apply to a number of users performing the same job need to be defined individually for each user, which is likely to lead to inconsistency in definitions of different users with the same role. If the access rights for a particular group of users needs to be changed (for example, if the database is restructured, or if the business rules change), then this change needs to be made not only to new users, but also to every existing user in that group.

In MAC, users do not own objects, and access is granted according to a hierarchical control structure. A typical MAC system has four levels of security ('Top Secret', 'Secret', 'Confidential' and 'Unclassified'), to which both data and users are assigned. A strict rule of 'write-up, read-down' is applied, where users can only write to data in a security classification *above* their own, and can only read data assigned to a security classification *below* their own (anyone can sign in the President, but no-one can check whether he is signed in). [28]. MAC is commonly used in the military, but is generally considered to be too rigid for use in the corporate context. [12]

2.2 Introduction to RBAC

Role-Based Access Control (RBAC) [29] is an increasingly popular security mechanism that unlike DAC and MAC does not directly assign access rights to users. Instead, users are assigned roles, and the roles are assigned access

rights. Thus, access rights of users are determined their according to their functions within the organization. “Control is based on employee functions rather than data ownership.” [27]

RBAC allows a business-oriented and non-technical administration approach to managing access to database records. The idea is to break the association between database users and their permissions for accessing databases, by introducing roles authorized to users and permissions authorized for such roles. In other words, access control is being administered by managing associations between users and roles and between roles and permissions [30][31][32]. Note that MAC can be regarded as a simple case of RBAC with four roles and a particular set of permissions. Similarly, DAC can be regarded as a simplified RBAC where each user has his own ‘role’.

A user may have more than one role. RBAC fits in very well with the division of roles in a typical health service, where doctors, nurses, receptionists, etc. all have particular jobs and thus access particular data. RBAC has been defined in terms of three factors, namely secrecy (confidentiality), integrity and availability [28]:

> “Security is compromised if information is disclosed to users not authorized to access it. Integrity is compromised if information is improperly modified, deleted or tampered. Availability is compromised if users are prevented from accessing data for which they have the necessary permissions.”

A fundamental difference between RBAC (and MAC) and DAC is that “users cannot pass access permissions on to other users at their discretion”. [27] Generally, only the database administrator is given the power to assign roles to users.

RBAC is seen as the most comprehensive access control method and a powerful concept for addressing security administration needs in database applications [33]. It was standardized by the National Institute of Standards and Technology (NIST) [34] and reduces the complexity of authorization management [35]. RBAC controls access to information based on users’ work activities [12]. It is easy to change users’ permissions without modifying the underlying access structure by simply adding or removing people from roles [35]. As a result, RBAC is adaptable to any organizational structure and can evolve over time as the organization changes. This concept has been further developed to allow roles to be composed of other roles [12] using a role hierarchy, which prevents repetition of role-permission assignments. It has also been developed for use in networked environments [36].

RBAC can be represented using predicate logic [37], which enables the building of models in logic programming languages such as Prolog [14]. RBAC has been implemented in a trust infrastructure [38], in order to comply with wider security and safety requirements in healthcare applications, using the DRIVE RBAC model. They distinguish between static role assignment to users and dynamic allocation of roles at session time. Zhang *et al.* [7] proposed a delegation framework. based on the RDM2000 RBAC model [39], that addresses how to advocate selective information sharing in role-based systems while minimizing the risks of unauthorized access in healthcare information systems. The formal specification of access control policies in clinical information systems can be found in [37]. They leverage characteristics of temporal First-Order Logic to cope with dynamic access control policies and reduce the risks to confidentiality, integrity and availability of medical data.

RBAC concepts appear to be exploited in many healthcare projects, and it has been accepted that RBAC is more appropriate than any other approach in the domain of security in medical databases [40][41]. Mavridis [1] developed a security policy called eMEDAC which was based on DAC, MAC and RBAC, and was able to preserve the availability, integrity, and confidentiality of a medical records system. In 2000, they extended eMEDAC with the development of DIMEDAC which incorporated additional features such as the hyper node hierarchies and the three-dimension access matrix [42]. Another solution that employs role hierarchies was proposed in [43]. It uses digital certificates,

cryptography and security policy to control access to clinical intranet applications. Their system consists of two phases: the ways users gain their security credentials; and how these credentials are used to access medical data.

2.2.1 Simple Static RBAC

Static RBAC consists of a set of fixed rules concerning access to data. In other words, a user in role A can read and write to object X, while a user in role B can read objects Y and Z. These permissions are determined at compile-time.

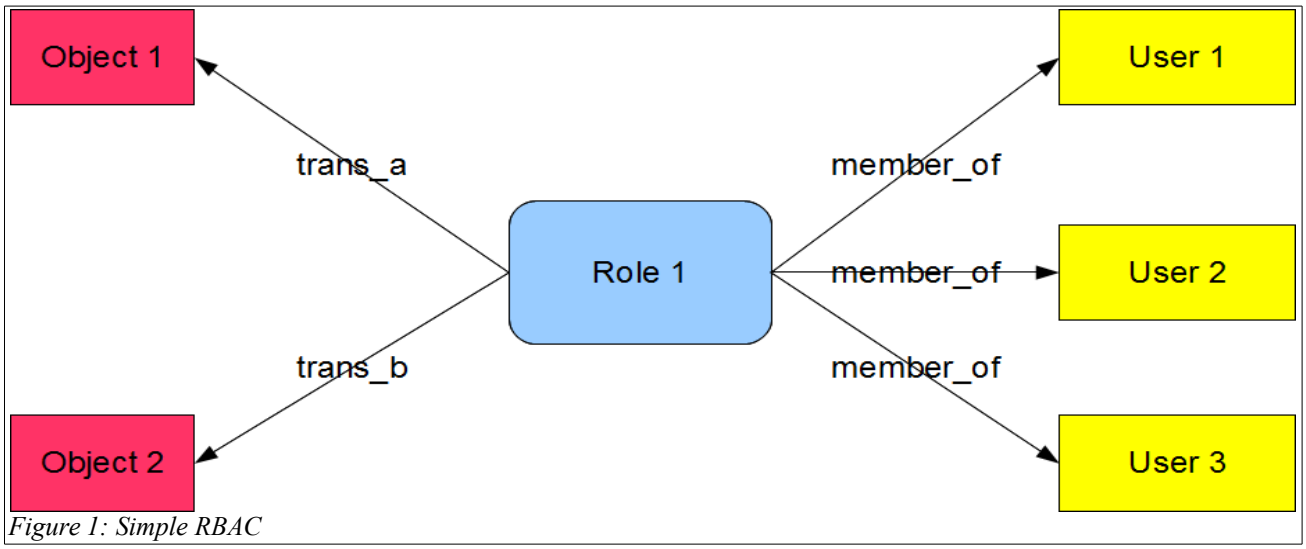

Figure 1: Simple RBAC

Figure 1, redrawn from an image in [27], depicts an example of a simple static RBAC scenario involving a role with several users and objects assigned to it. Generally, only the database administrator is given the power to assign roles to users. This figure shows an abstract assignment of a Role to two Objects, and of 3 Users to the Role.

2.2.2 Extensions to Static RBAC

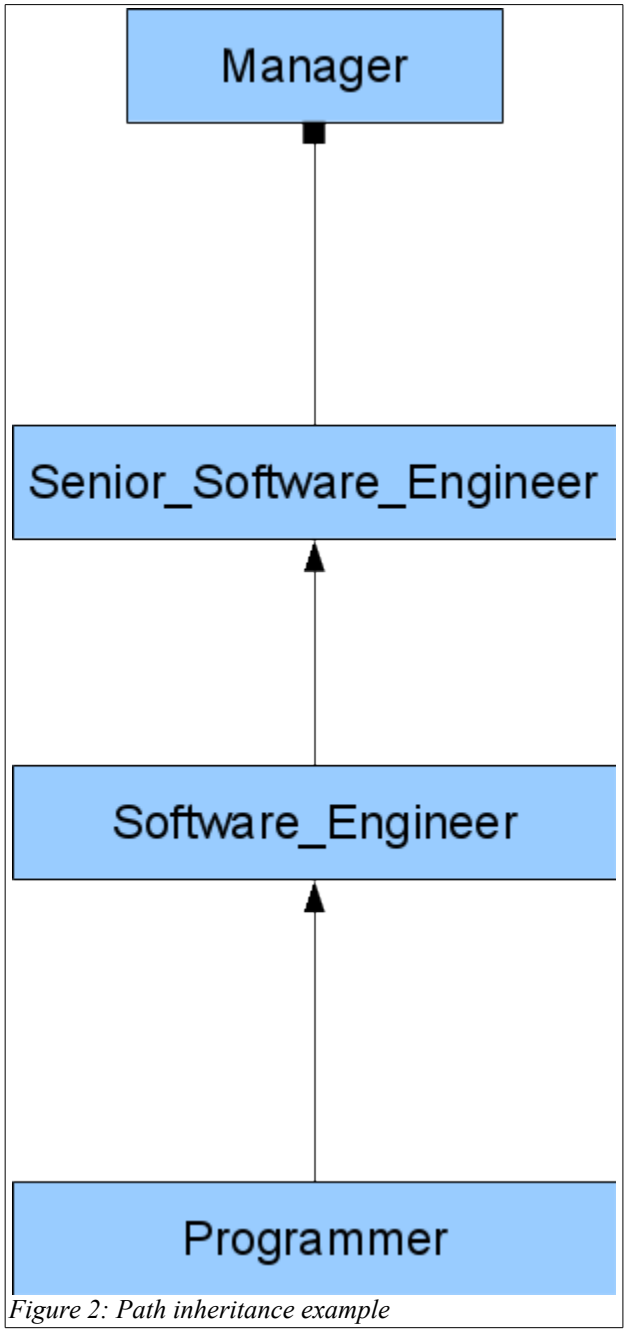

Figure 2: Path inheritance example

```
d_s(programmer,software_engineer).
d_s(software_engineer,senior_ software_engineer).
d_s(senior_ software_engineer,manager).
```

Code 1: An example of a seniority hierarchy written in Prolog.

A widely used model for static RBAC based on predicate logic is that of Barker & Stuckey [18]. Their model is discussed in detail in the next few paragraphs.

Permission refers to a user's right to perform an action. In a hierarchical RBAC model, permissions are inherited up the hierarchy. **Denial** means that a user is prevented from performing an action. **Denials** override permissions. **Seniority** refers to the grouping of roles into a hierarchy. **Permissions** are inherited up the hierarchy, while **denials** are

inherited down the hierarchy. The RBAC model considers when users are online ('**active**'), and only give permissions to currently active users.

While hierarchy in an RBAC system is useful in determining flow of command, it is not always appropriate that people higher up in a hierarchy should inherit the permissions of those below. For example, in a software development setting, managers might not inherit the permissions of senior software engineers to modify programs or technical settings. A seniority hierarchy representing this scenario can be represented as in Code 1.

A **path inheritance** rule specifies that a permission can inherit this far, but no further, in this case only as far as the `senior_software_engineer` level, but not beyond into management grades. Figure 2 illustrates this scenario, where the square arrow pointing to manager indicates that the `manager` does not inherit any privileges from `senior_software_engineer`.

Equal-status roles with slightly different permissions can be defined using **role inclusion**. An example of its application is to define the permissions of employees on different duty rosters (employees on day and night duty have the same permissions, but are only allowed to access data at particular times). An inclusion relationship can be thought of as a relationship between an inner role and an outer role. Thus, if `is_a(inner_role, outer_role)` defines a direct inclusion relationship, then `inner_role` inherits the privileges of `outer_role`.

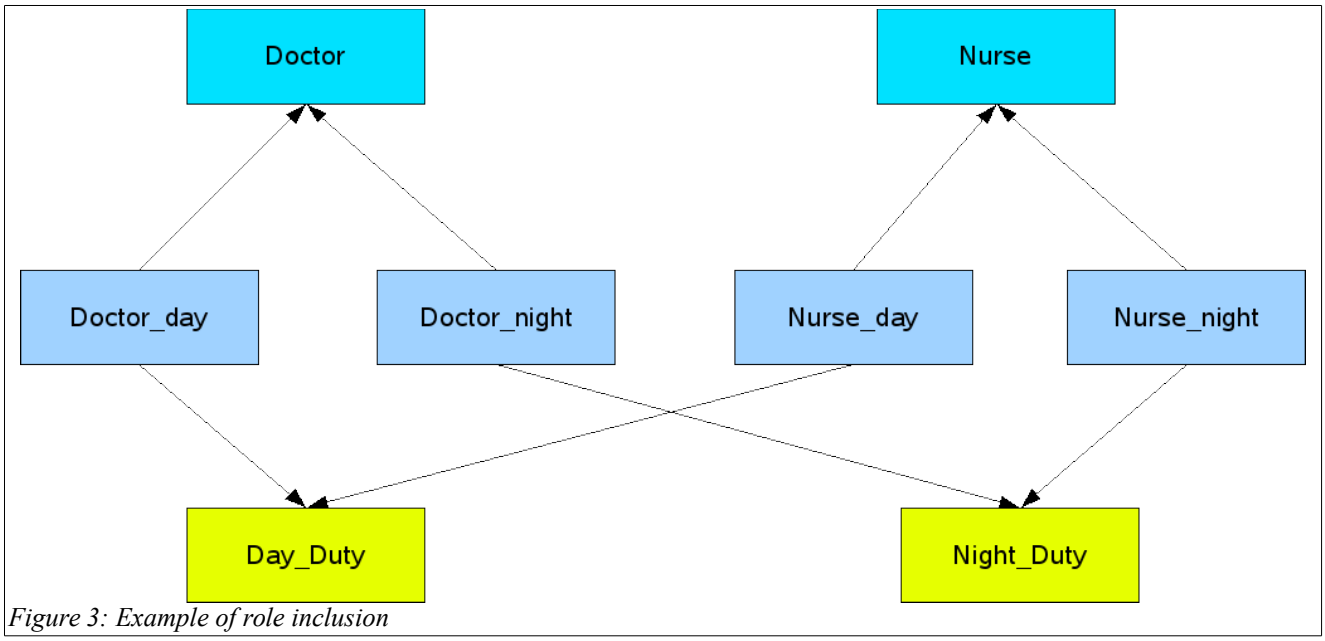

Figure 3: Example of role inclusion

Consider an example of hospital doctors and nurses with day and night duty rosters (Figure 3). Here, the role `doctor_day`, referring to a doctor on day duty, 'is a' doctor, and also 'is a' `day_duty` staff member. The role `doctor_day` thus inherits the privileges and constraints of both roles `doctor` and `day_duty`, but not vice versa. An inclusion hierarchy can also be created, similar to the seniority hierarchy. Generally, users will be assigned to roles within the inclusion hierarchy (inner roles). That is, a user will be assigned as a `doctor_day`, rather than to `doctor` or `day_duty`.

```
is_a(doctor_day, doctor).
is_a(doctor_day, day_duty).
is_a(doctor_night, night_duty).
is_a(nurse_day, nurse).
is_a(nurse_day, day_duty).
is_a(nurse_night, night_duty).
```

Code 2: Role inclusions of Figure 3

Permissions and denials are both inherited *inside* inclusion relationships. Thus, if `is_a(inner_role, outer_role)`, then `inner_role` inherits both the permissions and the denials of `outer_role`. This is in contrast to the seniority hierarchy, where permissions and denials are inherited in opposite directions. Figure 3 shows an example of role inclusion to represent day-duty and night-duty doctors and nurses. This figure represents the facts in Code 2.

Thus, the role `doctor_day`, referring to a doctor on day duty 'is a' `doctor`, and also 'is a' `day_duty` staff member. The role `doctor_day` thus inherits the privileges and constraints of both roles `doctor` and `day_duty`, but not vice versa. Generally, users will be assigned to roles inside the inclusion hierarchy (inner roles). That is, a user will typically be assigned to `doctor_day`, rather than to `doctor` or `day_duty`.

Separation of duties refers to rules whereby users are restricted in the combinations of roles that they can possess. **Static separation of duties** (SSD) means that a user can *never* be assigned to a particular combination of rules. The classic example where this is appropriate is that an employee of an organization cannot be both a purchasing manager and an accounts manager. **Dynamic separation of duties** (DSD) means that a user can have a combination of permissions, but cannot be activated for both at the same time. Using the above example, a user might be an accounts manager for one department, and purchasing manager for another. To prevent conflict, the user cannot be active as both roles at once.

Static RBAC is easy to implement in most standard off-the-shelf relational DBMSs, including Oracle (version 7 onwards) [44] and Postgres (v8.*x* onwards) [45], which have native support for roles (**`CREATE ROLE`**). However, these do not implement all the features described above.

2.3 Dynamic and context-aware RBAC

Barker & Douglas [19] have implemented an RBAC system for protecting federated DBMSs. This includes one context element, namely checking a user's IP address. However, their system is implemented in Java, which while having the advantage of being DBMS-independent, is inefficient and has an additional layer between the database and the application. A more efficient method would be to use the DBMS own features to define the context-aware RBAC model. These include procedural languages, such as PL/SQL in Oracle. While dynamic RBAC can be implemented in certain standard off-the-shelf database management systems, the implementation is often complex.

Traditional static RBAC is difficult to apply in context-aware applications, since it fixes a user's access privileges when the user logs on. In dynamic RBAC, the access rules are determined at run-time when a user attempts to access data. These may be based on time of access, or specific values in the data being accessed, or environmental factors such as ambient temperature or location. [21]

Various alternatives and extensions to RBAC have been proposed to provide context-aware access control, in which rules are enforced according to runtime parameters. [46] The basic RBAC model can be extended to take account of contexts, by varying the range of active roles, the roles in which users may be active or assigned, or the permissions

that are assigned to roles, according to context. In other words, in which user-role assignments or role-permission assignments can be changed dynamically during program run-time, rather than statically when a user logs on. This is called context-aware RBAC, or Dynamic RBAC (DRBAC). Factors that may cause changes in the RBAC assignments include time of day, ambient temperature and user location. Users may also be restricted to accessing particular rows in a database table, rather than necessarily being given access to an entire table. These include location-based access control system such as M-Zones Access Control [47] and GEO-RBAC [48], as well as Temporal RBAC (TRBAC).[49] TRBAC has been further extended as Generalized T-RBAC (GTRBAC) to incorporate hierarchy and separation of duties. [50] Another extended RBAC model is the OASIS (Open Architecture for Securely Interworking Services) model [51][52] for RBAC in heterogeneous data sources. Belokosztololszky *et al.* [53] propose a mechanism for using defining parameters used by RBAC models as contexts.

Tolone *et al.* compared and contrasted the applicability of various access control models for collaborative systems. [54] These are DAC (called the Access Matrix Model in their paper), static RBAC, TBAC (Task-Based Access Control), TMAC (Team-Based Access Control), Spatial Access Control and Dynamic RBAC (called Context-Aware Access Control). They found that context-aware RBAC provides the best support for such applications, but also that it is the most complex.

Corradi *et al.* [55] have designed a context-based access control system where "privileges are directly associated with contexts, and users acquire their proper set of permissions dependently on their current contexts." This approach appears to regard context awareness as a substitute for roles.

The proposed RBAC system has dynamic features. The dynamic RBAC features can be divided into two types: temporal and row-level. In TRBAC, access to a resource depends on the time when it is accessed. In row-level RBAC, particular staff can only access particular rows in the table depending on some formulation. Context constraints filter down the seniority hierarchy, and inside the inclusion hierarchy. TRBAC has been successfully implemented in Oracle 8*i*. [49]

Environment roles have been proposed as a way of modelling environmental factors by specifying each one as a role [56].

The extension of roles beyond users to environments has been proposed [56]. Roles are used to capture environment conditions. Whereas a user role is 'active' when a user assigned to that role is logged in, an environment role is active when a particular set of environment conditions are true. For instance, an environment role for `office_hours` can be set up, which is automatically activated at 09:00 and deactivated at 17:00. Other environment roles can be related to factors such as ambient temperature and locations. Permissions assigned to user roles may only be valid if particular environment roles are also active.

```
rpa(child,use,intercom) ←
  active(weekday),
  active(free_time)
  ;
  active(weekend).
```

Code 3: Example of a dynamic RBAC rule.

For example, the predicate logic code in Code 3 can be used to determine whether a child can use an intercom in a home. The rule states that a user with the role `child` can use the intercom during their free time at weekdays, and at

any time at weekends. In this case, `weekday`, `free_time` and `weekend` are environment roles, which are defined as active in particular conditions using additional predicates.

From this, it can be noted that a security model can be applied to access control for resources of any kind, such as access to buildings or rooms, or permission to use particular computers or systems, not only to access to data.

Although this is not specifically mentioned in the paper, environment roles could also be used to capture environment conditions intrinsic to a particular person. For example, this indicates the validity of the rule `rpa(nurse, read, patient)` (a user with role nurse can read a patient's file).

Here, `P` is a patient, and the predicate states that the `rpa` rule is valid for patient `P` if the `high_heartbeat` role is active for `P`. Additionally, role hierarchies can be applied to environment roles in an analogous fashion to user roles, as can static and dynamic separation of duties.

Extending this concept further, GRBAC (Generalized RBAC) [57] treats not only environment conditions, but also objects, as roles. The treatment of objects as roles, when applied to a hierarchical RBAC system, makes sense in object-oriented databases, but perhaps less applicable to relational database systems, where database objects are not organized hierarchically.

```
rpa(nurse,read,patient(P)) ←
  active(high_heartbeat, P).
```

Code 4: Another example of a dynamic RBAC rule.

An alternative method of modelling contexts is to define context constraints and assigning these to user roles, as described by Strembeck & Neumann [20][21], who also devised a series of predicates to aid in modelling.

This model can be represented in predicate logic using an example in Code 4, based on the model in [21].

A dynamic RBAC system based on this model has been implemented in Oracle 10g using its row-level access control feature [58]. We propose to extend this work, to implement more complex context-aware access control models using DBMSs, so that access to data from database-driven applications can be controlled directly by the database according to the usernames entered by the application user, without the need for any additional middleware or application-level access control.

Applying context-aware access control models to DBMSs involves capturing the environmental conditions, and using the DBMS security features to ensure that only the appropriate level of access is achieved by any user, based on the rules determined by the environmental conditions. An extension to RBAC is the ARBAC97 (Administrative RBAC 1997) model that brings in the possibility of "using RBAC itself to manage RBAC" [33] by allowing users who are members of administrative roles to assign (and revoke) users and permissions to (and from) roles. TMAC [59] and TBAC [60] both model access control from a context-oriented perspective. However, it has been argued that such an approach does not bring anything new from the RBAC perspectives in medical database security [7]. TRBAC restricts the roles available to users depending on the time period, by enabling or disabling roles using role triggers at specified times. TRBAC has been implemented using database triggers on an Oracle database [49].

```
context_clause(patient_consulted_by_doctor,Doctor_ID,Patient_ID) ←
  ae_consultation(_,_,_,Patient_ID,Doctor_ID).

context_clause(patient_diagnosed_by_doctor,Doctor_ID,Patient_ID) ←
  ae_consultation(Cons_Number,_,_,Patient_ID,_),
  patient_diagnosis(_,Doctor_ID,_,Cons_Number,_).

context_constraint(patient_treated_by_doctor,Doctor_ID,P,
 patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)) ←
  context_clause(patient_consulted_by_doctor,Doctor_ID,Patient_ID)
  ;
  context_clause(patient_diagnosed_by_doctor,Doctor_ID,Patient_ID).
```

Code 5: Context clauses and constraints with predicate logic.

A context hierarchy can be devised by defining context constraints in terms of other context constraints. Hu & Weaver [46] suggest how this can be done systematically by defining context conditions, clauses and context constraints as in Code 5.

Context Constraint := Clause1 ∪ Clause2 ∪ … ∪ Clause3
Clause := Condition1 ∩ Condition2 ∩ … ∩ Condition3
Condition := <CT> <OP> <VALUE>

Formula 1: Mathematical definition of context constraints and conditions.

In Formula 1, <CT> is a context parameter, <OP> is a comparison operator, and <VALUE> is a parameter value for comparison. Thus, an example of a context condition might be Temperature ≥ 25.

Bertino *et al.* [61] proposed a general framework for reasoning about access control models using C-Datalog [62], which is an object-oriented extension of Datalog [63]. Their framework is "general enough to model discretionary, mandatory, and role-based access control models". It can model static and dynamic RBAC, and the object-oriented nature of the C-Datalog language allows hierarchical RBAC to be modelled more simply in their model than in models using traditional predicate logic languages such as Prolog and Datalog.

Seitz *et al.* [64] proposed an access management system, called Semantic Access Certificates, for use in grid environments, and illustrate its use in a medical environment. This system extends RBAC by enabling access control based not only on the role of the users wishing to access the data, but also the semantics of the data. In this sense, it acts like dynamic RBAC.

3 RBAC Implementation in Prolog and Relational DBMS

3.1 Introduction

In this section we examine the feasibility of implementing, in a relational DBMS, a dynamic RBAC model for a hospital database originally written in Prolog. The static RBAC model is first described, followed by its implementation in Oracle. The dynamic RBAC is then described in the same way.

The dynamic RBAC is built upon the static RBAC model of Barker & Stuckey [18], and has dynamic constraints separately defined through their context constraints. Table 28 (Appendix VII, page 219) shows the roles of Hospital staff and the permissions assigned to them as defined in the static RBAC model. The static RBAC schema is reused from earlier research prototypes where we experimented with solutions for data sharing across the NHS [25]. The dynamic RBAC design is built upon the semantics of the static RBAC and extended by Strembeck & Neumann [21].The DRBAC model was written in SWI-Prolog [14], a free/*libre* implementation of Prolog using the Edinburgh syntax. The model requires SWI-Prolog v5.6.17 or above to run, due to the use of date and time handling syntax only available in the most recent versions. We have implemented a hierarchical DRBAC model in which denials override permissions. Additionally, the proposed DRBAC model has the following extended features discussed in [18]: separation of duties, inheritance paths and role inclusion.

Table 1: Tables for hospital database used in testing

Table	***Description***
Ward	Hospital wards.
Room	Rooms within hospital wards.
Bed	Beds within rooms in wards.
Patient	Patient demographic details.
Diagnosis	List of coded diagnoses.
AE_Consultation	Consultations by doctors with patients. Links to RBAC table usr for doctor.
Patient_Diagnosis	Diagnoses given during consultations. Links to RBAC table usr for doctor performing diagnosis.
Nurse_Ward	Assignments of nurses to Wards. Links to RBAC table usr for nurse.

Ward(Ward_ID, Type, Ward_Capacity)
Room(Room_ID, *Ward_ID*, Type, Bed_Capacity)
Bed(Bed_ID, *Room_ID*, Type)
Patient(Patient_ID, Last_Name, First_Name, Address, DOB, *Bed_ID*)
Diagnosis(Diagnosis_code, Illness_name, Usual_Symptoms)
AE_Consultation(Cons_Number, Cons_Date, Cons_Description, *Patient_ID*, Doctor_ID)
Patient_Diagnosis(Patient_Diagnosis_Number, Diagnosing_Doctor, Diagnosis_Desc, Cons_Number, Diagnosis_Code)
Nurse_Ward(usr, *Ward*)

Text 1: Hospital Database Schema

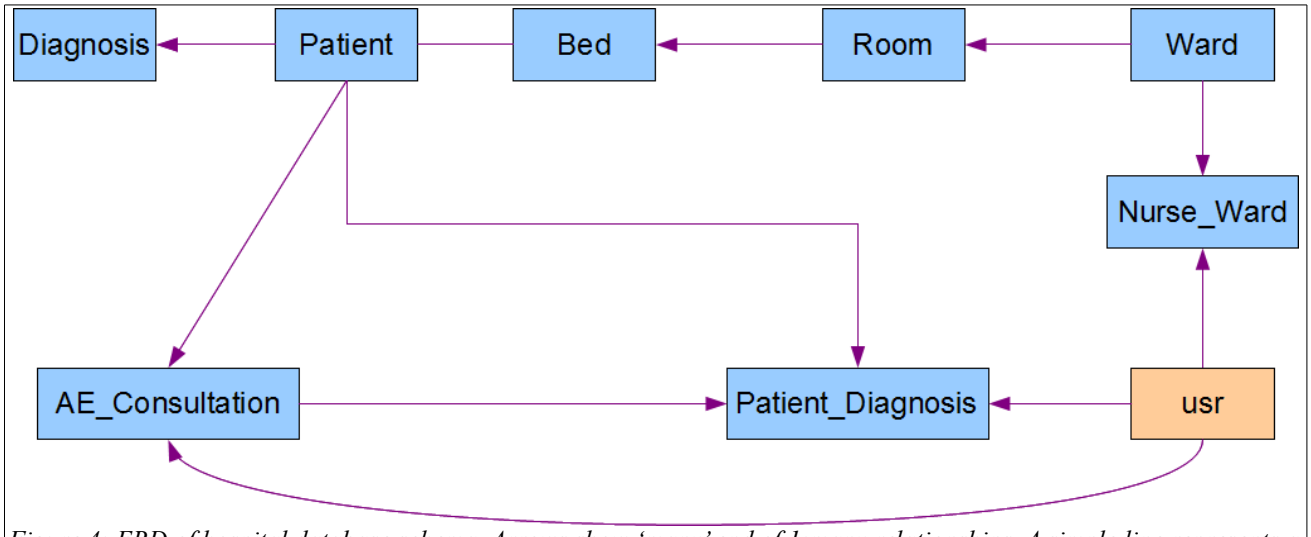

Figure 4: ERD of hospital database schema. Arrows show 'many' end of 1:many relationships. A simple line represents a 1:1 relationship.

The hierarchical RBAC is modelled in this section using a database and RBAC model that could be applied to a hospital scenario. The data model includes basic information about Patients hospital Beds. There are a number of beds in each Room, and a number of Rooms in each Ward. A ward may be looked after by one more Nurses. A Patient may be diagnosed during a Consultation with a Doctor, according to a specified list of diagnosis. Table 1 shows the database tables of a hospital database that is used to model RBAC in the system. Note that the information about Doctors and Nurses is stored in an RBAC table, usr, to which the tables AE_Consultation, Patient_Diagnosis and Nurse_Ward link. Therefore, no table in Table 1 lists either Doctors or Nurses. Text 2 shows the schema. Figure 4 shows the ERD (Entity Relationship Diagram).

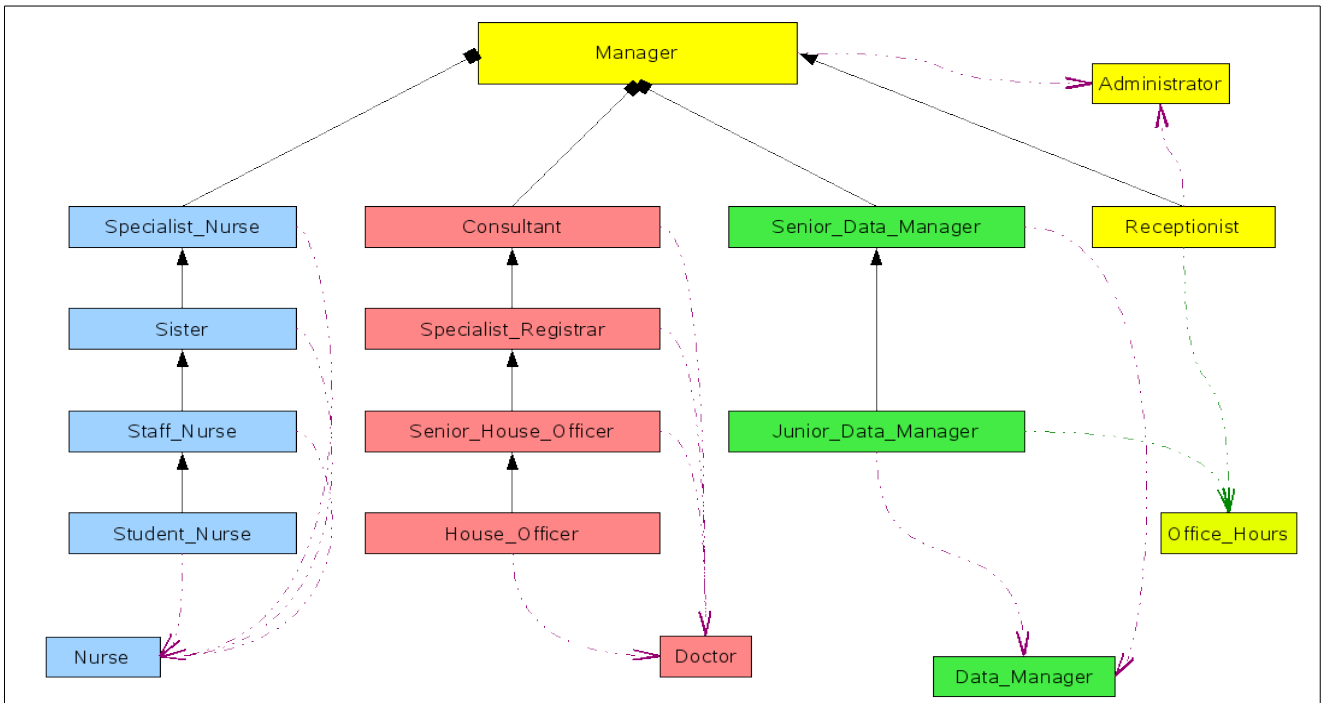

Figure 5: Role hierarchy in Hospital database, excluding `day_duty` and `night_duty` in doctor and nurse roles. Solid lines show `d_s` relationships; dotted lines show `is_a` relationships.

Figure 5 shows the seniority hierarchy of the RBAC model used in this section. This model has four role hierarchies, for doctors (shown in red), nurses (blue), data managers (green) and administrators (yellow). Figure 115 (Appendix VI, page 217) shows both the seniority and inclusion relationships for doctor and nurse type roles. Table 28 lists the roles in this RBAC model, and permissions assigned to them. The ERD of RBAC tables (relations that are used to store RBAC meta-data) is shown in Figure 116 (Appendix, page 217) shows both the seniority and inclusion relationships for doctor and nurse type roles. Table 28 (Appendix, page 217) lists the roles in the model, and the permissions assigned to them.In the following sections, Prolog code is depicted in a `light typewriter font`, while SQL and PL/SQL code is depicted in a **`heavy typewriter font`**.

3.2 Defining and Implementing Static RBAC in Relational Database

3.2.1 Representation of Static RBAC Model in Prolog

Table 2: Fact definitions used in RBAC design in Prolog.

Fact Formula	***Description***	***Example***	
`d_s(SeniorRole, JuniorRole).`	defines a direct seniority relationship: `SeniorRole` is directly senior to `JuniorRole`	`d_s(consultant, specialist_registrar).`	
`inherits_rpa_path(SeniorRole, JuniorRole, Permission, Object).`	Privileges represented by `Permission` for object `Object` are inherited up the role hierarchy from `JuniorRole` to `SeniorRole`, and no further.	`inherits_rpa_path(senior_data_manager, junior_data_manager, _,_).`	In this example, `Permission` and `Object` are set using the Prolog anonymous variable _, meaning that this inheritance path from `senior_data_manager` to `junior_data_manager` applies to all values of `Permission` and `Object`.
`is_a(InnerRole, OuterRole).`	Direct inclusion relationships.	`is_a(student_nurse_night,student_nurse).`	
`user(Username, LastName, FirstName, Address, DOB).`	Defines a user's personal details.	`user(u0001,'Sugar', 'Ed','1 Montgomery Ave','12/06/1975').`	
`password(Username, Password).`	User passwords. In the Prolog implementation, the passwords are unencrypted, but in a real implementation they would obviously encrypted.	`password(u0001,'desk').`	
`role(Role).`	Defines the role named `Role`	`role(consultant).`	
`object(Object).`	Defines the object named `Object`, and includes its full data structure.	`object(room(Room_ID, Ward_ID,Type,Bed_Capacity)).`	
`rpa(Role, Permission, Object).`	Role Permission Assignment: `Role` is assigned `Permission` on `Object`.	`rpa(house_officer, select,ward(Ward_ID, Type,Ward_Capacity)).`	
`ura(User,Role).`	User Role Assignment: `User` is assigned to `Role`.	`ura(u0017, senior_data_manager).`	

Fact Formula	*Description*	*Example*	
`activate(User, Role,Time, Password).`	`User` is activated as `Role` at `Time` with `Password`.	`activate(u0005,house_officer_day, date(2006, 8, 23, 12, 39, 0, -3600, 'BST', true),'mother')`	This means that user `u0005` signed on as role `house_officer_day` at 12:39:00 BST on 23 August 2006 with password `mother`. Note that the date is expressed using a SWI-Prolog `date` constructor.
`deactivate(User, Role,Time)`	`User` is deactivated as `Role` at `Time`.	`deactivate(u0005, house_officer_day, date(2006, 8, 23, 12, 50, 0, -3600, 'BST', true))`	

Assignments of users to roles, and roles to permissions, are represented as Prolog facts and Prolog rules. Table 2 lists the syntax of the Prolog facts used in this static RBAC model. Notice that an assertion of fact in Prolog takes the form `relation(term`$_1$`,term`$_2$`,…,term`$_k$`)`. An initial capital letter means that the term is a variable; otherwise, it is a constant atom.Appendix IV (page 206) lists all facts defined in the Scenario used for testing this Prolog model.

Table 3: Rules in Prolog static RBAC design

Rule Name	*Description*
`permittable`	Privileges assigned to users.
`authorizable`	Privileges assigned to users, filtered by denials.
`permitted`	Privileges assigned to currently active users.
`authorized`	Privileges assigned to currently active users, filtered by denials.
`included_in`	All inclusion relationships.
`senior_to`	All seniority relationships.
`inherits_rpa`	All pairs of roles linked by an inheritance path.
`rpa_full`	Explicit and implicit (by seniority) permissions given to roles.
`d_rpa_full`	Explicit and implicit (by seniority) denials given to roles.
`denied`	Denials assigned to users.
`currently_active`	Users with current open sessions.
`dsd_conflict`	All pairs of roles that produce a DSD conflict.
`ssd_conflict`	All pairs of roles that produce an SSD conflict.
`inconsistent_ssd`	Whether a role violates an SSD rule.
`inconsistent_dsd`	Whether a role violates a DSD rule.

Prolog rules are used to deduce who can access what. A Prolog rule takes the form `result :- condition`, where the result is true if the condition is true. Table 3 lists all rules used in the static RBAC Prolog model. The complete Prolog code for the Prolog rules describing the full static RBAC meta-model is given in Appendix II. These are discussed in detail here.

```
% inclusion of equal-status roles
    i.  included_in(R1,R1).
    ii. included_in(R1,R2) :- is_a(R1,R2).
    iii.       included_in(R1,R3) :- is_a(R1,R2),
                       included_in(R2,R3).
```

Code 6: included_in

Code 6 shows the `included_in` rules, for deducing role inclusions. Predicate i in Code 6 states that any role is included in itself. Predicate ii states that `R1` is included in `R2` if `R1` is directly defined as a type of `R2` by an `is_a` fact.

Predicate iii states that `R1` is included in `R3` if:–

a) `R1` is directly defined as a type of `R2`, and

b) `R2` is included in `R3`.

Note that the comma (`,`) signifies logical "AND" in Prolog. Predicate iii is recursive, because the condition clause also contains `included_in`.

Note that the three predicates are independent, so, for example variable `R1` in i has no connection with `R1` in ii.

```
    i.  senior_to(R1,R1) :- d_s(R1,_).
    ii. senior_to(R1,R1) :- d_s(_,R1).
    iii.       senior_to(R1,R2) :- d_s(R1,R2).
    iv. senior_to(R1,R2) :- d_s(R1,R3), senior_to(R3,R2).
```

Code 7: senior_to roles.

The rules for `senior_to` are shown in Code 7. These are defined similarly to `included_in`. However, a role is automatically considered to be "senior to" itself. For it to be defined as such, it must be either directly senior to another role (as in Predicate i) or have a role directly senior to it (Predicate ii). If a role participates in a seniority hierarchy, then it is "senior to" itself. [This means that senior_to is really "senior to or equal".] Predicates iii and iv are analogous to Predicates ii and iii in Code 6 for `included_in`.

```
inherits_rpa(R1,R1,_,_).
inherits_rpa(R2,R3,P,O) :- senior_to(R1,R2),
                           senior_to(R3,R4),
                           inherits_rpa_path(R1,R4,P,O).
```

Code 8: inherits_rpa

Code 8 shows the predicates for `inherits_rpa`, which determines how far along a seniority hierarchy access privileges can be inherited.

```
rpa_full(R1,P,O) :- included_in(R1,R2),
                    senior_to(R2,R3),
                    rpa(R3,P,O),
                    inherits_rpa(R2,R3,P,O).
```

Code 9: rpa_full

Code 9 shows the rule, `rpa_full`, which determines the entire set of permissions that a particular role has, whether explicit or implicit. It states the `R1` has permission `P` over object `O` if:–

a) `R1` is a type of (included in) role `R2;`

b) `R2` is senior to `R3;`

c) `R3` has permission `P` over object `O`, and

d) `R2` and `R3` are part of an inheritance path.

```
permittable(U,P,O) :- permittable(U,P,O,R).
permittable(U,P,O,R) :- ura(U,R)
                        rpa_full(R,P,O).
```

Code 10: definition of `permittable`.

Code 10 shows the definition of the Prolog rule `permittable`, which determines whether a user would have permission to access an object if they were active in their role.

```
permitted(U,P,O) :-    ura(U,R),
                       permitted(U,P,O,R).
permitted(U,P,O,R) :-  currently_active(U,R,_),
                       permittable(U,P,O,R).
```

Code 11: definition of `permitted`.

Code 11 shows the definition of the Prolog rule `permitted`, which determines whether a user does have permission to access an object. `permitted` is true if `permittable` is true, and the user is `currently_active` in the role.

```
d_rpa_full(R1,P,O) :- included_in(R1,R2),
                      senior_to(R3,R2),
                      d_rpa(R3,P,O).
```

Code 12: `d_rpa_full`

Code 12 shows the definition of the Prolog rule `d_rpa_full`, which is analogous to `rpa_full` for denials. Note the following differences from `rpa_full`.

a) The `included_in` term is the same, but in the arguments of the `senior_to` clause are reversed. All access control rules, whether permissions or denials, are inherited from an outer (parent) role to an inner role. However, whereas permissions are inherited up the seniority hierarchy, denials are inherited down it.

b) the rule for `d_rpa_full` does not consider `inherits_rpa`: in this model, denials are always inherited all the way down a seniority hierarchy.

```
denied(U,P,O) :- ura(U,R),
                 d_rpa_full(R,P,O).
```

Code 13: `denied`

Code 13 shows the definition of the Prolog rule, which determines whether a user is denied access. This is defined simply as a user being a member of a role that it part of a `d_rpa_full`. It is analogous to both

`permittable` and `permitted`: there is no concept of being denied access depending on whether a user is active in a role. Either the user is denied access to an object, or is not.

```
authorizable(U,P,O) :-   ura(U,R),
                         authorizable(U,P,O,R).
authorizable(U,P,O,R) :- permittable(U,P,O,R),
                         not(denied(U,P,O)).
```

Code 14: `authorizable`

```
authorized(U,P,O) :-   ura(U,R),
                       authorized(U,P,O,R).
authorized(U,P,O,R) :- permitted(U,P,O,R),
                       not(denied(U,P,O)).
```

Code 15: `authorized`

Code 14 shows the definition of the Prolog rule `authorizable`, which is defined as "`permittable` but not `denied`". Code 15 shows the definition of the Prolog rule authorized, which is defined as "permitted but not denied".

3.2.2 Transformation of Static RBAC Model from Prolog to SQL Database

The Prolog representation described in Section 3.2.1 above, and in Appendix IV and Appendix III, was transformed into a representation in an SQL database. That is, a relational database model was created for holding data about assignments of users to roles and roles to permissions, so that permissions and denials of particular users could be inferred using SQL queries on this database model. This model was stored in a database schema separate from the main data tables.

A Prolog fact can be transformed in one of three ways.

(1) ***Transformation into* `INSERT` *statements on database tables (most Prolog facts).***

(2) ***Transformation into* `CREATE TABLE` *statements to create tables in the main data schema (Prolog facts* `object`***).***

(3) ***Transformation into rows of table* `usr_session` *in the RBAC schema (Prolog facts* `activate` *and* `deactivate`***).***

```
CREATE TABLE d_s (
  senior_role  VARCHAR(64) NOT NULL,
  junior_role  VARCHAR(64) NOT NULL,
    Primary Key (senior_role,junior_role),
    FOREIGN KEY (senior_role) REFERENCES role(role),
    FOREIGN KEY (junior_role) REFERENCES role(role)
);
```

*Code 16: **`CREATE TABLE`** statement for **`d_s`** table in SQL.*

In general, ***Prolog facts are transformed into* `INSERT` *statements on database tables (method (1)).*** The conversion was generally straightforward. For example, Code 16 shows the table definition of the **`d_s`** table corresponding to the `d_s` Prolog fact.

```
INSERT INTO d_s( senior_role, junior_role )
VALUES ( 'consultant','specialist_registrar' );
```

Code 17: INSERT statement for a `d_s` fact

Code 17 shows the **`INSERT`** statement equivalent to `d_s(consultant,specialist_registrar).`. Like all `d_s` facts, this takes the form `d_s(Senior_role, Junior_role)` (Table 2, page 31).

In some cases, the transformation is more complex due to the different ways in which SWI-Prolog and Oracle represent dates and times. Additionally, there are differences in structure between the Prolog facts and Oracle tables.

```
CREATE TABLE room
(
room_id      VARCHAR(10),
ward_id      VARCHAR(10),
type         VARCHAR(10),
bed_capacity VARCHAR(10),
primary key (room_id),
Foreign Key (ward_id) references ward(ward_id)
);
```

*Code 18: **CREATE TABLE** statement for **room**.*

```
CREATE TABLE usr_session(
  usr VARCHAR(16) NOT NULL,
  role VARCHAR(64) NOT NULL,
  start_time TIMESTAMP NOT NULL,
  end_time TIMESTAMP,
    FOREIGN KEY (usr) REFERENCES usr(user_id),
    FOREIGN KEY (role) REFERENCES role(role)
);
```

*Code 19: **CREATE TABLE** statement for **usr_session** table.*

The `object` facts (e.g. `object(room(Room_ID,Ward_ID,Type,Bed_Capacity)).`) are represented as **CREATE TABLE** statements in the database schema for the main data (not in the RBAC schema) (method (2)). This is because they define the object types existing in the data model of the data being accessed by RBAC, and thus represent database tables to which the RBAC model grants access. Therefore, they. For example, this `object` fact is represented by the **CREATE TABLE** statement in Code 18.

Prolog facts `activate` and `deactivate` ***are transformed into rows of table*** **usr_session** (method (3)). This table does not store the password. Code 19 shows the structure of the **usr_session** table.

Unlike in Prolog, where logging on (activation) and logging off (deactivation) are represented by separate `activate` and `deactivate` Prolog facts, the SQL table represents an activation and corresponding deactivation by an entry in **usr_session**. The absence of a value for **end_time** in **usr_session** corresponds to the lack of a `deactivate` fact corresponding to an `activate` fact, i.e. a currently active session. Unlike Prolog, SQL can validate a data record before entering it, and can modify and delete records. Therefore, there is no need to store sessions with incorrect passwords, only for them to be rejected by the `currently_active` rule. Instead, attempts to **INSERT** rows from logins with incorrect passwords can be rejected using a database trigger. Similarly, rather than representing the end of a session by asserting a `deactivate` fact, it can be represented by **UPDATE**ing the row in the RBAC database table **usr_session** with the **end_date**.

```
usr    role                    start_time             end_time
------ --------------------  ---------------------  ---------------------

u0005  house_officer_day      2006-08-23 12:23:39     NULL
```

*Text 2: Row in **usr_session** corresponding to `activate(u0005,house_officer_day,date(2006, 8, 23, 12, 39, 0, -3600, 'BST', true),'mother').` with no corresponding `deactivate` fact.*

For example, the fact

```
activate(u0005,house_officer_day,date(2006, 8, 23, 12, 39, 0, -3600, 'BST', true),'mother').
```

is represented in the SQL model as a row in **usr_session**, given in Text 2.

```
usr    role                    start_time             end_time
------ --------------------  ---------------------  ---------------------

u0005  house_officer_day      2006-08-23 12:23:39     2006-08-23 12:23:50
```

*Text 3: Row in **usr_session** corresponding to `activate(u0005,house_officer_day,date(2006, 8, 23, 12, 39, 0, -3600, 'BST', true),'mother').` with `deactivate(u0005,house_officer_day,date(2006, 8, 23, 12, 55, 0, -3600, 'BST', true).`*

If a `deactivate` fact

```
deactivate(u0005,house_officer_day,date(2006, 8, 23, 12, 55, 0, -3600, 'BST', true).
```

is added, then the equivalent in the SQL model is to update the row in Text 2 with a value for end_time, giving the row in Text 3.

Text 57 (page 219) shows the RBAC schema. Figure 116 (page 217) shows the ERD of the RBAC schema.

```
INSERT INTO role( role ) VALUES ( '_' );
```

*Code 20: **INSERT** statement for Dummy role.*

After creating the table **role**, a dummy record is added to the table, with value **_** (Code 20). This is needed as an equivalent of the Prolog anonymous variable _, as used in some Prolog rules.

Prolog rules were transformed in either of two ways:

(1) ***Non-recursive Prolog rules were transformed into SQL views.*** However, `currently_active` is a special case.

(2) ***Recursive Prolog rules were transformed into SQL tables populated by triggers and stored procedures.***

```
CREATE VIEW permittable AS
-- permittable(U,P,O) :- ura(U,R),
--                       permittable(U,P,O,R).
-- permittable(U,P,O,R) :- rpa_full(R,P,O).
  SELECT DISTINCT usr, object, action, ura.role AS role FROM ura,
rpa_full
    WHERE ura.role = rpa_full.role;
```

*Code 21: SQL view for **permittable** (including the Prolog code on which it is based as a comment).*

Most Prolog rules were converted into SQL views (method (1)). Code 21 shows an example for **inherits_rpa**, including the original code for the Prolog rule as SQL comments.

```
CREATE VIEW rpa_full AS -- all permissions to all roles, both explicit and implicit (by inheritance)
-- rpa_full(R1,P,O) :- included_in(R1,R2),
--                     senior_to(R2,R3),
--                    rpa(R3,P,O),
--                    inherits_rpa(R2,R3,P,O).
  SELECT DISTINCT included_in.inner_role AS role, action, object, senior_role, junior_role FROM rpa,
included_in, senior_to
    WHERE included_in.outer_role = senior_to.senior_role
    AND senior_to.junior_role = rpa.role
    AND (
      (senior_to.senior_role,senior_to.junior_role) IN
        (SELECT senior_role,junior_role FROM inherits_rpa WHERE action = '_' AND object = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,action) IN
        (SELECT senior_role,junior_role,action FROM inherits_rpa WHERE object = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,object) IN
        (SELECT senior_role,junior_role,object FROM inherits_rpa WHERE action = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,action,object) IN
        (SELECT senior_role,junior_role,action,object FROM inherits_rpa)
    )
;
```

*Code 22: SQL view for **rpa_full**.*

Querying **permittable** involves querying **ura** and **rpa_full**. This is represented in SQL by a view joining the tables **ura** and **rpa_full**. However, some transformations are more complex, as shown in Code 22 for **rpa_full**.

The view for **rpa_full** is created by joining **rpa**, **included_in**, **senior_to** and **inherits_rpa**, which is itself based on **inherits_rpa_path** as well as **senior_to**. The `inherits_rpa_path` facts in Prolog contain anonymous variables, represented by the underscore, which match any value. These were imported directly into the equivalent SQL table **inherits_rpa** as columns with the value **'_'**. However, this has no special meaning in SQL, which has no equivalent concept. Therefore, the meaning of the underscores as 'match-all' values has to be explicitly defined for the columns where they might appear. This technique is also employed in **ssd_conflict** and **dsd_conflict**. Due to the use of foreign key references to **role** in the relevant tables, the dummy record in **role**, shown in Code 20, is needed.

```
currently_active(U,R1,D1) :- activate(U,R1,D1,Password),
                    password(U,Password),
                    ura(U,R1),
                    (
                      not(deactivate(U,R1,_));
                      deactivate(U,R1,D2),
                      date_time_stamp(D1,T1),
                      date_time_stamp(D2,T2),
                      T2 < T1
                    ).
```

Code 23: Prolog rule for `currently_active`.

The SQL view **currently_active** is defined entirely differently from the equivalent Prolog rule, due to the different representation of the Prolog facts on which it is based (`activate` and `deactivate`). Code 23 shows the `currently_active` Prolog rule.

This rule tests whether a user `U` is currently active in a role `R1` at a given date/time `D1`, in the following steps:

1. Check whether an `activate` fact `activate(U,R1,D1,Password)` exists.
2. Verify that user `U` has password `Password`.
3. Verify that user `U` is assigned to role `R1`.
4. Check that `U` has not been deactivated before `D1` (no corresponding `deactivate` fact exists for `U` in role `R1`).

```
CREATE VIEW currently_active AS
-- currently_active(U,R1,D1)
  SELECT DISTINCT usr, role, start_time FROM usr_session
  WHERE usr_session.start_time < SYSTIMESTAMP
  AND (usr_session.end_time > SYSTIMESTAMP or usr_session.end_time is null);
```

*Code 24: SQL view for **rpa_full**.*

The SQL view **currently_active** (Code 24) is based only on **usr_session**, and checks whether the current time is between **start_time** and **end_time**. The password, role assignment and DSD checking are performed by triggers on **usr_session**.

Some rules in the RBAC model are recursive. These cannot be handled by SQL views, since SQL is not Turing-complete, so cannot handle recursion or iteration. Instead, they are handled by triggers on the underlying tables, which populate a table with the rows that would be produced by the rule.

```
CREATE OR REPLACE TRIGGER is_a_after_all_sl
AFTER INSERT OR UPDATE OR DELETE ON is_a
BEGIN
    insert_included_in();
END;
```

Code 25: row-level post-action trigger on table ***is_a***

The two recursive rules in this RBAC model are `included_in` and `senior_to`, which use classic parent/ancestor recursion. The following text uses `included_in` as an example of transforming a recursive Prolog rule to SQL. The transformation of `senior_to` is similar, and is not shown. The Prolog rule `included_in` is shown in Code 12 (page 34). `included_in` is deduced by querying on `is_a`, and if required by recoursing to `included_in`. In the database RBAC model, **included_in** is a table, populated by a row-level post-action trigger on **is_a** as in Code 25.

```
INSERT INTO included_in(
    SELECT DISTINCT is_a.inner_role,is_a.outer_role
        FROM is_a WHERE (inner_role,outer_role) NOT IN
        (SELECT inner_role,outer_role from included_in)
);
```

Code 26: SQL statement run by ***insert_included_in***

This trigger runs a stored procedure **insert_included_in**, which procedure runs the SQL statement in Code 26 to insert values into **included_in**.

```
DELETE FROM included_in_staging;
-- included_in(R1,R3) :- is_a(R1,R2), included_in(R2,R3).
INSERT INTO included_in_staging(
  (SELECT DISTINCT included_in.inner_role, is_a.outer_role
  FROM is_a JOIN included_in
  ON is_a.inner_role = included_in.outer_role)
  MINUS
  (SELECT inner_role,outer_role from included_in)
);

SELECT COUNT(*) INTO v_rows FROM included_in_staging;
IF (v_rows > 0 ) THEN
  INSERT INTO included_in (SELECT * FROM included_in_staging);
END IF;
```

Code 27: statements run by ***recourse_included_in***

Only records that are not already in **included_in** are inserted. A row-level post-action trigger on **included_in** is thus run, for each row entered into **included_in**. This runs the procedure **recourse_included_in** in Code 27.

The table **included_in_staging** is used to temporarily store records for inclusion in **included_in**. **included_in_staging** is cleared, and then populated with records that are to be added to **included_in** in this recursive call, and are not already in **included_in**. These rows are then inserted into **included_in**, recursively triggering the running of the same procedure.

```
INSERT INTO included_in(
  (SELECT DISTINCT included_in.inner_role, is_a.outer_role
  FROM is_a JOIN included_in
  ON is_a.inner_role = included_in.outer_role)
  MINUS
  (SELECT inner_role,outer_role from included_in)
);
```

Code 28: This does not work.

It would look as if the code in Code 28 would accomplish the objective much more simply, dispensing with the intermediate table. However, this leads to infinite recursion, apparently because the clause following the **MINUS** operator uses an out-of-date copy of **included_in**. Therefore, the same insertions, and consequently the same recursions, are run over and over. Integrity constraints would theoretically solve this problem, by preventing duplicate records from being inserted, but their violation causes a program error, preventing the procedure from running to completion.

The use of a table for temporary storage is certainly not the most elegant or efficient way of solving this problem. A PL/SQL or equivalent data structure within the procedure could probably be used instead, but this solution would be more programmatically complex to implement, as well as being DBMS-specific.

The table **senior_to** is populated by triggers on **d_s** and **senior_to** itself in a similar way to **included_in** with **is_a**.

A pre-action row-level trigger on **usr_session** performs validity checking in relation to the user's role assignment and dynamic separation of duties. Therefore, no records can be inserted into **usr_session** that activate a user for an incorrect role.

A pre-action row-level trigger on **ura** checks that a role to be assigned to a user does not violate static separation of duties rules defined in the table **ssd**, rejecting any record that would violate such conditions. In contrast, while the Prolog implementation defines `ssd` facts, it cannot enforce them, because it cannot check facts to be inserted for validity against a schema or set of rules.

Table 30 (page 221) summarizes the triggers necessary to create the RBAC data (data about users, roles and assignments of access rights to data tables).

Some tables have integrity constraints to prevent duplicate records from being inserted. Although **senior_to** and **included_in** are intended to have unique records, they do not have integrity constraints because these would interfere with the running of the triggers that populate them, as explained above.

Data diagrams for the RBAC data model are in Appendix VI (from page 217). Figure 116 (page 217) shows the ERD of the RBAC data model, showing tables only. Figure 117 (page 218) shows the ERD, showing tables and views. Figure 118, on page 218, graphically illustrates the **CREATE VIEW** relationships, linking each view with the objects involved in creating it. The schema diagrams for the RBAC data model are in Appendix VII (from page 219). Text 57 (page 219) and Text 58 (page 220) show the tables and views in the schema.

3.2.3 Enforcement of Static RBAC in DBMS Meta-data

After modelling the RBAC in a DBMS schema, it was set up in the meta-data of an Oracle DBMS. That is, the RBAC was set up so that it would be enforced by the built-in access control system of Oracle.

Oracle implements hierarchical RBAC in its meta-data. A role is created using **CREATE ROLE** *role_name*. The **GRANT ROLE** command is used to assign a user to a role, or a role to a role, as: **GRANT** *role1* **TO** *role2*/*user*. In this case, *role2* would inherit privileges assigned to *role1*.

Triggers on RBAC tables run the **CREATE**, **GRANT** and **REVOKE** statements necessary to create an enforceable RBAC model. Table 30 (Appendix VII, page 221) summarizes these triggers. Note that the users are **GRANT**ed roles through **usr_session**, not through **ura**. This is because merely being assigned to a role is not enough to obtain privileges: the user needs to be active in the role. Some actions on RBAC tables, particularly **UPDATE** actions, are prevented, mainly because allowing them would require programmatically complex triggers to ensure that the correct **CREATE**, **GRANT** and **REVOKE** commands were run following the action. For example, updating a record in **ura** is prevented. To change a role assigned to a user, the RBAC administrator is prevented from running an update query such as **UPDATE ura SET role = "house_officer_n" WHERE usr = "u0001" AND role = "house_officer_n";** instead, the RBAC administrator must **DELETE** the record in **ura** and **INSERT** a new record to replace it.

```
d_s(senior_role, junior_role): GRANT junior_role TO senior_role
is_a(inner_role, outer_role): GRANT outer_role TO inner_role
```

*Code 29: **GRANT**s performed through **d_s** and **is_a***

As Table 30 shows, this enforcement mechanism does not handle denials. This is due to the contradictory ways in which permissions and denials are inherited in the two hierarchies. As far as the enforcement mechanism is concerned, inheritance of a permission is the same whether it is performed through an **is_a** relationship or a **d_s** relationship: both lead to **GRANT role1 TO role2** statements being executed. Permissions filter up the seniority hierarchy, and inside the inclusion hierarchy, as in Code 29.

Denials filter *down* the seniority hierarchy: a junior role inherits denials from a senior role. However, denials filter inside the inclusion hierarchy, in the same way as permissions. Since permissions inherited either way look and act the same in meta-data, and denials are supposed to over-ride permissions, using **REVOKE** to enforce denials would be complex. Therefore, denials are ignored here.

Denials are handled by the RBAC mechanism, in the same way as context constraints. This makes sense, because denials filter in the same direction as context constraints for both hierarchies.

Inheritance paths are not handled in this implementation, but could be partially handled by appropriate use of **NOINHERIT** in **GRANT** commands.

3.3 Dynamic RBAC

After modelling the static RBAC model described in Sections 2.2.1 and 2.2.2 in Oracle SQL, the dynamic RBAC model described in Section 2.3 was then modelled.

3.3.1 Representation of Dynamic RBAC Model in Prolog

Table 4: Fact definition used in dynamic RBAC design in Prolog

Fact Formula	*Description*
`associated_cc(Role,Permission, Object,ContextConstraint).`	The context condition `ContextConstraint` applies when a user with role Role accesses object Object using Permission.
`ssd(Role1,Role2).`	A static separation of duties relationship exists between `Role1` and `Role2` (i.e. no user can be assigned to both roles using `ura`).
`dsd(Role1,Role2).`	A dynamic separation of duties relationship exists between `Role1` and `Role2` (i.e. no user can be active in both roles simultaneously using activate).

Table 5: Rules in Prolog dynamic RBAC design

Rule Name	*Description*
`applied_cc`	Whether a context constraint applies to a user performing an action.
`fail_context_constraint`	Whether an action fails a context constraint, considering its applicability.
`violated`	Whether an action would fail a context constraint, irrespective of its applicability.
`context_condition`	Defines the circumstances in which a user can perform an action on an object.

Table 4 and Table 5 list the Prolog facts and rules used for dynamic RBAC.

In dynamic RBAC, permissions assigned to users and roles vary dynamically according to "context conditions". These are internal (determined by database values) or external (determined by the environment) rules that affect permissions. A context condition may affect all roles defined in the RBAC model, or only specific rules. It may affect access to all columns in all tables in the database, or only some of these. When context conditions are applied to the RBAC model, they become "context constraints". The rules are defined in `context_condition` predicates.

```
context_condition(
  patient_treated_by_doctor, Doctor_ID, P,
  patient(
    Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID
  )
) ← ae_consultation(_,_,_,Patient_ID,Doctor_ID).
```

Code 30: Example of `context_condition` definition.

Code 30 gives an example of a `context_condition` predicate called `patient_treated_by_doctor`. This defines a condition that is true only if both `Doctor_ID` and `Patient_ID` appear in the same `ae_consultation` fact. `patient_treated_by_doctor` is thus a simple row-level internal context condition. The implementation of context constraints for dynamic RBAC is based on the model of Strembeck & Neumann [2004] [21].

The following paragraphs describe the Prolog rules that are used to apply the context conditions to the RBAC model, thus making it a dynamic model. These Prolog rules are listed in full in Appendix III.

```
applied_cc(R1,P,O,CC) ← associated_cc(R3,P,O,CC),
                        senior_to(R3,R2),
                        included_in(R1,R2).
```

Code 31: Context constraint testing with `applied_cc`.

Code 31 shows Prolog rule `applied_cc`, which determines whether a particular context condition `CC` applies to role `R1`, based on seniority and inclusion rules. In this implementation, Context Constraints filter down a role

hierarchy, and inside an inclusion hierarchy. That is, a junior role inherits the context constraints of a senior role, and an included role inherits context constraints of a role to which it belongs.

```
violated(CC,U,P,O) ← not(context_condition(CC,U,P,O)).
```

Code 32: Context constraint violation with `violated`.

Code 32 shows Prolog rule `violated(CC,U,P,O)`, from tests whether user `U` would violate context condition `CC` when trying to perform action `P` on object `O`, without considering whether the context constraint applies to the particular user. `violated` simply negates `context_condition` for `CC`.

```
fail_context_constraint(U,R,P,O) ← applied_cc(R,P,O,CC),
                                   violated(CC,U,P,O).
```

Code 33: Test for access attempt failing context constraint with `fail_context_constraint`.

Code 33 shows Prolog rule `fail_context_constraint`, which determines whether an access `P` by user `U` logged in as role `R` on object `O` that violates some context constraint `CC` that applies to role `R`. This applies `violated` to test whether `CC` would be violated, and `applied_cc` to determine whether `CC` is applicable to user `U`.

```
permitted(U,P,O,R) ← currently_active(U,R,_),
                     not(fail_context_constraint(U,R,P,O)),
                     permittable(U,P,O,R).
```

Code 34: Prolog rule `permitted` in the dynamic RBAC model.

Finally, `permitted` is modified from the rule in Code 11 (page 34, Section 3.2.1) to take account of context constraints. The new Prolog rule `permitted` is shown in Code 34. For `permitted(U,P,O,R)` to be true, the following must be true:

1. User `U` must be currently active in role `R`.
2. The combination of `U`, `R`, action `P` and object `O` must not fail any context constraint.
3. `U` must have the relevant static potential permission as determined by `permittable`.

Appendix V lists all the context constraints in the RBAC model used in testing.

3.3.2 Transformation of Dynamic RBAC Model from Prolog to SQL Database

```
CREATE TABLE tbl_rows (
  row_id VARCHAR(256),
  object VARCHAR(64) -- table name
);
```

*Code 35: **`CREATE TABLE`** for **`tbl_rows`**.*

The dynamic RBAC model was implemented in SQL in a similar way to the implementation of static RBAC, transforming the Prolog rules described in Section 3.3.1 into SQL views and PL/SQL triggers. Additionally, a table **`tbl_rows`** is defined (Code 35), storing all primary keys of all data tables. Triggers are defined on each table to modify the **`rows`** table whenever a data table is modified.

row_id is the primary key for a table row (in composite keys, the columns are separated by tildes), while **object** is the name of the table in which the key in **row_id** appears.

This table is necessary to uniquely identify each row of each object that may be accessed, and store them in the same place. Without it, the views discussed below would need to perform Cartesian products on tables affected by a context constraint.

Each context constraint has three (for temporal constraints) or four (for row-level constraints) associated views:

- **<CC_name>**, defining the constraint itself, equivalent to the `context_condition` predicates in Prolog.
- **applies_<CC_name>**, which specifies the roles, actions and objects to which the context condition applies;equivalent to the `associated_cc` predicates in Prolog.
- **fails_<CC_name>**, which determines whether a context constraint is violated; equivalent to the `violated` predicate in Prolog. It returns users, actions, objects and rows that would violate the context constraint, determined by the user, action, object and (if applicable) row *not* appearing in **<CC_name>**, and the role and object appearing in **applies_<CC_name>**. This view uses **permittable** to obtain the list of permissions that would apply without the context constraints, and joins it with rows in the **rows** table corresponding to the appropriate table object. For example, if the object to which the constraint applies is the **patient** table, then each row of rows in **rows** where **object=patient** is joined with each row of the relevant subset of **permittable**.

```
CREATE VIEW permittable_by_row AS
  SELECT usr, permittable.object as object, action, role, row_id
    FROM permittable, tbl_rows
    WHERE permittable.object = tbl_rows.object;
```

*Code 36: Definition of **permittable_by_row**.*

The view **permittable_by_row**, shown in Code 36, shows all rows that each user in each role would be permitted to access before context constraints are applied. It is a join of **permittable** and **tbl_rows**.

permitted_by_row, **authorizable_by_row** and **authorized_by_row** are defined analogously to **permittable_by_row**.

The view **fails_context_constraints** (not shown) is a **UNION** of all **fails_<CC_name>** views, and thus retrieves all potential attempts by a user to perform an action on a row in an object that would fail a context constraint.

The view **permittable_cc**, shown in Code 37, retrieves all rows that do *not* fail any context constraints. It selects all rows from **permittable_by_row** that do not appear in **fails_context_constraints**, using **MINUS** to filter out the unwanted rows.

```
CREATE VIEW permittable_cc AS
  SELECT usr, object, row_id, action, role FROM permittable_by_row
    MINUS
  SELECT usr, object, row_id, action, role FROM fails_context_constraints
;
```

*Code 37: Definition of **permittable_cc**.*

permitted_cc, **authorizable_cc** and **authorized_cc** are defined analogously to **permittable_cc**.

3.3.3 Enforcement of Dynamic RBAC in DBMS Meta-data

A feature in Oracle known variously as Row-Level Access Control, Fine-Grained Access Control and VPD (Virtual Private Databases) [65] was used to implement both row-level and temporal context constraints at the meta-data level. This means that Oracle's own permission-granting mechanism can be used to allow or deny access dynamically, rather than using ordinary database tables.

Because denials are inherited in the same way as context constraints (and indeed can be regarded as a type of context constraint), they are also handled using this feature.

The method for implementing this system is a four-step method described by Finnigan [58]. These four steps are listed below and each described in turn.

1. Create a security context to manage application sessions.
2. Create a procedure or function to manage setting of the security context for users.
3. Write a package to generate the dynamic access predicates for access to each table.
4. Register the policy function / package with Oracle using the DBMS_RLS package.

Step 1: Create a security context to manage application sessions.

```
CREATE OR REPLACE CONTEXT hosp USING set_context;
```

*Code 38: Setting security context for **hosp** database.*

A security context is a set of name/value pairs that can be used to bind a particular user to named context constraints. It must be set through a PL/SQL package. Code 38 shows the command for setting the security context for the **hosp** database. This command is run from the script that prepares the database for setup.

Step 2: Create a procedure or function to manage setting of the security context for users.

```
PROCEDURE set_day_duty
  IS
  BEGIN
    dbms_session.set_context('hosp', 'day_duty', 'y');
  END;
```

Code 39: An example of a context-setting function

Code 39 shows the code for the **set_day_duty** procedure the function for setting the **day_duty** context. The built-in procedure **dbms_session.set_context('hosp', 'day_duty', 'y')** sets the session context variable **'day_duty'** of the current user in the context **hosp** to the value **'y'**. Of course, this context variable was previously not set.

```
PROCEDURE set_cc( p_role VARCHAR )
  IS
  BEGIN
    IF( is_part_of(p_role,'day_duty' )) THEN
      set_day_duty;
    END IF;
    IF( is_part_of(p_role,'night_duty' )) THEN
      set_night_duty;
    END IF;
    IF( is_part_of(p_role,'sister')) THEN
      set_nurse_ward;
    END IF;
    IF( is_part_of(p_role, 'snr_house_officer')) THEN
      set_patient_doctor;
    END IF;
    IF( is_part_of(p_role, 'jnr_data_manager')
    OR is_part_of(p_role, 'receptionist')) THEN
      set_office_hours;
    END IF;
    IF( is_part_of(p_role, 'student_nurse' )) THEN
      set_staff_sister_active_2_h;
    END IF;
  END;
```

*Code 40: **set_cc** procedure to manage setting of the security context for users*

```
CREATE OR REPLACE FUNCTION is_part_of( p_inner_role VARCHAR, p_outer_role VARCHAR )
RETURN BOOLEAN
IS
  v_num_rows1 INT;
  v_num_rows2 INT;
BEGIN
SELECT COUNT(*) INTO v_num_rows1 FROM included_in, senior_to WHERE
  p_inner_role = included_in.inner_role AND
  included_in.outer_role = senior_to.junior_role AND
  senior_to.senior_role = p_outer_role;
SELECT COUNT(*) INTO v_num_rows2 FROM included_in WHERE
  p_inner_role = included_in.inner_role AND
  included_in.outer_role = p_outer_role;
RETURN (v_num_rows1 + v_num_rows2 > 0);
```

*Code 41: **is_part_of** procedure to determine inheritance of context constraints by roles*

```
 PROCEDURE set_denials( p_role VARCHAR )
 IS
   v_action VARCHAR(64);
   v_object VARCHAR(64);
 CURSOR c_get_denials IS
   SELECT action, object FROM d_rpa_full WHERE role = p_role;
 BEGIN

   OPEN c_get_denials;

   LOOP
     FETCH c_get_denials INTO v_action, v_object;
     EXIT WHEN c_get_denials%NOTFOUND;

     -- set context constraints
     dbms_session.set_context('hosp', 'denied_' || v_object || '_' || v_action, 'y');
   END LOOP;

   CLOSE c_get_denials;

 END;
```

*Code 42: **set_denials** procedure*

Code 40 shows the procedure to manage setting the security context of users, which is called **set_cc** and is contained in the PL/SQL package through which the security context is manipulated. The package is called **set_context** in this implementation. **set_cc** sets the appropriate context constraints for a user depending on his roles. It calls any (none, one or more than one) of a series of procedures to set context constraints depending on the user's role. Denials are then set using the **set_denials** procedure.

The function **is_part_of(role1, role2)**, shown in Code 41, uses the seniority and inclusion hierarchies (as given by **senior_to** and **included_in**) to determine whether **role1** inherits constraints from **role2**.

The procedure **set_denials**, shown in Code 42, checks whether a denial is applicable to the role, action and object by querying **d_rpa_full**. If a row is found in **d_rpa_full**, then the session context variable **'denied_<object>_<action>'** is set.

Step 3: Write a package to generate the dynamic access predicates for access to each table.

```
if sys_context('hosp', 'denied_' || object_name || '_select') = 'y' then
    return '0 <> 0'; -- return always-false condition and bail out
else
    return (cc(schema_name, object_name));
end if;
```

Code 43: code for ***cc_select***

```
if object_name = 'PATIENT' AND sys_context('hosp','nurse_in_same_ward_as_patient') = 'y' then
      if( has_cc ) THEN
        lv_predicate:=lv_predicate || ' AND ';
      END IF;
      lv_predicate:=lv_predicate || ' Patient_id IN (SELECT Patient_id FROM patient_bed, Bed, Room,
Ward, nurse_ward
      where get_usr = nurse_ward.usr
      AND nurse_ward.Ward = Room.Ward_id
      AND Room.Room_id = Bed.Room_id
      AND Bed.Bed_id = patient_bed.Bed_id)';
```

Code 44: Code for context ***nurse_in_same_ward_as_patient***

```
SELECT * FROM patient WHERE Patient_id IN (SELECT Patient_id FROM patient_bed, Bed, Room, Ward,
nurse_ward
      where get_usr = nurse_ward.usr
      AND nurse_ward.Ward = Room.Ward_id
      AND Room.Room_id = Bed.Room_id
      AND Bed.Bed_id = patient_bed.Bed_id;
```

Code 45: Actual SQL run as a result of ***nurse_in_same_ward_as_patient***

In this implementation, the package is called **policy**. It has a series of functions, **cc_<action>**, where **<action>** is **SELECT**, **INSERT** etc. These functions test each session context variable to see if it is set, and adds an appropriate predicate to restrict the rows accessible. In this model, all context constraints apply to all actions in the same way. Therefore, each **cc_<action>** function first calls the appropriate denial session context variable. If there are no denials, then the function **cc** is called which applies the context constraint. Code 43 shows the code for **cc_select**.

cc tests each session context variable (other than those related to denials) to see if it is set, and adds an appropriate predicate to restrict the rows accessible. For example, Code 44 shows the code for the context **nurse_in_same_ward_as_patient**.

This applies the restrictive SQL in **lv_predicate** if the object being accessed is the **patient** table, and the session context variable **nurse_in_same_ward_as_patient** is set to **y**. The restrictive SQL acts like a **WHERE** clause, turning **SELECT * FROM patient** into the **SELECT** statement in Code 45.

One might expect the last line of the SQL clause in Code 45 to read **AND Bed.Bed_id = patient.Bed_id**, since the information about patient beds is stored in the **patient** table. However, this would produce an error, since the restrictive predicate is trying to access the very table to which it is being applied, thus causing infinite recursion. To solve this problem, triggers were defined on any modification of data in the **patient** table, to add/modify/delete as appropriate the values for **patient_no** and **bed_id** to a table **patient_bed**, which can be read by the predicate without causing errors.

```
elsif sys_context('hosp','office_hours') = 'y' then
      lv_predicate:='TO_CHAR (SYSDATE, ''HH24'') >= 9 AND TO_CHAR (SYSDATE, ''HH24'') < 17 AND
TO_CHAR (SYSDATE, ''D'') >= 2  AND TO_CHAR (SYSDATE, ''D'') <= 6';
```

*Code 46: Conditional clause testing for the **office_hours** context*

Code 46 shows the conditional clause testing for the **office_hours** context. Note that Oracle's date formatting scheme treats the days of the week as 1–7, Sunday–Saturday, so Monday is 2 and Friday is 6.

Step 4: Register the policy function / package with Oracle using the DBMS_RLS package.

```
begin
  dbms_rls.add_policy(
    object_schema => 'HOSP',
    object_name => 'PATIENT',
    policy_name => 'CC_PATIENT',
    function_schema => 'HOSP',
    policy_function => 'POLICY.CC_SELECT',
    statement_types => 'select',
    update_check => TRUE,
    enable => TRUE,
    static_policy => FALSE);
end;
/
```

Code 47: Procedural code for registering policy function

```
-- security context
CREATE OR REPLACE TRIGGER cc_logon_trigger
AFTER LOGON ON DATABASE
DECLARE
 ...
  CURSOR c_get_roles IS
    SELECT role FROM currently_active WHERE
        usr = (SELECT get_usr FROM DUAL);
BEGIN
  -- get user
  OPEN c_get_roles;
  LOOP
    FETCH c_get_roles INTO v_role;
    EXIT WHEN c_get_roles%NOTFOUND;
--      -- set context constraints
    set_context.set_cc(v_role);
  END LOOP;
  CLOSE c_get_roles;
END;
```

Code 48: Setting a user's security context when he logs on

The policy function needs to be registered for each table to which it applies, using procedural code such as that in Code 47. Additionally, the security context must be set whenever a user logs on. This is achieved using a trigger run when a user logs onto the database (**AFTER LOGON ON DATABASE**), as shown in Code 48. This trigger performs determines the roles in which the user is active, using the user-defined function **get_usr** to obtain the user name as stored in **usr** and **password** from the DBMS username. Then, for each role, it calls **set_context.set_cc** to set appropriate context constraints, if any.

3.4 Testing the Implementation of RBAC in Oracle

3.4.1 Overview: Parts and Conditions

The RBAC model described in Sections 3.2 and 3.3 was tested in an Oracle 10*i* database system by running a series of SQL batch files (Appendix XIV) on an Oracle database containing the RBAC model and sample data.

Parts

The testing was performed on the RBAC model in the following order.

a) ***Representation of RBAC*** in database tables: output the results of querying RBAC views;

b) ***Enforcement of RBAC*** in meta-data: attempting to log in and access and manipulate data as different users to test whether the access control assignments in meta-data gave the correct permissions;

c) Testing whether static and dynamic ***Separation of duties*** worked correctly.

Conditions

Users are not supposed to be able to access any data unless they are 'active', which would mean that they are logged in to and using some application that accesses the database. This is represented by the table **`usr_session`** in the RBAC schema. To make sure that user authorizations were correctly applied and revoked when uses were activated and deactivated, testing in parts a and b was run under four conditions:

1. ***No users activated:*** this was to confirm that no users were able to access any part of the database when they were not activated.

2. ***Some (17 out of a total 29) users activated***: to determine the effects of activating users on their access to data, and confirming that only activated users could access data. The number of users activated was arbitrary.

3. ***All users activated***: to confirm the system behaviour when all users had access to the system, and ensure that Representation and Enforcement both produced the same results.

4. ***Some users deactivated***: leaving 21 users active, to confirm that deactivating a user resulted in the withdrawal of the user's access. The number of users deactivated was arbitrary.

3.4.2 Representation of RBAC

The tables and views were queries in the following order:

1. Role Permissions and Denials (*`rpa`* and *`d_rpa`*)
2. Static User Permissions and Authorizations (**`permittable`**, **`authorizable`**, **`permitted`** and **`authorized`**)
3. Dynamic User Permissions and Authorizations (**`permittable_cc`**, **`authorizable_cc`**, **`permitted_cc`** and **`authorized_cc`**)

1 Role Permissions and Denials (rpa and d_rpa)

For each of the four test runs, queries on *`rpa`* and *`d_rpa`* were first displayed, to identify permissions applied to roles. Then, queries on static and dynamic RBAC views were displayed, identifying permissions assigned to users.

```
select role "Role", action "Action", object "Object" from rpa where
role = 'role_name' order by role, action, object;
select role "Role", action "Action", object "Object" from rpa_full
where role = 'role_name' order by role, action, object;
```

Code 49: Displaying **rpa** *table and* **rpa_full** *view*

First, the **rpa** table and the **rpa_full** view were displayed for each role (Code 49). The **rpa** table stores all permissions explicitly assigned to roles. The **rpa_full** view returns all permissions available to each role, whether explicitly assigned or inferred from relationships with other roles.

Results from **d_rpa** and **d_rpa_full** were then displayed. These correspond to **rpa** and **rpa_full**, respectively, for denials rather than permissions.

2 Static User Permissions and Authorizations (permittable, authorizable, permitted and authorized)

Views showing individuals' permissions and authorizations were displayed, for both static and dynamic RBAC.

```
select usr "User", object "Object", action "Action", role "Role" from permittable where role =
'role_name' ORDER BY usr, object, action;
select usr "User", object "Object", action "Action", role "Role" from authorizable where role =
'role_name' ORDER BY usr, object, action;
select usr "User", object "Object", action "Action", role "Role" from permitted where role =
'role_name' ORDER BY usr, object, action;
select usr "User", object "Object", action "Action", role "Role" from authorized where role =
'role_name' ORDER BY usr, object, action;
```

Code 50: Displaying **permittable**, **permitted**, **authorizable** *and* **authorized** *views for static RBAC.*

The static permissions and authorizations were first displayed. These do not take account of dynamic context constraints. The **permittable**, **authorizable**, **permitted** and **authorized** views were displayed for each role in turn (Code 50). **permittable** and **authorizable** respectively display permissions and authorizations that can be applied to each user and role. **permitted** and **authorized** respectively display permissions and authorizations that currently apply, depending on whether users are logged in. In other words, **permittable** displays the permission that a user would have if he had been logged in, while **permitted** displays it only if the user is actually logged in. The **permittable** and **permitted** views are derived from permissions (**rpa** assignments) only. The **authorizable** and **authorized** views are derived from permissions filtered by denials (**d_rpa** assignments).

3 Dynamic User Permissions and Authorizations (permittable_cc, authorizable_cc, permitted_cc and authorized_cc)

```
select distinct usr "User", role "Role", object "Object", action "Action", row_id "Row" from
permittable_cc where role = 'day_duty' ORDER BY usr, object, action;
select distinct usr "User", role "Role", object "Object", action "Action", row_id "Row" from
authorizable_cc where role = 'day_duty' ORDER BY usr, object, action;
select distinct usr "User", role "Role", object "Object", action "Action", row_id "Row" from
permitted_cc where role = 'day_duty' ORDER BY usr, object, action;
select distinct usr "User", role "Role", object "Object", action "Action", row_id "Row" from
authorized_cc where role = 'day_duty' ORDER BY usr, object, action;
```

Code 51: Displaying dynamic permissions and authorizations for day_duty *and* night_duty *roles.*

The dynamic permissions and authorizations were displayed from the **permittable_cc**, **authorizable_cc**, **permitted_cc** and **authorized_cc** views were displayed for each role in turn (Code 51,

for *day_duty* and *night_duty* roles). These show permissions and authorizations by row rather than by object, because some dynamic constraints mean that only some rows in tables are visible.

Running

```
DELETE FROM ae_consultation WHERE 0 <> 0;
```

Code 52: A sanitized ***DELETE*** *statement.*

The script in Appendix XIV was run logged in as each database user in turn to determine whether the RBAC model translated correctly into user permissions in the meta-data. Data manipulation commands were run for all data, including RBAC data. However, to ensure that no damage was done to the data, **DELETE** statements were suffixed with **WHERE 0<>0**, as in Code 52.

In this case, the **DELETE** statement is run, as long as the user has the appropriate permission, thus an appropriate error results if the user does not have the appropriate permission. However, nothing is actually deleted, because the **WHERE** clause always evaluates to **FALSE**.

Finally, a test was also run to test the enforcement of static and dynamic separation of duties in the **ssd** and **dsd** tables of the RBAC schema. Various users were assigned or activated to various combinations of roles, to discover whether the static and dynamic separation of duties constraints worked correctly.

3.5 Results

This section summarises the results of testing according to the procedure in Section 3.4, in the following order.

a) ***Representation of RBAC*** in database tables: output the results of querying RBAC views;

b) ***Enforcement of RBAC*** in meta-data: attempting to log in and access and manipulate data as different users to test whether the access control assignments in meta-data gave the correct permissions;

c) Testing whether static and dynamic ***Separation of duties*** worked correctly.

The detailed results are given in Appendix XVII. The static and dynamic user permissions and authorisations were then checked according to the Conditions in Section 3.4.1 relating to the numbers of users who were activated. To recap, the four conditions were

1. ***No users activated***
2. ***Some users activated***
3. ***All users activated***
4. ***Some users deactivated***

First, the contents of **rpa**, **rpa_full**, **d_rpa** and **d_rpa_full** were output to ensure that they contained the correct role permissions and denials. This was done only for part a. The permissions and denials associated with roles did not change according to user activity, so were the same for all the above conditions. The output of **rpa** and **d_rpa** is described by type of role, in the following order:

1. Temporal RBAC Roles: *day_duty* and *night_duty*
2. Job Roles: Data Managers
3. Job Roles: Doctors

4. Job Roles: Nurses
5. Job Roles: Administrators

All role permission and denial assignments were found to be correct. The static and dynamic user permissions and authorisations were then checked according to the Conditions in Section 3.4.1 for both parts a and b. The results of these are summarised below.

No Users Activated: Queries on **permittable** and **authorizable** correctly retrieved all static permissions and authorizations allocated to users. Queries on **permitted** and **authorized** correctly produced empty recordsets, because for users to be actually permitted to perform any actions, they would need to be activated. No users were activated in this test run. Queries on **permittable_cc** and **authorizable_cc** correctly retrieved all dynamic permissions and authorizations allocated to users and applicable at the time of running. Queries on **permitted** and **authorized** correctly produced empty recordsets, because for users to be actually permitted to perform any actions, they would need to be activated. No users were activated in this test run. Attempts to log in as each user and manipulate data were unsuccessful, because none of the users were activated. Therefore, the meta-data correctly recorded that no users had any permission to change the data.

Some Users Activated: Queries on **permittable** and **authorizable** correctly retrieved all static permissions and authorizations allocated to users. Queries on **permitted** and **authorized** correctly produced the static permissions and authorizations for active users only. Queries on **permittable_cc** and **authorizable_cc** correctly retrieved all dynamic permissions and authorizations allocated to users and applicable at the time of running. Queries on **permitted** and **authorized** correctly retrieved all dynamic permissions and authorizations of active users only. When users were logged on, active users were able to manipulate data according to their dynamic authorizations. Non-active users were unable to manipulate data. Thus, the meta-data correctly reflected authorizations of active users.

All Users Activated: Queries on **permittable** and **authorizable** correctly retrieved all static permissions and authorizations allocated to users. Queries on **permitted** and **authorized** retrieved the same results as **permittable** and **authorizable**, since all users were active on this test run. Queries on **permittable_cc** and **authorizable_cc** correctly retrieved all dynamic permissions and authorizations allocated to users and applicable at the time of running. Queries on **permitted** and **authorized** also retrieved the same results as **permittable_cc** and **authorizable_cc**, since all users were active on this test run. Since all users were active, they were all able to log on and manipulate data according to their dynamic authorizations. However, the *manager* user **u0021** was incorrectly given the authorizations of *snr_data_manager*, *specialist_nurse* and *consultant*. This is because the VPD cannot handle path inheritance, which the RBAC Model uses to prevent users of role manager from having these authorizations despite being **senior_to** these roles. Apart from this problem, the meta-data correctly reflected authorizations of all users.

Some Users Deactivated: Queries on **permittable** and **authorizable** correctly retrieved all static permissions and authorizations allocated to users. Queries on **permitted** and **authorized** correctly produced the static permissions and authorizations for active users only, i.e. only those users who had not been deactivated for this test run. Queries on **permittable_cc** and **authorizable_cc** correctly retrieved all dynamic permissions and authorizations allocated to users and applicable at the time of running. Queries on **permitted** and **authorized**

correctly retrieved all dynamic permissions and authorizations of active users only. When users were logged on, active users were able to manipulate data according to their dynamic authorizations. Non-active users were unable to manipulate data. Thus, the meta-data correctly reflected authorizations of active users.

Finally, for ***Separation of Duties***, both static and dynamic separation of duty constraints were successful. Users could not be assigned to roles such as to cause static SSD conflicts, and could not be activated in roles such as to cause DSD conflicts.

3.6 Conclusion

The dynamic RBAC model was implemented in Oracle, first by transforming the original Prolog rules into RBAC data tables, then by encoding it in Oracle meta-data using VPD. With various sets of users activated, the RBAC data tables were queried, and VPD implementation was tested by logging users in and attempting to manipulate data while logged in. Finally, SSD and DSD constraints were tested.

With one exception, all tests produced the expected results. Querying RBAC data tables and views retrieved the correct static and dynamic permissions for roles and for both active and inactive users in all cases. When attempting to manipulate data as logged-in users, inactive users were unable to manipulate data in all cases. Active users were able to perform data manipulation operations that they were authorized to perform according to the dynamic RBAC rules, and (with one exception) were unable to perform operations that they were not authorized to perform. The exception relates to the inability to program selective path inheritance in Oracle VPD: senior roles cannot be prevented from inheriting permissions from junior roles. Therefore, any user in role `manager` (**`u0021`** here), being senior to all other roles, was able to perform all actions, although the **`inherits_rpa_path`** rules intended that a manager would only inherit from receptionist. The permissions and authorizations of user **`u0021`** were correctly displayed when querying the RBAC data, but were not reflected correctly in the meta-data enforcement.

Thus, Oracle's VPD feature can implement most features of the RBAC model discussed in this section. Specifically, it can implement Seniority, Denials, Activity, Separation of Duties and Context Constraints, but not selective Role Inclusion or Path Inheritance.

The free-software DBMS, PostgreSQL, has an add-on called Veil that provides row-level access control [66]. This feature might allow dynamic RBAC to be implemented in PostgreSQL. Implementing dynamic RBAC in MySQL would be tricky, because MySQL does not natively support RBAC.

4 The Problem

4.1 Problems with Current RBAC

RBAC has been a much exploited model of access control in database communities for more than a decade. It has been deployed in numerous applications, and has already been chosen as a mandatory authorization mechanism in many healthcare systems across the world, including the National Health Service (NHS) in the United Kingdom. From that perspective, current RBAC models and their implementations in database management systems need improvement. A currently unresolved issue is the automatic extraction of semantics from a given relational database schema, which are essential for creating RBAC models. That is, no automated mechanisms are available to help understand the semantics stored in database schemas. Therefore, systems are needed to automatically 'read' and 'understand' metadata before generating RBAC models. There are no commercially available solutions that allow automatic creation of RBAC models and their implementation, by reading metadata and using semantic web tools to deal with RBAC semantics.

In the experiments in Chapter 3, two different approaches were compared for implementing an RBAC model based on Prolog facts in a relational database: (1) storing the RBAC-related Prolog facts as records in database tables, and (2) storing them in the meta-data of the DBMS.

When using method (1), the tables holding RBAC data were stored in the same database as the data over which the RBAC was run, but (as would be typical for this approach) in a different schema. Using this method, all aspects of the RBAC models can be implemented, and the RBAC can be determined by issuing standard SQL queries on RBAC schema tables. This approach can be used to provide access control at the application level. At the database level, the application always accesses the data using one user ID, which is likely to be locked to accessing data from the application interfaces. The application would pass the user ID of the person who is logged into it as a parameter to the database when the user attempts to access data, and this would form part of the query to determine whether the application-level user gains the access. Furthermore, we can easily program both static and dynamic RBAC at the application level because the rules for both can easily be translated into either SQL views or PL/SQL (or equivalent) procedures.

Method (2), of implementing RBAC on a relational database provides access control at the *database* level by using the meta-data (or data dictionary) of the RDBMS. In this method, we have to distinguish between static and dynamic RBAC, which are implemented in different ways. The static RBAC was mostly implemented using standard SQL **CREATE ROLE**, **CREATE USER** and **GRANT** commands. However, while RBAC permissions can be implemented this way, denials cannot be so implemented because **GRANT** is only a positive granting of permission: there is no negative authorisation in SQL access control syntax. The dynamic RBAC was then implemented using Oracle's Virtual Private Databases (also called Row-Level Access Control) feature. [58] We found that most, but not all, of the features of the RBAC model could be implemented. We could not implement path inheritance restrictions. However, denials can be implemented using this feature, because a rule can be set up such that a role is denied access to data in a table even if given access to it via a **GRANT** command. The implementation of dynamic RBAC is product-specific, as it is not part of the SQL standard. Postgres has a feature called VEIL [66] that also implements dynamic RBAC, but its syntax is different from that of Oracle VPD. By contrast, the static RBAC implementation uses standard

SQL commands, and is likely to be very similar across RDBMSs, although some, such as MySQL, do not support RBAC in their data dictionary.

Prolog and relational database systems cannot naturally represent hierarchical data, which is the backbone of any semantic representation of RBAC models. A role being a type of another role is represented as a predicate, such as `is_a(role1, role2)`. A user's membership of a role is also represented as a predicate, such as `ura(user, role)`. These predicates are represented in an RDBMS as either rows in database tables (method (1)) or metadata (method (2)). This way of representing hierarchical data means that implementation of RBAC in Prolog can be complex and rigid, due to the need to chain many joins to represent traversing an RBAC hierarchy. This makes predicate logic and relational database systems especially cumbersome when trying to model dynamic RBAC, in which permissions may change according to context. An RBAC model should be able to infer user authorisations from a hierarchy of both roles and data types, that is, determine permission or denial from not just the type of role (which may include sub-roles), but also the type of data (which may include sub-types).

However, OWL reasoner-enabled ontologies could resolve both these problems by allowing us to describe and manipulate the semantics of RBAC differently. OWL naturally represents data and concepts in a hierarchical fashion, and its implementation is not vendor-specific. Therefore, this thesis considers the possibilities offered by OWL for developing models and reasoning processes for RBAC, which are domain and implementation independent, and can be run from any distributed computing environment.

4.2 Literature Review

4.2.1 RBAC and XML

XACL (XML Access Control Language), also known as XACML (XML Access Control Markup Language) [67][68] is the standard representation of access control using XML, and has provision for RBAC. MOSQUITO (Mobile Workers' Secure Business Applications in Ubiquitous Environments) [69] is an example of a system based on XACML for providing dynamic RBAC in a ubiquitous computing environment.

Chandramouli [70] devised an XML/DTD model for determining access control in an XML-based banking database, in which the record and field types are written in a DTD (Document Type Definition), and the access control rules and data in XML documents bound by the DTD.

Vuong *et al.* [71] presented a Java-based system for assigning and applying RBAC permissions to data in XML documents. Bertino & Ferrari proposed Author-X [72], a Java-based system for securing XML documents on a network. The access permissions are stored in XML documents, and determine access to parts of documents according to its DTD or security information held in the document itself. The documents protected by the system are stored in encrypted form. Bertino *et al.* also [73] devised an infrastructure for managing secure updates to XML documents.

Bhatti *et al.* presented X-RBAC [74], an XML-based model for applying RBAC in Web Services. This was then extended for context-aware RBAC [75] and multi-domain environments [76], thus providing dynamic X-RBAC. The same authors also proposed X-FEDERATE [77] an XML-based model for managing access control in federated distributed environments, in which each node has a direct connection to all other nodes in the network.

XML-based RBAC models have also been proposed by He & Wong (RBXAC) [78] and Stoupa & Vakali [79]. Yang & Zhang [80] proposed a similar model for securing web-based applications.

GTRBAC [50], from Chapter 2.3, has been implemented in XML as X-GTRBAC [81],and an administration module, X-GTRBAC Admin, has been built for this [82].

Yang *et al.* [83] and Warner *et al.* [84] proposed XML-based dynamic RBAC models that use semantic matching in heterogeneous databases to dynamically determine access permissions by linking semantically equivalent but differently named entities in each of them.

Finance *et al.* [85] proposed a model for access control in XML documents in which access rules can be set on any node anywhere in the relational hierarchy of the document (not only leaf nodes), and can be used control access to ancestor and sibling relationships. This allows the creation of different "authorized views" of an XML document, depending on the access right of a user.

Bouna *et al.* [86] proposed an XML-based RBAC model for determining access to multimedia objects based on the low-level data in these objects. This allows the same access to be given to, for example, any object described as relating to Charles de Gaulle in the 2nd World War.

Another XML-based access control model is PERMIS (PrivilEge and Role Management Infrastructure Standards) [87], which uses X.509 attribute certificates [88] to hold user roles. This makes it more than just a policy language, like XACML, but also an authorization system. The authorization tool is beyond the scope of this thesis, which is concerned with the access control policy, rather than the methods of authorizing users based on the policy. PERMIS policies are written in XML, but the syntax is briefer than that of XACML. A Java-based PERMIS policy-writing tool has also been developed [89].

4.2.2 RBAC and the Semantic Web

Semantic web technologies have been used to represent access control models, thus facilitating the incorporation of access control systems into software applications using the semantic web. Many RBAC implementations that address interoperability, or allow automatic creation of RBAC models by reading meta-data using the Semantic Web (OWL [90], RDF [91] and XML), have been proposed.

Several previous works on designing ontologies for RBAC have addressed aspects of static and dynamic RBAC.

Pan *et al.* [92] proposed Semantic Access Control (SAC), an RBAC-based model for access control in heterogeneous systems, and developed a middleware application, called Semantic Access Control Enabler (SACE), to implement it on the Web.

Wu *et al.* [93] modelled RBAC using OWL, with separation of duty and prerequisite constraints, but without considering constraints typical for dynamic RBAC. Furthermore, their use of 'constraints' is not the same as ours, because we follow the work of [21], where 'context constraints' refer to dynamic constraints applied to RBAC rules. Additionally, they do not use Prolog facts or rules in order to specify the semantics stored within the RBAC. Wu *et al.* [94] extends [93] with their OBAC (Ontology-Based Access Control), an RBAC specification for distributed systems using OWL [90] and SWRL [95]. Roles are modelled using classes in OWL, as in [93], while role constraints are modelled in SWRL. Their model maps roles, users and objects among different domains. However, while they address static role constraints of prerequisite (to be assigned to role B, a user must also be assigned to role A) and conflicting (separation of duties) roles, but they do not consider dynamic RBAC.

Priebe *et al.* [96] extended XACML to specify ABAC (Attribute-Based Access Control) models, in which user access rights are determined dynamically from user attributes. They implemented their model using OWL, SWRL and

SPARQL [97]. This model considers both static (e.g. name) and dynamic (e.g. age, user location) attributes, and therefore goes some way towards supporting dynamic access control using Semantic Web. However, it does not consider attributes other than those of users (thus it neglects, for example, object attributes and environmental conditions), or the application of dynamic attributes to RBAC models.

Finin *et al.* [98][99][100] used OWL to model a static RBAC hierarchy with static and dynamic separation of duties, and positive and negative authorizations (permissions and denials). They also discussed static RBAC constraints of coupling (where a user must be in both role A and role B, or in neither) and exclusive assignment (where each user can only be assigned to one role), and used N3Logic to enforce these rules, as well as for enforcing sessions. They discussed the pros and cons of different approaches to modelling RBAC roles, namely as classes or as values. Modelling roles as classes means that inheritance is expressed naturally, and reasoning is easy, but the specification is complex. When modelling roles as values, inheritance is expressed using rules. This makes reasoning difficult, but simplifies specification. Their work provides an extensive discussion of modelling static RBAC in OWL, and also considers dynamic access control, discussed as ABAC. Their model is called R*OWL*BAC. However, they do not discuss reasoning.

Helili *et al.* [101] presented an RBAC meta-model with negative authorization (called RBAC(*N*)), formalized it in OWL-DL (with roles as classes), and discussed various cases where conflicts can occur between positive and negative authorization in a hierarchical RBAC model. However, again, they did not consider dynamic RBAC rules.

Cirio *et al.* [102] developed a context-aware (dynamic) RBAC model using OWL-DL, queried using SPARQL; their model is combines RBAC with ABAC, so that users are assigned roles at access time according to their attributes. This approach contrasts with our favoured approach, which is to assign roles statically to users and dynamically determine access given to roles according to object attributes and the environment. SPARQL has one major benefit when reasoning in access control modelling, in that it uses a closed-world assumption, and so can be used to query an ontology in a similar manner to predicate logic. The authors also provided a proof-of-concept implementation of their model written in Java. He *et al.* [103] also described a dynamic RBAC model, like [102] using a combination of RBAC and ABAC, and written in OWL-DL, but their model uses SWRL rather than SPARQL as the reasoning language. They adopted Protégé and Jess rule engine as the ontology processing tool and reasoning system, respectively. They also wrote a proof-of-concept implementation of their model in Java.

Calero *et al.* [104] describe the development of an RBAC model from the CIM (Common Information Model) [105], an open standard for representing managed elements of an IT environment, into OWL-DL for use in distributed computing systems. They first wrote a representation of CIM in RDF/OWL, then developed an RBAC authorisation model in OWL and SWRL to be used with it. Their RBAC model supports dynamic RBAC and separation of duties.

Cadenhead *et al.* [106] proposed a scalable TRBAC model for distributed computing systems, written in OWL-DL and using SWRL and SPARQL for reasoning. They achieve scalability by partitioning the DL knowledge base a set of smaller knowledge bases, which have the same TBox (Terminology Box: statements that define terms that model a domain in an ontology) but a subset of the original ABox (Assertion Box: statements that define instances in an ontology). This allows reasoning on subsets of the ontology, because in an OWL-DL model, the number of instances grows in a model while the terms in the ontology largely remain the same. This approach might work for our SO-RBAC model, which is modelled in OWL-DL, but might not work so well for ESO-RBAC, which is modelled in OWL-FULL, and which defines roles as classes, and therefore has to define the relationships between roles in TBox statements.

Coma *et al.* [107] modelled OrBAC (Organization Based Access Control) [108] using OWL-DL. OrBAC differs from RBAC in that it not only abstracts *subjects* (users) into *roles* (sets of subjects), but also abstracts actions into *activities* (sets of *actions*) and objects into *views* (sets of *objects*). This abstraction is hierarchical, so that roles, views and activities can all be sub-classed. The hierarchy of roles in OrBAC is an 'is-a' hierarchy, rather than a seniority hierarchy, describing types of roles rather than superordinate and subordinate relationships. OrBAC natively supports context-aware access control, with a hierarchy of contexts. The hierarchical nature of OrBAC seems to make it naturally suited to modelling in OWL. The authors of [107] demonstrated their OrBAC model in the peer-to-peer collaboration environment.

Toninelli *et al.* [109] developed a dynamic access control model that combines OWL-DL with predicate logic The DL reasoning is used in static RBAC, with dynamic constraints being programmed using predicate logic. This is an attempt to combine the best of both worlds, with OWL being used to classify objects and contexts, and predicate logic being used to determine the results of dynamic querying.

We have already mentioned that traditional static RBAC is difficult to apply in context-aware applications, which appear in pervasive computing spaces, since it fixes a user's access privileges when the user logs on.

4.3 Conclusion

The power of OWL reasoner-enabled ontologies allows us to describe and manipulate the semantics of RBAC differently, and consequently address the previous two problems efficiently. Other works have attempted to use OWL to model RBAC, but they do not exploit the ability of the OWL hierarchy to model hierarchical relationships that are naturally part of an RBAC model. This may be due to the inherent limitations of OWL-DL, which those works use for their models. However, it means that they do not fully exploit the semantics of OWL when modelling RBAC, and retain some of the drawbacks of RBAC models based on predicate logic.

An approach is needed that uses the natural hierarchy of OWL to model hierarchical relationships in both RBAC rules and the data on which these rules operate. The proposed SO-RBAC and ESO-RBAC aim to do this. SO-RAC is an OWL-based RBAC model, written in OWL-DL for ease of translation from Prolog to OWL. As such, it does not represent a major breakthrough in approach to modelling RBAC, but is done as a stepping stone to prove the feasibility of modelling RBAC in OWL. ESO-RBAC represents a complete rewrite of the model in OWL-Full. It uses the class-individual duality of OWL-Full to define RBAC role inclusion using OWL sub-classes, rather than having to define it in object properties and preform reasoning on them. This represents a novel way of exploiting OWL and its reasoners for the purpose of defining and manipulating the semantics of RBAC. The semantic ontological reasoning processes defined in ESO-RBAC, which are domain and implementation independent, can be run from any distributed computing environment. These can then be developed through integrated development environments such NetBeans and using OWL APIs.

The following Chapters (5 and 6) describe SO-RBAC and ESO-RBAC, respectively.

5 The Proposal: Semantic and Ontology-based Role-Based Access Control (SO-RBAC)

5.1 Introduction

This chapter describes the proposed **S**emantic and **O**ntology-based **R**ole-**B**ased **A**ccess **C**ontrol (SO-RBAC) process for creating permissions and denials based upon a user's roles and the activities that the user may perform on a selection of objects. In other words, the process uses the semantics stored in the SO-RBAC ontology in terms of manipulating its ontological concepts and their individuals for the purpose of determining if a particular user, who holds a particular "role" is allowed to access an "object" and perform a particular "activity" upon it and therefore would be granted permission for the activity denied access to the object.

SO-RBAC is the first step in modelling RBAC using Semantic Web technology, as suggested in Section 4.2.2. The model is essentially a direct translation of the Prolog rules for static RBAC in Section 3.2.1 into OWL. Permissions and denials are given similar to traditional RBAC supported by Prolog facts and rules and functionalities of database management systems in controlling access control in databases. This use of Prolog rules as a basis means that it cannot address the complexity of traditional RBAC models. In some respects, the differences between ontologies and predicate logic introduce additional complexity. For this reason, we chose not to model context-aware or dynamic RBAC using SO-RBAC, although it would be possible to do so.

The purpose of SO-RBAC was not to create a pure ontological RBAC model, but to demonstrate the feasibility of mapping an RBAC model based on Prolog facts and rules into an ontology. Therefore, the proposed SO-RBAC is not designed from 'scratch'. It is instead based on a set of existing Prolog facts and rules, which are translated into an ontological schema. Prolog facts are modelled as instances within OWL classes, or as properties of these classes. RBAC rules are modelled through domain and range constraints, is-a relationships and inheritance, or using SWRL [17][95] rules.

Although SO-RBAC model has its roots in predicate logic, it models RBAC using OWL ontological concepts, and reasons upon these to strengthen the semantics stored in an ontology, and to manipulate individuals of ontological concepts for making decisions on denials and permissions. Consequently, the SO-RBAC process and ontological model are suitable for any repository where a user may have roles and may not necessarily be involved with the manipulation of database elements. However, the SO-RBAC ontological model is also generic enough to accommodate data structures from any domain, and our mechanism of reasoning allows successful manipulation of ontological individuals which characterise a particular instance of SO-RBAC and its process.

Section 5.2 demonstrates the SO-RBAC ontological model and reasoning. The section is divided into three subsections.

Section 5.2.1 defines the SO-RBAC ontological model through three distinctive steps:

(a) Definition of OWL classes and their hierarchies

(b) Definition of Necessary & Sufficient conditions and

(c) Definitions of object properties.

Section 5.2.2 describes the way of populating SO-RBAC classes with individuals by assertion. That section explains exactly which classes must be populated before the reasoning process starts and why. Consequently, a portion of SO-RBAC ontological classes will remain 'empty' until a reasoning process determines which individuals from the asserted classes will be 'moved' (or copied) into SO-RBAC classes which were empty on SO-RBAC initialisation.

Section 5.2.3 explains the purpose and the outcome of the reasoning process upon SO-RBAC concepts using SWRL. SO-RBAC has two types of reasoning. The first reasoning step, described in 5.2.3.1, uses SWRL for creating a set of new object properties which use existing object properties defined in step (c). All of the object properties for which this is done have ROLE class as both domain and range, as the purpose of this step is to set up all the relationships between roles in the RBAC model. The second step, described in Section 5.2.3.2, performs reasoning to move individuals across SO-RBAC in order to determine permission or denials in particular request imposed by a user, who has a 'role' and would like to perform an 'activity' upon set of 'objects'.

```
% inclusion of equal-status roles
included_in(R,R) :- role(R).
included_in(R1,R2) :- is_a(R1,R2).
included_in(R1,R3) :- is_a(R1,R2),
                      included_in(R2,R3).

% Role hierarchies
senior_to(R,R) :- directly_senior_to(R,_).
senior_to(R,R) :- directly_senior_to(_,R).
senior_to(R1,R2) :- directly_senior_to(R1,R2).
senior_to(R1,R3) :- directly_senior_to(R1,R2), senior_to(R2,R3).

% Inheritance paths
inherits_pra(R,R) :- role(R).
inherits_pra(R2,R3) :- senior_to(R1,R2),
                       senior_to(R3,R4),
                       inherits_pra_path(R1,R4).

% Access control rules structure
pra_full(R1,P,O) :- senior_to(R1,R2),
                    pra(R2,P,O),
                    inherits_pra(R1,R2).

permittable(U,P,O_instance) :- ura(U,R1),
                       included_in(R1,R2)
                       instance_of(O_instance,O),
                       pra_full(R2,P,O).

permitted(U,P,O) :-    active_user_session(U),
                       permittable(U,P,O).

dra_full(R2,P,O) :- senior_to(R1,R2),
                    dra(R1,P,O).

denied(U,P,O_instance) :- ura(U,R1),
                          included_in(R1,R2),
                          instance_of(O_instance,O),
                          dra_full(R2,P,O).

authorizable(U,P,O) :- permittable(U,P,O),
                       not(denied(U,P,O)).

authorized(U,P,O) :- permitted(U,P,O),
                     not(denied(U,P,O)).
```

Code 53: Prolog rules on which the SO-RBAC model is based

Section 5.3 describes the SO-RBAC process and explains its steps, which are based on the model and reasoning introduced in Section 5.2.

Section 5.4 contrasts the proposed SO-RBAC solution with the traditional RBAC defined in Prolog (Code 53).

Section 5.5 gives a particular scenario of RBAC in terms of defining which individuals may populate one of SO-RBAC instances. The healthcare domain and a medical database is used to demonstrate the implementation of SO-RBAC.

Section 5.6 describes the implementation of SO-RBAC reasoning and the deployment of the SO-RBAC process. The SO-RBAC ontology is modelled in OWL-DL. Although OWL-DL is much less flexible than OWL-FULL in ontological modelling, it has a much wider range of available reasoners. SO-RBAC was modelled using Protégé [24], with SWRL rules defined using the Protégé SWRLTab [110]. The model was initialized using a Perl script to create the initial instances.

Section 5.7 shows screen shots from Protégé of the implementation and testing of SO-RBAC.

Section 5.8 draws conclusions.

5.2 Ontological Model and Reasoning

This section describes the SO-RBAC in terms of OWL and SWRL.

5.2.1 Definition of SO-RBAC Ontological Model

5.2.1.1 OWL classes and their hierarchies

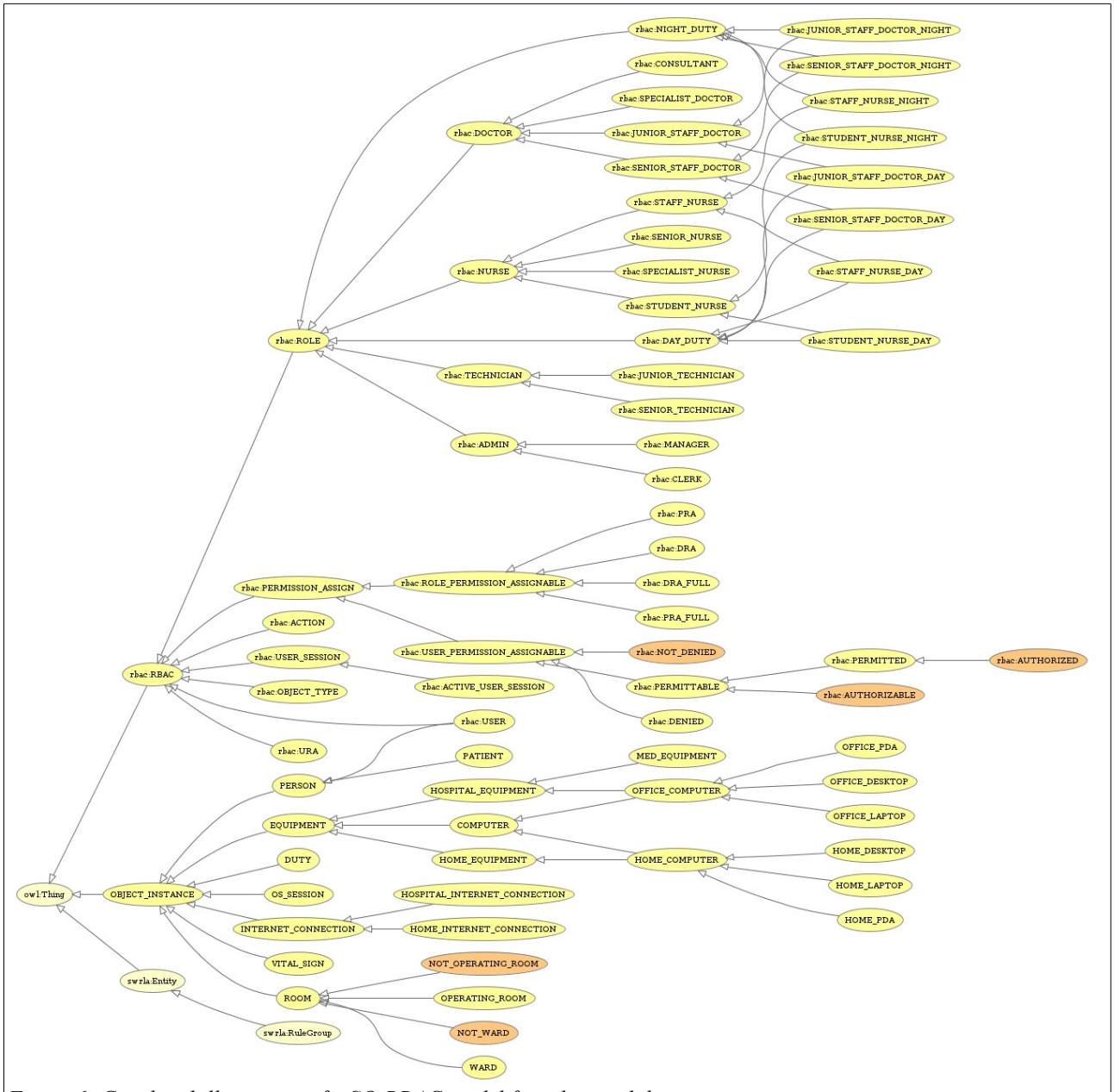

Figure 6: Graphical illustration of a SO-RBAC model for a hospital domain.

Figure 6 shows a graphical illustration of SO-RBAC. There are two main super-classes in SO-RBAC: OBJECT_INSTANCE and RBAC.

The super-class OBJECT_INSTANCE defines objects that may be accessed by users in SO-RBAC; examples of these include database tables, files and equipment. The RBAC administrator is free to define the sub-classes of

OBJECT_INSTANCE according to the domain. We have sub-classed it based on a simplified model of data and systems in a hospital, which is further explained in the scenario and implementation of SO-RBAC, as mentioned in Introduction (Section 1).

The super class RBAC defines concepts that are relevant to RBAC, which should be stored in a separate super-class from OBJECT_INSTANCE because it is conceptually different from other information, and is typically stored separately in other systems. For example, a relational DBMS would store the RBAC information as meta-data, which is not usually queried directly by users.

Sub-classes of the OBJECT_INSTANCE class are:

- EQUIPMENT: represents all machines, both computers and medical equipment (and possibly others) to which a user might be logged in. There are various sub-classes of EQUIPMENT, and multiple inheritance is used.
- INTERNET_CONNECTION: represents Internet settings of computers. This class is sub-classed into HOME_INTERNET_CONNECTION and HOSPITAL_INTERNET_CONNECTION.
- OS_SESSION represents operating system login settings of computers.
- PERSON represents all individuals with information stored about them. This includes users, so the class USER is a sub-class of this as well as of RBAC. The other sub-class of PERSON in this example is PATIENT.
- ROOM represents all rooms in a hospital, and is sub-classed into OPERATING_ROOM and WARD.
- VITAL_SIGNS represents vital signs recorded for patients.

Sub-classes of the RBAC class are:

- The USER sub-class defines the set of users of the system. However, USER also inherits from PERSON, which is a subclass of the OBJECT_INSTANCE class. On a superficial level, this is because user information might be stored both as ordinary data and as meta-data in a relational database. On a practical level, it is because the USER class, describing a user, contains information about users that is used in either ordinary information-retrieval situations or in RBAC processing, or both.
- ROLE sub-class contains a complex hierarchy of sub-classes, defining roles to which users and permissions may be assigned. The hierarchy of classes under ROLE represents sub-divisions of roles by type (not by seniority). The RBAC administrator is free to sub-class this class according to the domain. In this example, it is sub-classed according to roles that might be found in a hospital. The main sub-classes of ROLE in this example are DOCTOR, NURSE, ADMIN, TECHNICIAN, DAY_DUTY and NIGHT_DUTY. These sub-classes are further sub-classed, including multiple inheritance.
- USER_SESSION defines user login sessions. Its sub-class ACTIVE_USER_SESSION defines user login sessions that are active, and thus give permissions to users.
- OBJECT_TYPE defines the types of object that can be manipulated by SO-RBAC (as opposed to the objects themselves, which are in OBJECT_INSTANCE).
- URA sub-class defines user-role assignments.
- ACTION class defines actions that can be performed on objects, such as *read* and *write*.

PERMISSION_ASSIGN is a sub-class consisting of all classes that relate to permission assignments. However, it is also an abstract class in SO-RBAC, *i.e.* it never contains any instances directly assigned to it. It is defined to

provide the *role* and *action* properties to all permission-assignment classes in SO-RBAC. Its subclasses are ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE. They define permission assignments between users and objects, and between roles and objects. ROLE_PERMISSION_ASSIGNABLE defines permissions and denials assigned to roles, either explicitly or computationally by SO-RBAC. USER_PERMISSION_ASSIGNABLE defines permissions, authorizations and denials assigned to users by SO-RBAC computations.

The sub-classes of USER_PERMISSION_ASSIGNABLE are DENIED, NOT_DENIED, PERMITTABLE, AUTHORIZABLE, PERMITTED and AUTHORIZED. All these sub-classes, except NOT_DENIED, are equivalent to the similarly-named Prolog predicates. NOT_DENIED is the complement of DENIED. PERMITTED is defined as a sub-class of PERMITTABLE, because it can only contain individuals that are also in this.

The sub-classes of ROLE_PERMISSION_ASSIGNABLE are DRA, DRA_FULL, PRA and PRA_FULL, all of which are equivalent to the similarly-named Prolog predicates. PRA defines explicit role-permission assignments. PRA_FULL defines role-permission assignments that are inferred when the SO-RBAC model is run. Similarly, DRA defines explicit role-denial assignments, and DRA_FULL defines inferred role-denial assignments.

URA sub-class defines user-role assignments, and has two properties, *user* and *role*. Since URA is a binary predicate, it could just as easily be defined as a property. It is defined as a class to maintain the analogy with PRA, in the Prolog RBAC model. PRA is a ternary predicate, and therefore has to be defined as a class. Additionally, defining URA as a class mean that user-role assignments can be seen more easily in the model than if it were defined as an object property.

ACTION class defines actions that can be performed on objects, such as *read* and *write*.

5.2.1.2 Necessary & Sufficient conditions

Table 6: Necessary & Sufficient conditions imposed on SO-RBAC classes

Class	***Necessary & Sufficient condition***
NOT_DENIED	USER_PERMISSION_ASSIGNABLE ⊓ ¬DENIED
AUTHORIZABLE	PERMITTABLE ⊓ ¬DENIED
AUTHORIZED	PERMITTED ⊓ ¬DENIED

It is important to note that we had to impose a few Necessary & Sufficient conditions upon a selection of SO-RBAC classes in order to guarantee consistency of SO-RBAC when populating classes with individuals. In other words Necessary & Sufficient conditions are imposed on NOT_DENIED, AUTHORIZABLE and AUTHORIZED (see Table 6). If a class has a Necessary & Sufficient condition imposed on it, then populating the class in a way that violates this condition makes the ontology inconsistent. The SO-RBAC reasoning process populates these classes in a way that would always be consistent with the conditions.

In Figure 6 (page 62), the graphical illustration of SO-RBAC, OWL classes are in yellow, except classes bound by Necessary & Sufficient conditions, which are in amber.

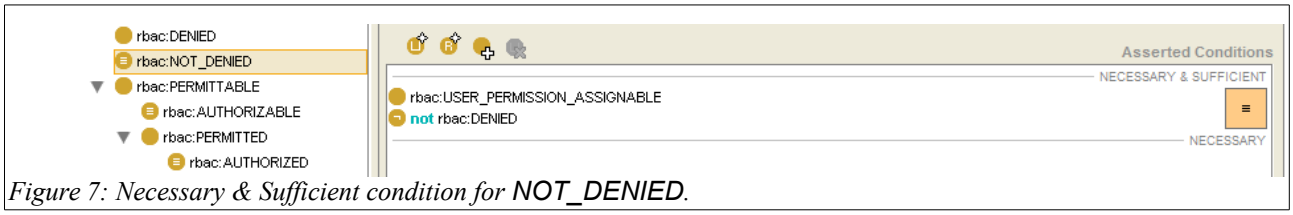

Figure 7: Necessary & Sufficient condition for NOT_DENIED.

Figure 7 shows how a Necessary & Sufficient condition appears in Protégé. As this figure shows, these Necessary & Sufficient conditions cause AUTHORIZABLE to become a sub-class of PERMITTABLE, and AUTHORIZED to become a sub-class of PERMITTED.

5.2.1.3 Object property relationships

Object properties between SO-RBAC classes are defined according to two reasons.

(a) Certain SO-RBAC classes rely on object properties to define semantically the individuals that they, or their sub-classes, may contain.

(b) Certain SO-RBAC classes allow definition of object properties between them to strengthen the semantics of SO-RBAC, as determined by our OWL modelling principles.

The next three paragraphs cover examples of (a).

We have already mentioned above that PERMISSION_ASSIGN provides the *role* and *action* properties to all permission-assignment classes in SO-RBAC. Therefore its full description must include object properties it holds. Naturally, PERMISSION_ASSIGN has object properties *role* and *action*. Just as all permission assignment predicates in the Prolog RBAC model described in 3.2.1 have role and action as arguments, so do all analogous classes in SO-RBAC. However, the hierarchical nature of ontologies makes it much easier to define a series of related classes with the same properties in an ontology than it is to define predicates with similar arguments in Prolog. In OWL, property inheritance can be used to define a super-class with certain properties, and define sub-classes representing related predicates that inherit its object properties. Accordingly, PERMISSION_ASSIGN sub-classes ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE, both inherit properties *role* and *action*, as well as defining other object properties. ROLE_PERMISSION_ASSIGNABLE has the additional property *object_type*. Since DRA, DRA_FULL, PRA and PRA_FULL are sub-classes of ROLE_PERMISSION_ASSIGNABLE, all have the object properties *role*, *action* and *object_type*. *role* and *action* are inherited from PERMISSION_ASSIGN (their grandparent super-class), while *object_type* is inherited directly from ROLE_PERMISSION_ASSIGNABLE.

USER_PERMISSION_ASSIGNABLE defines permissions, authorizations and denials assigned to users by SO-RBAC computations. As well as inheriting *role* and *action* from PERMISSION_ASSIGN, it also has the object properties *user* and *object_instance*.

URA sub-class has two properties, *user* and *role*. Since URA is a binary predicate, it could just as easily be defined as a property. It is defined as a class to maintain the analogy with PRA, in the Prolog RBAC model. PRA is a ternary predicate, and therefore has to be defined as a class. Additionally, defining URA as a class mean that user-role assignments can be seen more easily in the model than if it were defined as an object property.

We have introduced a few object properties in the previous section in order to explain the purpose and existence of some sub-classes of the RBAC super-class. However, in order to perform and guarantee successful outcome of

reasoning, we have strengthened the semantics of SO-RBAC with additional set of object properties imposed on SO-RBAC classes as highlighted in (b) above.

Table 7: Object properties in SO-RBAC

Domain	***Property***	***Description***	***Range***
rbac:PERMISSION_ASSIGN (sub-classes: rbac:USER_PERMISSION_ASSIGNABLE and sub-classes, rbac:ROLE_PERMISSION_ ASSIGNABLE and sub-classes)	*rbac:action*	Actions involved in role and user permission assignments.	rbac:ACTION
rbac:PERMISSION_ASSIGN and sub-classes	*rbac:role*	Roles involved in role and user permission assignments.	rbac:ROLE and sub-classes
rbac:USER_PERMISSION_ASSIGNABLE (sub-classes: rbac:DENIED, rbac:NOT_ DENIED, rbac:PERMITTABLE, rbac: AUTHORIZABLE, rbac:PERMITTED, rbac: AUTHORIZED)	*rbac:object_ instance*	Object instance to which a user is permitted, authorized or denied access.	OBJECT_INSTANCE and sub-classes
rbac:USER_PERMISSION_ASSIGNABLE and sub-classes	*rbac:user*	Users involved in user permission/denial/authorization assignments.	rbac:USER
rbac:ROLE_PERMISSION_ASSIGNABLE (sub-classes: rbac:DRA_FULL, rbac:DRA, rbac:PRA_FULL, rbac:PRA)	*rbac:object_type*	Object types associated with PRA and DRA relationships.	rbac:OBJECT_TYPE
rbac:ROLE_PERMISSION_ASSIGNABLE and sub-classes	*rbac:role*	Roles associated with PRA and DRA relationships.	rbac:ROLE and sub-classes
OBJECT_INSTANCE and sub-classes	*rbac:instance_of*	An instance of a type of object, as defined by a sub-class of rbac:OBJECT_TYPE.	rbac:OBJECT_TYPE
rbac:URA	*rbac:role*	A role in a URA assignment.	rbac:ROLE
rbac:URA	*rbac:user*	A user in a URA assignment.	rbac:USER
rbac:USER_SESSION	*rbac:user*	A user attached to a session.	rbac:USER
rbac:ROLE	*rbac:directly_ junior_to*	Inverse of *directly_senior_to*. Sub-property of *junior_to*.	rbac:ROLE
rbac:ROLE	*rbac:directly_ senior_to*	Assertions of direct seniority relationships. Sub-property of *senior_to*.	rbac:ROLE
rbac:ROLE	*rbac:included_in*	Direct and indirect inclusion relationships, inferred from *is_a* relationships.	rbac:ROLE
rbac:ROLE	*rbac:inherits_pra*	Roles that participate in inheritance paths, inferred from *inherits_pra_path*.	rbac:ROLE
rbac:ROLE	*rbac:inherits_ pra_path*	Assertions of ends of inheritance paths.	rbac:ROLE
rbac:ROLE	*rbac:is_a*	Assertions of direct inclusion relationships. Sub-property of *senior_to*.	rbac:ROLE
rbac:ROLE	*rbac:junior_to*	Inverse of *senior_to*.	rbac:ROLE
rbac:ROLE	*rbac:senior_to*	Direct and indirect seniority relationships, inferred from *senior_to*.	rbac:ROLE

Table 7 lists ALL object properties with their Domains and Ranges, and includes both asserted and inherited object properties.

Most object properties are named after their Ranges. These properties may have different functions depending on the Domain: each function of a property is listed separately in the table.

Object properties not named after their Ranges are *instance_of* and the properties that have ROLE as both Domain and Range (*directly_junior_to*, *directly senior_to*, *included_in*, *inherits_pra*, *inherits_pra_path*, *is_a*, *junior_to*, *senior_to*).

instance_of, which links object types to instances, is separate from the *object_type* property of PERMISSION_ASSIGN and its sub-classes, which is involved in role-permission assignments. This is because SO-RBAC could be used to manage access to classes relating to RBAC. If this is done, then the object type to which PERMISSION_ASSIGN and its sub-classes belong must be defined; to do so using the same property as is used in assignment relations would cause confusion.

It is important to note that some object properties from Table 7 are *asserted* and some of them are *inferred*. For example, the object properties *action*, *user*, *role* and *object_instance* are *asserted* between USER_PERMISSION_ASSIGNABLE and OBJECT_INSTANCE, ACTION, USER and ROLE (and its sub-classes), but *inferred* between all subclasses of USER_PERMISSION_ASSIGNABLE and these classes.

Similarly, the object properties *action*, *object_type* and *role* are *asserted* between ROLE_PERMISSION_ASSIGNABLE and ACTION, OBJECT_TYPE and ROLE (with its subclasses) classes, but *inferred* between all subclasses of ROLE_PERMISSION_ASSIGNABLE and these classes.

```
+ OBJECT_INSTANCE
- RBAC
  - ACTION
  - OBJECT_TYPE
  + ROLE {directly_junior_to ROLE, directly_senior_to ROLE, junior_to ROLE, senior_to ROLE,
included_in ROLE, inherits_pra ROLE, inherits_pra_path ROLE, is_a ROLE}
  - PERMISSION_ASSIGN {action ACTION, role ROLE}
    - ROLE_PERMISSION_ASSIGNABLE {object_type OBJECT_TYPE}
      - DRA
      = DRA_FULL
      - PRA
      = PRA_FULL
    - USER_PERMISSION_ASSIGNABLE {object_instance OBJECT_INSTANCE, user USER}
      - DENIED
      ≡ NOT_DENIED {USER_PERMISSION_ASSIGNABLE ⊓ ¬DENIED}
      = PERMITTABLE
        ≡ AUTHORIZABLE {PERMITTABLE ⊓ ¬DENIED}
        = PERMITTED
          ≡ AUTHORIZED {PERMITTED ⊓ ¬DENIED}
  - URA {role ROLE, user USER}
  - USER
  - USER_SESSION {user USER}
    - ACTIVE_USER_SESSION
```

Text 4: SO-RBAC Ontology (some classes are collapsed).

```
+ COLLAPSED_CLASS
- CLASS
   - SUB-CLASS {object_property_1 CLASS, object_property_2 CLASS}
   - ABSTRACT_CLASS
= SWRL-INFERRED_CLASS
   ≡ N&S_BOUND_CLASS {N&S CONDITION (⊓: and; ¬: not}
```

Text 5: Legend for SO-RBAC Ontology.

Text 4 illustrates a collapsed version of the ontology from Figure 6, and highlights main SO-RBAC classes involved in ontological reasoning. It is important to note that Text 4 should be read in conjunction with Text 5. Object properties are listed, with their ranges, in grey text in curly brackets after the classes that have them as their domains. Classes that contain inferred individuals as the result of our ontological reasoning are listed in blue and preceded by the = symbol. Classes on which Necessary & Sufficient conditions are imposed are in green and preceded by the ≡ symbol. The key to the colours and symbols is shown in Text 5.

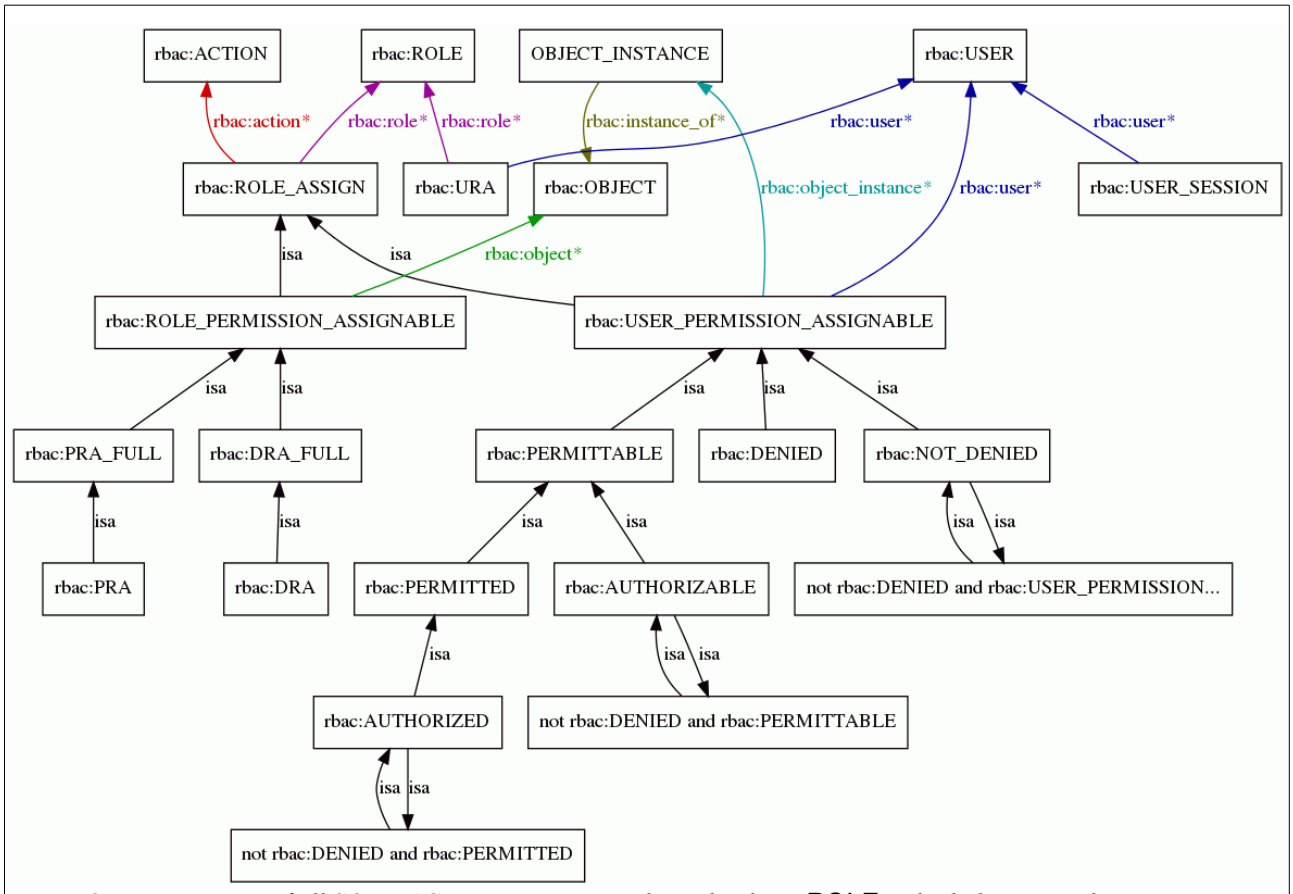

Figure 8: Property map of all SO-RBAC properties except those that have ROLE as both domain and range.

Figure 8 graphically illustrates all object properties defined in Table 7 except those that have ROLE as both domain and range. The label 'isa' in Figure 8 refers to the sub-class–super-class relationship: a sub-class 'isa' super-class. It has nothing to do with the *is_a* property used in SO-RBAC. In this diagram, each property is distinguished by colour: where the same property appears several times, it is shown in the same colour. However, these colours are not used anywhere else.

There are several ways in which ROLE individuals can be related affecting user-permission assignment in RBAC. These need separate attention.

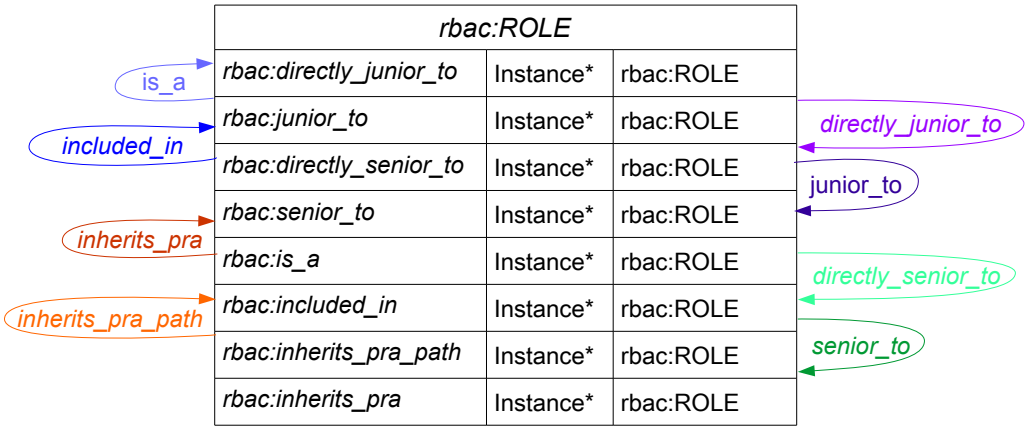

Figure 9: Property map of all SO-RBAC properties with ROLE as both domain and range.

Figure 9 depicts all object properties in SO-RBAC that have ROLE as both domain and range. These properties are *directly_junior_to*, *directly_senior_to*, *included_in*, *inherits_pra*, *inherits_pra_path*, *is_a*, *junior_to* and *senior_to*. These separate properties represent different relationships between ROLE instances, as described in Table 7. Each object property relating instances of the same class is indicated by an arrow from the node representing the ROLE class and pointing back to this box. For clarity, these object properties are also listed in the node. The box in Figure 9 signifies that a ROLE class instance (represented by the ROLE at the top of the box) has can be linked to any instances of ROLE via any of the properties listed.

Note that all object properties in Figure 9 apply to the same ROLE class. Therefore, they appear in Figure 9 twice: in the first column of the figure and as coloured labels of arcs which graphically illustrate these object properties defined upon class ROLE.

5.2.2 Populating SO-RBAC classes by assertion

Classes populated in this stage are classified into two types. Note that ROLE and PERMISSION_ASSIGN are abstract classes, which contain no asserted individuals.

i. Auto-populated on initialization: ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE are populated on initialization with individuals representing possible role and user permission assignments. Individuals asserted under these classes are not active: they have to be moved to sub-classes of these classes to be active in SO-RBAC.

ii. Populated according to RBAC model on initialization: URA, USER, ACTION, ROLE, OBJECT_TYPE, USER_SESSION, ACTIVE_USER_SESSION, DRA, PRA and all classes under ROLE and OBJECT_INSTANCE are populated, by the RBAC administrator and application, with individuals that define the RBAC rules and environment.

Individuals in the OBJECT_TYPE class specify types of object, linked to the appropriate individuals in OBJECT_INSTANCE as the range of the property *instance_of*.

Individuals in ROLE specify RBAC roles. In the example RBAC, ROLE is an abstract class, and has a hierarchy below this indicating types of role such as DOCTOR and NURSE, but this hierarchy is not used by the

present implementation of SO-RBAC, because OWL-DL cannot address classes directly. Instead the properties *is_a* and *included_in* represent role inclusion. In the example R BAC, each class has a single individual representing a role; however, this is not necessary in SO-RBAC. The RBAC roles are defined by the individuals in ROLE and its sub-classes, not by the classes themselves. The ROLE class hierarchy is defined for illustration only, as it has no semantic significance in SO-RBAC.

ROLE_PERMISSION_ASSIGNABLE is populated on initialization with individuals representing all possible relationships between roles, actions and object types. These are then moved in the reasoning step into any of the sub-classes.

The sub-classes of ROLE_PERMISSION_ASSIGNABLE are DRA, DRA_FULL, PRA and PRA_FULL. DRA and PRA are populated on initialization with explicit denial and permission assertions, respectively, copied from the super-class. They are exactly equivalent to `dra` and `pra` assertions in the Prolog model.

USER_PERMISSION_ASSIGNABLE is populated on initialization with individuals representing all possible combinations of user assignments of access to perform actions on object instances.

URA is populated with individuals describing user-role assignments.

The USER class is populated by individuals representing users who may be assigned roles.

The USER_SESSION class is populated with user login sessions. It contains sub-class ACTIVE_USER_SESSION, which represents active user sessions.

5.2.3 Reasoning in SO-RBAC using SWRL

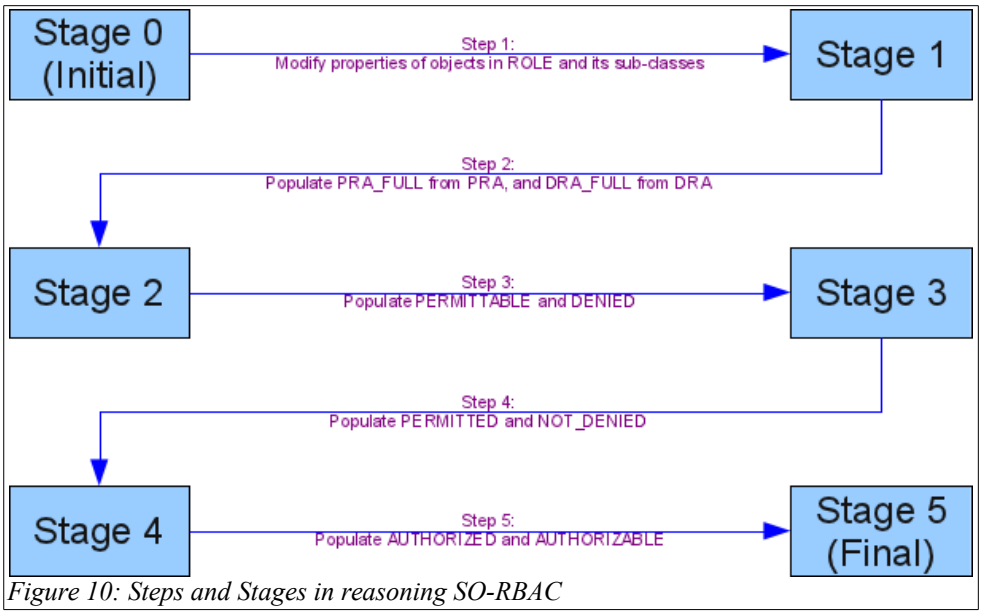

Figure 10: Steps and Stages in reasoning SO-RBAC

Figure 10 shows the five steps of reasoning. Step 1 significantly differs from the others because it uses SWRL for inferring more object properties. In other words, Step 1 modifies object properties in individuals in ROLE and its sub-classes for the purpose of determining all the relationships between roles within RBAC.

Steps 2–5 of the reasoning process infer individuals in SO-RBAC classes according to strictly defined matching of SO-RBAC sub-classes. The final result of our reasoning through SWRL and ontological matching will be shown in Stage 5, when certain individuals will be moved into SO-RBAC classes AUTHORIZABLE and AUTHORIZED.

Step 1 is shown in Sub-section 5.2.3.1, and Steps 2–5 are shown in Sub-section 5.2.3.2.

The steps are designed such that each stage populated the ontology with all axioms that may be required for the immediately following stage (except that Step 1 creates all object property relationships).

The reader should be aware that if more than one rule affects the same class or property, then the relationship between the rules is a logical OR (this cannot be represented any other way in SRWL).

The SWRL rules were named according to the following conventions:

- The rules are numbered according to the step in which they are executed when rule chaining. There are five steps, 1–5 (Figure 10).
- The SWRL rules are named according to the convention s_relation[_n], where s is the step number, relation is the class or property affected by the rule, and n is a sequence number (if there is more than one rule relating to the same relation in the same stage).

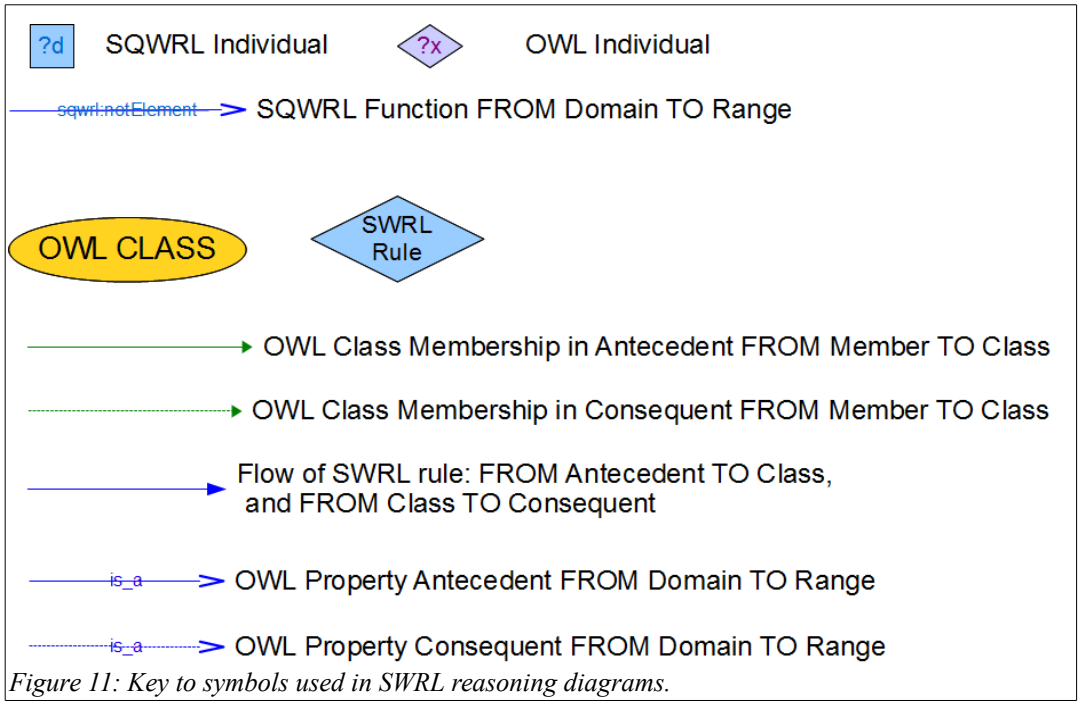

Figure 11: Key to symbols used in SWRL reasoning diagrams.

Figure 11 shows the key to the symbols in the diagrams in Figs. 12–29 showing the inference processes for object properties. The shapes used for OWL individuals and classes are intentionally similar to those used for equivalent entities in Protégé. In SWRL syntax, variables always begin with a ? symbol, and this convention is followed in the diagrams.

SQWRL (Semantic Query-enhanced Web Rule Language) [111][112][113] is a SWRL-based query language that can be used to query OWL ontologies. It is used in Step 4 to provide semantics needed in this step that are not available in SWRL itself. OWL object properties and SQWRL functions are represented as arrows from the Domain to the Range, with the name of the property or function (in the legend, *is_a* and *sqwrl:notElement*) is used as an example) appearing over the arrow.

5.2.3.1 Defining new object properties

We define new object properties in Step 1, from Figure 10. These definitions are based on previously defined object properties, where the ROLE class is the Range and Domain. Step 1 consists of 7 SWRL rules, named as 1_senior_to_1, 1_senior_to_2, 1_senior_to_4, 1_included_in_1, 1_included_in_3, 1_inherits_pra_1 and 1_inherits_pra_3.

rbac:ROLE(?r) ∧ rbac:directly_senior_to(?_, ?r) → rbac:senior_to(?r, ?r)

Text 6: SWRL for rule 1_senior_to_1.

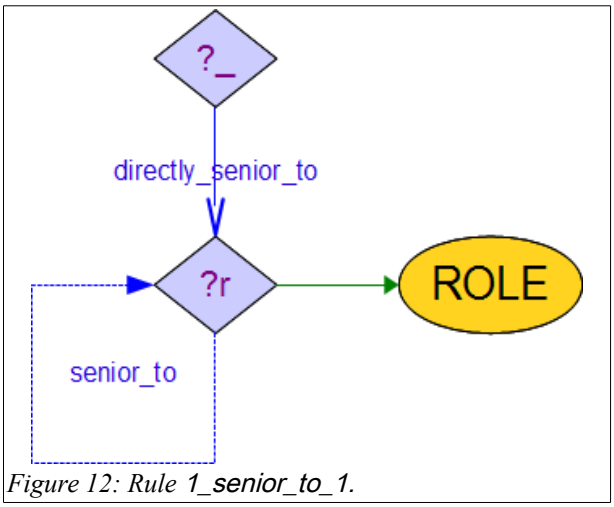

Figure 12: Rule 1_senior_to_1.

The first rule in Step 1, given in Figure 12, is called 1_senior_to_1. It defines a role as is senior to itself if it has at least one role directly senior to it. Figure 12 is converted into SWRL syntax in Text 6 above.

rbac:ROLE(?r) ∧ rbac:directly_senior_to(?r, ?_) → rbac:senior_to(?r, ?r)

Text 7: SWRL for rule 1_senior_to_2.

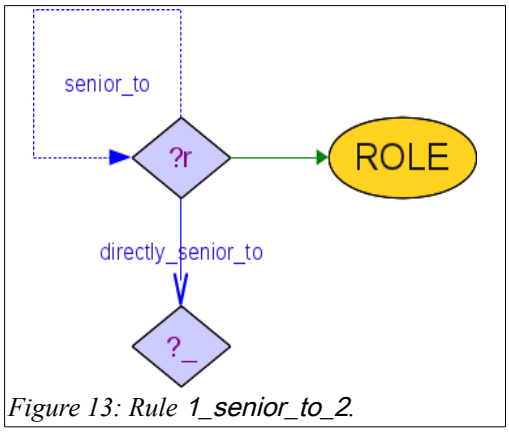

Figure 13: Rule 1_senior_to_2.

The second rule in Step 1, given in Figure 13, is called 1_senior_to_2. It defines a role as senior to itself if it is directly senior to at least one role. Figure 13 is converted into SWRL syntax in Text 7 above.

rbac:directly_senior_to(?r1, ?r2) ∧ rbac:senior_to(?r2, ?r3)

→ rbac:senior_to(?r1, ?r3)

Text 8: SWRL for rule 1_senior_to_4.

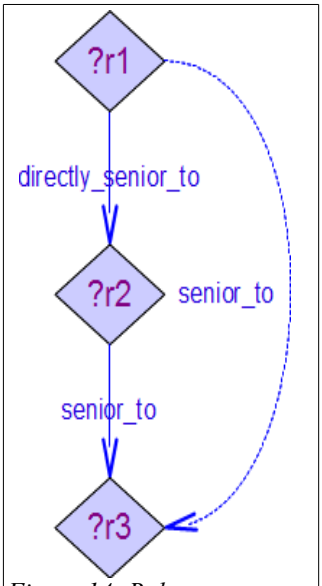

Figure 14: Rule 1_senior_to_4.

The third rule in Step 1, given in Figure 14, is called 1_senior_to_4. It defines a seniority of roles as being transitive. In other words, Role *?r1* is senior to *?r3* if it is directly senior to another role (*?r2*) that is senior to *?r3*. Figure 14 is converted into SWRL syntax in Text 8 above.

rbac:ROLE(?r) → rbac:included_in(?r, ?r)

Text 9: SWRL for rule 1_included_in_1.

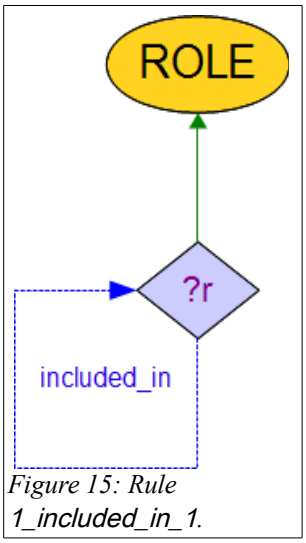

Figure 15: Rule 1_included_in_1.

The fourth rule in Step 1, given in Figure 15, is called 1_included_in_1. It defines a role as always being included in itself. Figure 15 is converted into SWRL syntax in Text 9 above.

```
rbac:is_a(?r1, ?r2) ∧ rbac:included_in(?r2, ?r3)
 → rbac:included_in(?r1, ?r3)
```

Text 10: SWRL for rule 1_included_in_3.

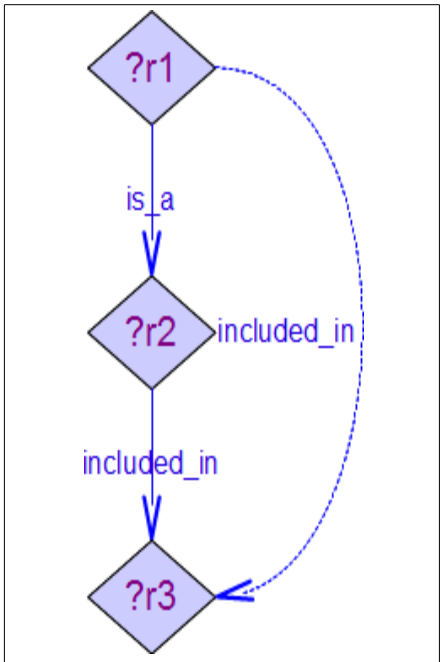

Figure 16: Rule 1_included_in_3.

The fifth rule in Step 1, given in Figure 16, is called 1_included_in_3. It defines role inclusion as being transitive. In other words, Role *?r1* is included in *?r3* if it is directly included in (is_a) another role (*?r2*) that is included in *?r3*. Figure 16 is converted into SWRL syntax in Text 10 above.

```
rbac:ROLE(?r) → rbac:inherits_pra(?r, ?r)
```

Text 11: SWRL for rule 1_inherits_pra_1.

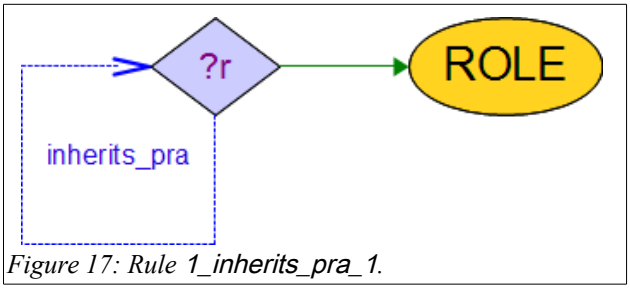

Figure 17: Rule 1_inherits_pra_1.

The sixth rule in Step 1, given in Figure 17, is called 1_inherits_pra_1. It defines a role as being part of an inheritance path involving itself. An inheritance path is a path along which permissions can be inherited. This rule is necessary to set up recursion when defining inheritance paths. Figure 17 is converted into SWRL syntax in Text 11 above.

rbac:senior_to(?r1, ?r2) ∧ rbac:senior_to(?r3, ?r4) ∧
rbac:senior_to(?r3, ?r4) ∧ rbac:inherits_pra_path(?r1, ?r4)
→ rbac:inherits_pra(?r2, ?r3)

Text 12: SWRL for rule 1_inherits_pra_3.

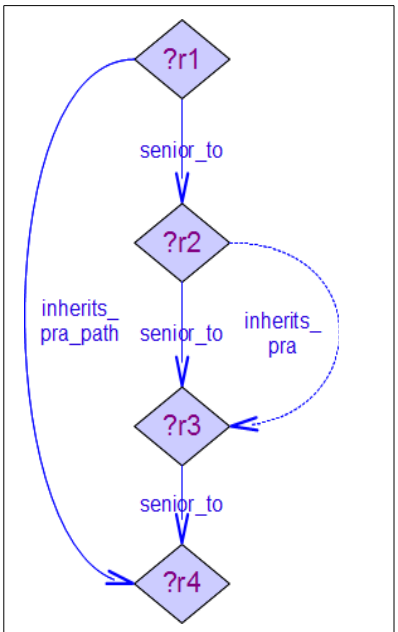

Figure 18: Rule 1_inherits_pra_3.

The seventh rule in Step 1, given in Figure 18, is called 1_inherits_pra_3. It defines that Roles *?r2* and *?r3* are in an inheritance path, where *?r3* is the senior role, if:

i) *?r2* has a senior role *?r1* that is at the senior end of an inheritance path, and

ii) *?r3* is senior to role *?r4* that is at the junior end of an inheritance path.

Figure 18 is converted into SWRL syntax in Text 12 above.

5.2.3.2 Assigning individuals to SO-RBAC classes

Individuals are assigned to SO-RBAC classes by the reasoning rules in Steps 2–5. All rules in Steps 2 and 3, and rule 2 of Step 4, match individuals according to object properties. Rule 1 of Step 4, and both rules in Step 5, match individuals by a simple set operation (set difference or intersection).

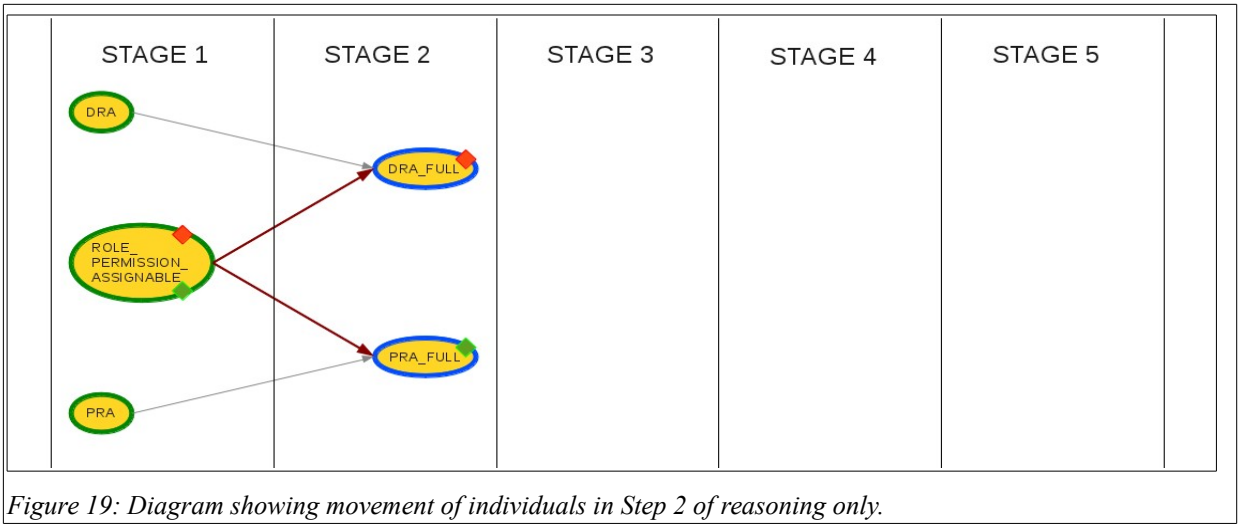

Figure 19: Diagram showing movement of individuals in Step 2 of reasoning only.

Step 2 is shown in Figure 19. It takes class ROLE_PERMISSION_ASSIGNABLE and matches its individuals with individuals of classes PRA and DRA. If individuals from ROLE_PERMISSION_ASSIGNABLE satisfy the rules for their matching, then they are moved to PRA_FULL and DRA_FULL. It is important to note that only individuals from ROLE_PERMISSION_ASSIGNABLE are being moved into PRA_FULL and DRA_FULL, according to the object properties of these and of the individuals in PRA and DRA.

These two matchings are performed though two different SWRL rules *2_dra_full* and *2_pra_full*. Both of these rules are explained separately through written explanations and diagrams which show object properties responsible for ontological matching and the way we populate classes with inferred individuals.

rbac:DRA(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧
rbac:object_type(?x, ?o) ∧ rbac:senior_to(?r1, ?r2) ∧
rbac:ROLE_PERMISSION_ASSIGNABLE(?z) ∧
rbac:role(?z, ?r2) ∧ rbac:action(?z, ?a) ∧ rbac:object_type(?z, ?o)
→
rbac:DRA_FULL(?z)

Text 13: SWRL for rule 2_dra_full.

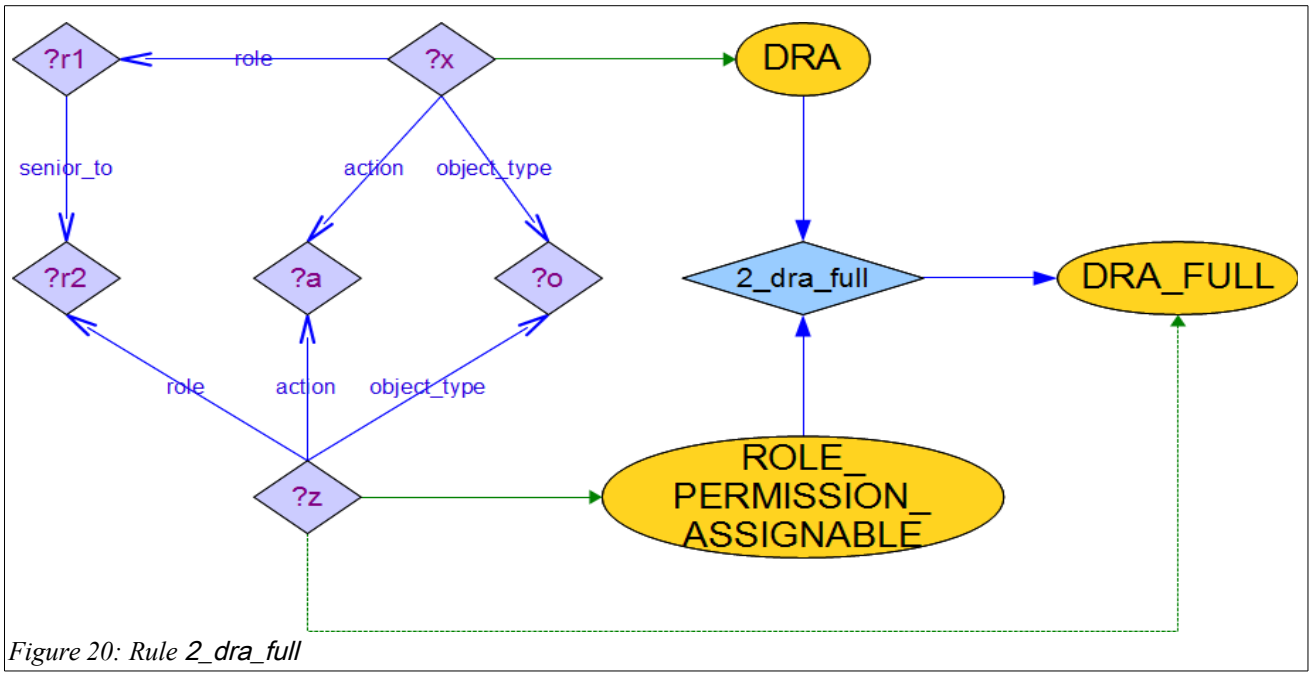

Figure 20: Rule 2_dra_full

The first rule in Step 2, given in Figure 20, is called 2_dra_full. This rule moves an individual from ROLE_PERMISSION_ASSIGNABLE to DRA_FULL if there exists an individual in DRA that has the same *action* and *object_type* properties as that in ROLE_PERMISSION_ASSIGNABLE, and if the role property of the individual in DRA is senior to that of the individual in ROLE_PERMISSION_ASSIGNABLE.

A formal description of the matching in rule 2_dra_full is below. ROLE_PERMISSION_ASSIGNABLE instance *?z* represents a potential user-role assignment with the following properties:

- *rbac:action ?a*;
- *rbac:role ?r2*, and
- *rbac:object_type ?o*.

?z is moved to DRA_FULL if:

i) *?z* is linked by object property *rbac:role* to *?r2*;

ii) *?r1* is senior to *?r2* (is linked to *?r1* via object property *rbac:senior_to*);

iii)DRA instance *?x* is linked by object property *rbac:role* to *?r1*, and

iv)both *?z* and *?x* have *rbac:action ?a* and *rbac:object_type ?o*.

Figure 20 is converted into SWRL syntax in Text 13 above.

rbac:PRA(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧
rbac:object_type(?x, ?o) ∧ rbac:senior_to(?r2, ?r1) ∧
rbac:ROLE_PERMISSION_ASSIGNABLE(?z) ∧
rbac:role(?z, ?r2) ∧ rbac:action(?z, ?a) ∧
rbac:object_type(?z, ?o) ∧ rbac:inherits_pra(?r2, ?r1) →
rbac:PRA_FULL(?z)

Text 14: SWRL for rule 2_pra_full

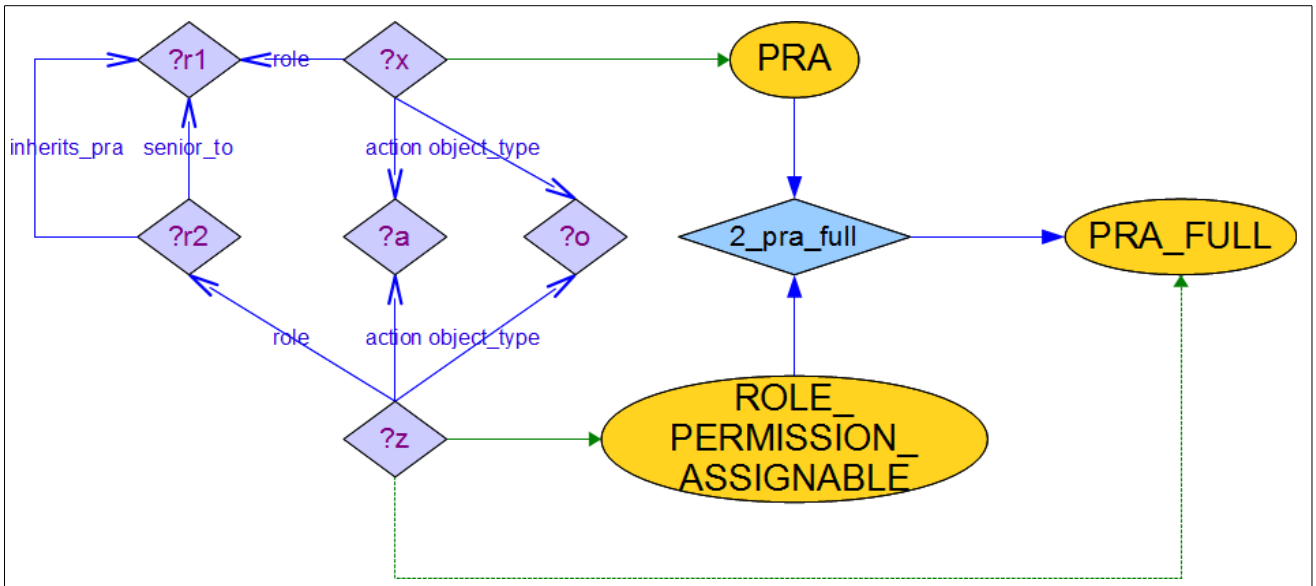

Figure 21: Rule 2_pra_full

The second rule in Step 2, given in Figure 21, is called 2_pra_full. This rule moves an individual from ROLE_PERMISSION_ASSIGNABLE to PRA_FULL if there exists an individual in PRA that has the same action and object_type properties as that in ROLE_PERMISSION_ASSIGNABLE, and if the role property of the individual in PRA is junior to that of the individual in ROLE_PERMISSION_ASSIGNABLE.

A formal description of the matching in rule in 2_dra_full is below. ROLE_PERMISSION_ASSIGNABLE instance ?z represents a potential user-role assignment with the following properties:

- *rbac:action ?a*;
- *rbac:role ?r2*, and
- *rbac:object_type ?o*.

?z is moved to PRA_FULL if:

i) *?z* is linked by object property *rbac:role* to *?r2*;

ii) *?r2* is senior to *?r1* (is linked to *?r1* via object property *rbac:senior_to*);

iii)*?r2* and *?r1* are in an inheritance path (linked via object property *rbac:inherits_pra*);

iv)PRA instance *?x* is linked by object property *rbac:role* to *?r1*, and

v) both *?z* and *?x* have *rbac:action ?a* and *rbac:object_type ?o*.

Figure 21 is converted into SWRL syntax in Text 14 above.

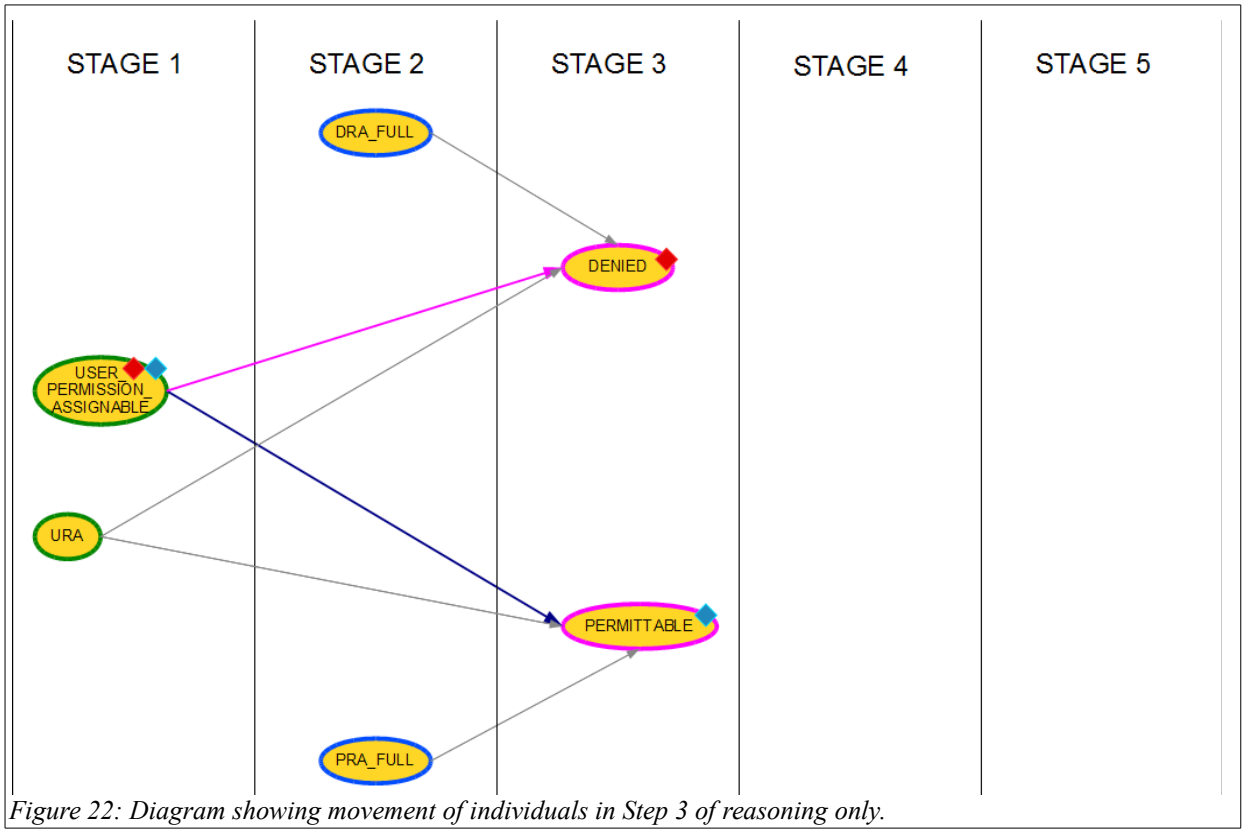

Figure 22: Diagram showing movement of individuals in Step 3 of reasoning only.

Figure 22 shows the movement of individuals in Step 3, in which PERMITTABLE and DENIED are populated from individuals in USER_PERMISSION_ASSIGNABLE, as determined by individuals in URA, PRA_FULL and DRA_FULL, as well as relationships between roles defined by *included_in* axioms.

rbac:PRA_FULL(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:included_in(?r2, ?r1) ∧ rbac:instance_of(?oi, ?o) ∧ rbac:USER_PERMISSION_ASSIGNABLE(?z) ∧ rbac:action(?z, ?a) ∧ rbac:object_instance(?z, ?oi) ∧ rbac:user(?z, ?u) ∧ rbac:URA(?y) ∧ rbac:role(?y, ?r2) ∧ rbac:user(?y, ?u) → rbac:PERMITTABLE(?z)

Text 15: SWRL for rule 3_permittable

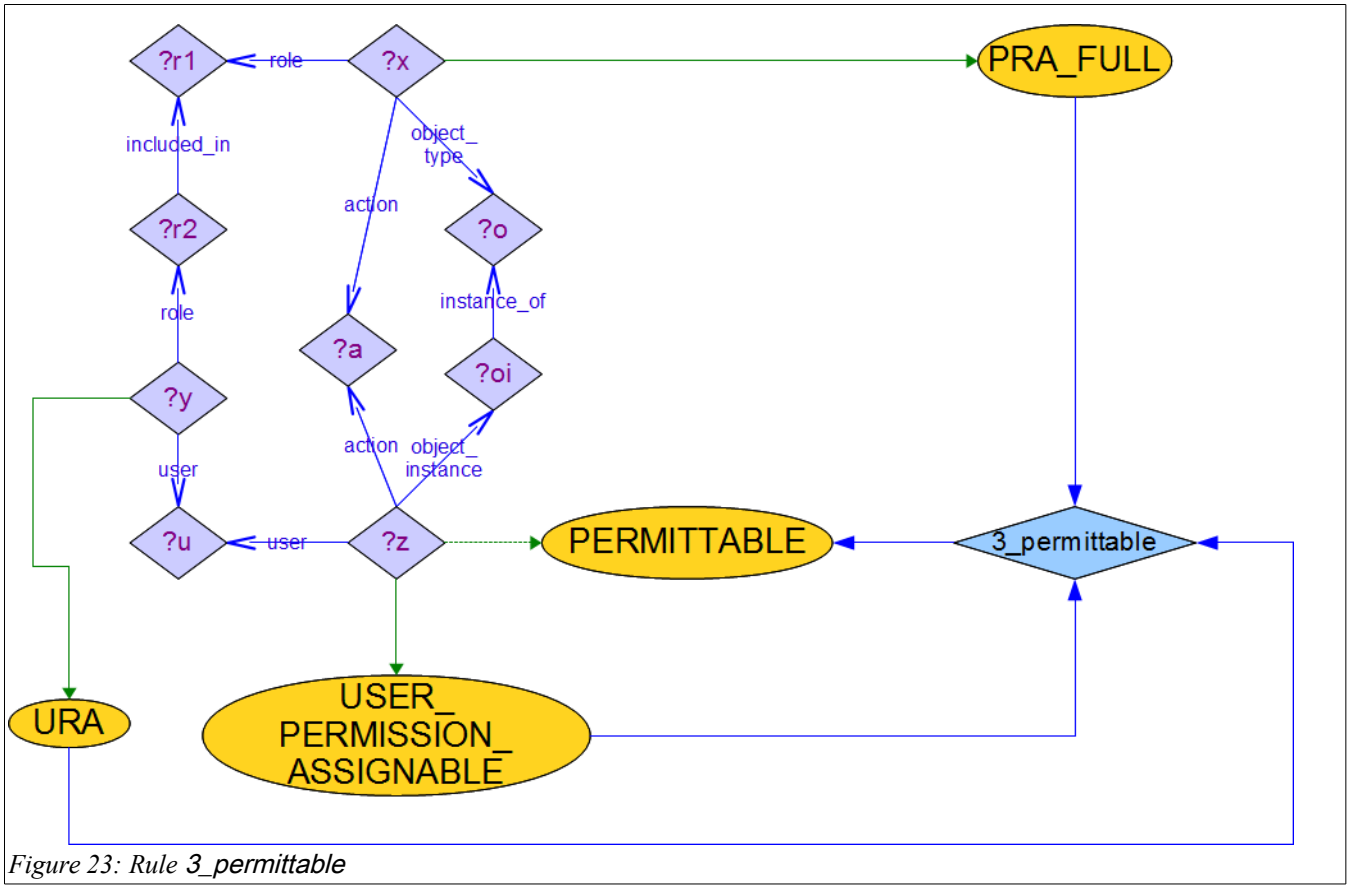

Figure 23: Rule 3_permittable

The first rule in Step 3, given in Figure 23, is called 3_permittable. This rule moves an individual from USER_PERMISSION_ASSIGNABLE to PERMITTABLE if that individual is found to represent an actual user-permission assignment in the RBAC model. That is, if an individual in USER_PERMISSION_ASSIGNABLE has the same action as an individual in PRA_FULL; has object instance that is linked to an object type in this USER_PERMISSION_ASSIGNABLE individual, and has a user that is assigned to a role in this USER_PERMISSION_ASSIGNABLE individual, or a role that is included in this role, then it is moved to PERMITTABLE.

A formal description of the matching in rule in 3_permittable is below. USER_PERMISSION_ASSIGNABLE instance *?z* represents a potential user-permission assignment. It has the following properties:

- *rbac:action* linked to *?a*, representing an action performed by a user;
- *rbac:user ?u*, and

- *rbac:object_instance ?oi*, representing a specific data object that may be accessed by user *?u*.

?x is an instance in PRA_FULL with the following properties:

- *rbac:action* linked to *?a*;
- *rbac:role ?r1*, and
- *rbac:object_type ?o*, representing a type of object that may be accessed by users in role *?r1*.

?z is moved to PERMITTABLE if it is found to be an actual user-permission assignment in the RBAC model, according to the following rules:

i) *?z* has user *?u*;

ii) *?u* is assigned to role *?r2* by URA instance *?y*;

iii) *?r2* is included in role *?r1* (*?r2* is linked to *?r1* via property *rbac:included_in*);

iv) PRA_FULL instance *?x* has role *?r1*;

v) Both *?z* and *?x* have *rbac:action ?a*;

vi) *?z* has *rbac:object_instance ?oi*;

vii) *?oi* is a data object of type *?o* (*?oi* is linked to *?o* via *rbac:instance_of* property), and

viii) *?x* has object_type *?o*.

Figure 23 is converted into SWRL syntax in Text 15 above.

rbac:DRA_FULL(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:included_in(?r2, ?r1) ∧ rbac:instance_of(?oi, ?o) ∧ rbac:USER_PERMISSION_ASSIGNABLE(?z) ∧ rbac:action(?z, ?a) ∧ rbac:object_instance(?z, ?oi) ∧ rbac:user(?z, ?u) ∧ rbac:URA(?y) ∧ rbac:role(?y, ?r2) ∧ rbac:user(?y, ?u) → rbac:DENIED(?z)

Text 16: SWRL for rule 3_denied

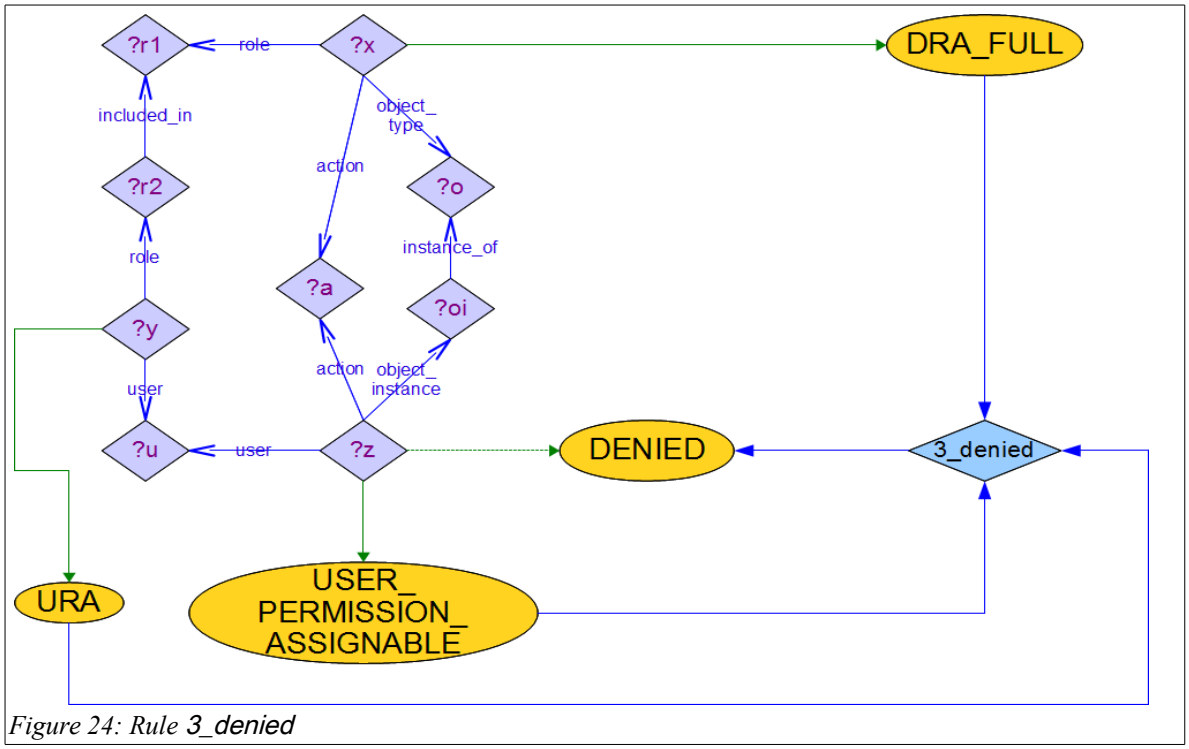

Figure 24: Rule 3_denied

The second rule in Step 3, given in Figure 24, is called 3_denied. This rule moves an individual from USER_PERMISSION_ASSIGNABLE is moved to DENIED if it is found to represent an actual user-denial assignment in the RBAC model. That is, if an individual in USER_PERMISSION_ASSIGNABLE has the same action as an individual in DRA_FULL; has object instance that is linked to an object type in this USER_PERMISSION_ASSIGNABLE individual, and has a user that is assigned to a role in this USER_PERMISSION_ASSIGNABLE individual, or a role that is included in this role, then it is moved to DENIED.

A formal description of the matching in rule in 3_denied is below.

USER_PERMISSION_ASSIGNABLE instance *?z* is moved to DENIED if:

i) *?z* has *rbac:user ?u*;

ii) *?u* is assigned to *rbac:role ?r2* by URA instance *?y*;

iii)*?r2* is included in *?r1*;

iv)DRA_FULL instance *?x* has *rbac:role ?r1*;

v) Both *?z* and *?x* have *rbac:action ?a*;

vi)*?z* has *rbac:object_instance ?oi*;

vii)*?oi* is a data object of type *?o* (*?oi* is linked to *?o* via property *rbac:instance_of*), and *?x* has *rbac:object_type ?o*.

Figure 24 is converted into SWRL syntax in Text 16 above.

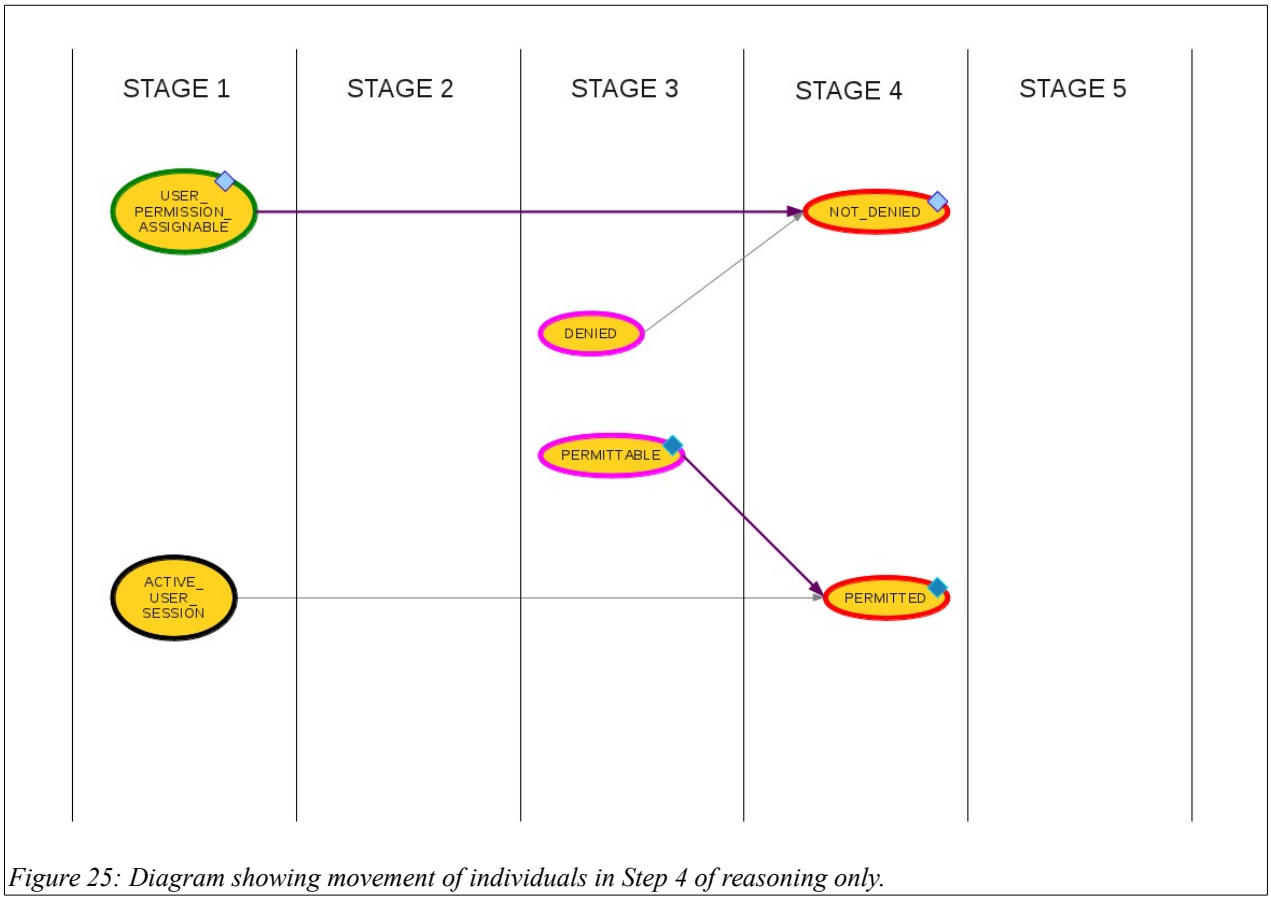

Figure 25: Diagram showing movement of individuals in Step 4 of reasoning only.

Figure 25 shows the movement of individuals in Step 4, in which PERMITTED is populated from PERMITTABLE and ACTIVE_USER_SESSION, and NOT_DENIED from USER_PERMISSION_ASSIGNABLE and DENIED.

rbac:USER_PERMISSION_ASSIGNABLE(?x) ∧ rbac:DENIED(?y) ˚ sqwrl:makeSet(?d, ?y) ˚ sqwrl:notElement(?x, ?d) → rbac:NOT_DENIED(?x)

Text 17: SWRL for rule 4_not_denied

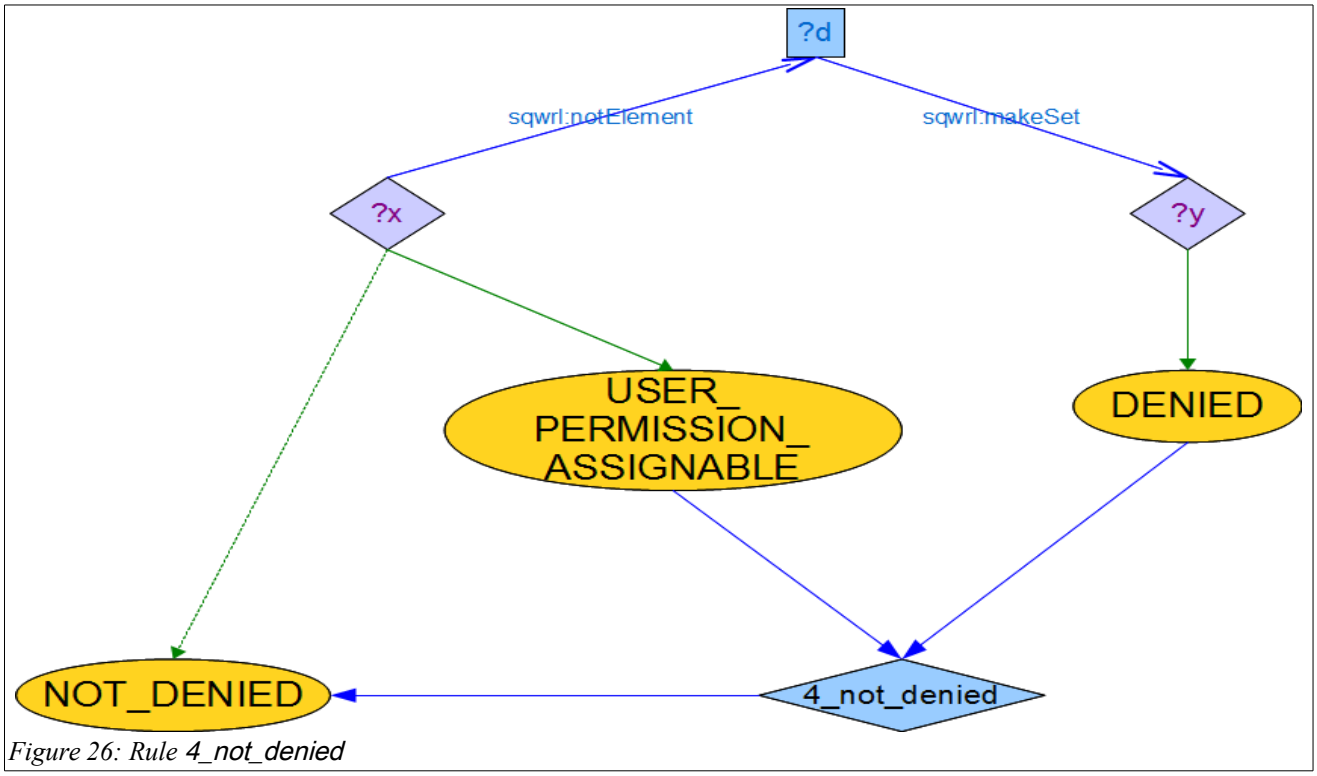

Figure 26: Rule 4_not_denied

The first rule in Step 4, given in Figure 26, is called 4_not_denied. This rule populates NOT_DENIED as all individuals in USER_PERMISSION_ASSIGNABLE that are not in DENIED. Mathematically, NOT_DENIED is defined as the set difference of USER_PERMISSION_ASSIGNABLE and DENIED (Formula 2).

NOT_DENIED = USER_PERMISSION_ASSIGNABLE – DENIED

Formula 2: Definition of NOT_DENIED.

A formal description of the matching in rule in 4_not_denied is below. USER_PERMISSION_ASSIGNABLE instance *?x* is moved to NOT_DENIED if *?x* is not in DENIED. This is determined as follows:

i) *?d* is a set of all instances *?y* in DENIED, and

ii) *?x* is not in *?d*.

Mathematically, this can be represented as in Formula 3.

$$\forall y, \forall x, y \in \text{DENIED}, x \in \text{USER\_PERMISSION\_ASSIGNABLE}, x \neq y \Rightarrow x \in \text{NOT\_DENIED}$$

Formula 3: Matching NOT_DENIED.

Or, more simply, as in Formula 4.

$$\forall x, x \notin \text{DENIED}, x \in \text{USER\_PERMISSION\_ASSIGNABLE} \Rightarrow x \in \text{NOT\_DENIED}$$

Formula 4: Simplified matching NOT_DENIED.

The implementation of this negation formula is quite complex in SWRL, due to the way that OWL handles negation, which is different from that of languages such as Prolog which are based on predicate logic. Prolog uses classical negation, also called "negation as failure". [30] This means that if the truth of a query cannot be inferred, then the query is assumed to be false (if a query fails, then its negation succeeds). This is also called the 'closed world assumption'. In contrast, OWL implements an 'open world' assumption: a fact that cannot be inferred is not necessarily false. Since SWRL is a tool for querying ontologies, it does not have a negation operation. However, it is possible to simulate classical negation in SWRL using functions in SQWRL, which is discussed at the start of Section 5.2.3. The SQWRL function makeSet makes a set consisting of a list of previously defined individuals. The SQWRL functions element and notElement check whether a given individual is a member of a set. notElement enables negation-as-failure to be used with OWL and SWRL.

Figure 26 is converted into SWRL syntax in Text 17 above.

```
rbac:PERMITTABLE(?x) ∧ rbac:user(?x, ?u) ∧
rbac:ACTIVE_USER_SESSION(?s) ∧  rbac:user(?
s, ?u)
→  rbac:PERMITTED(?x)
```

Text 18: SWRL for rule 4_permitted

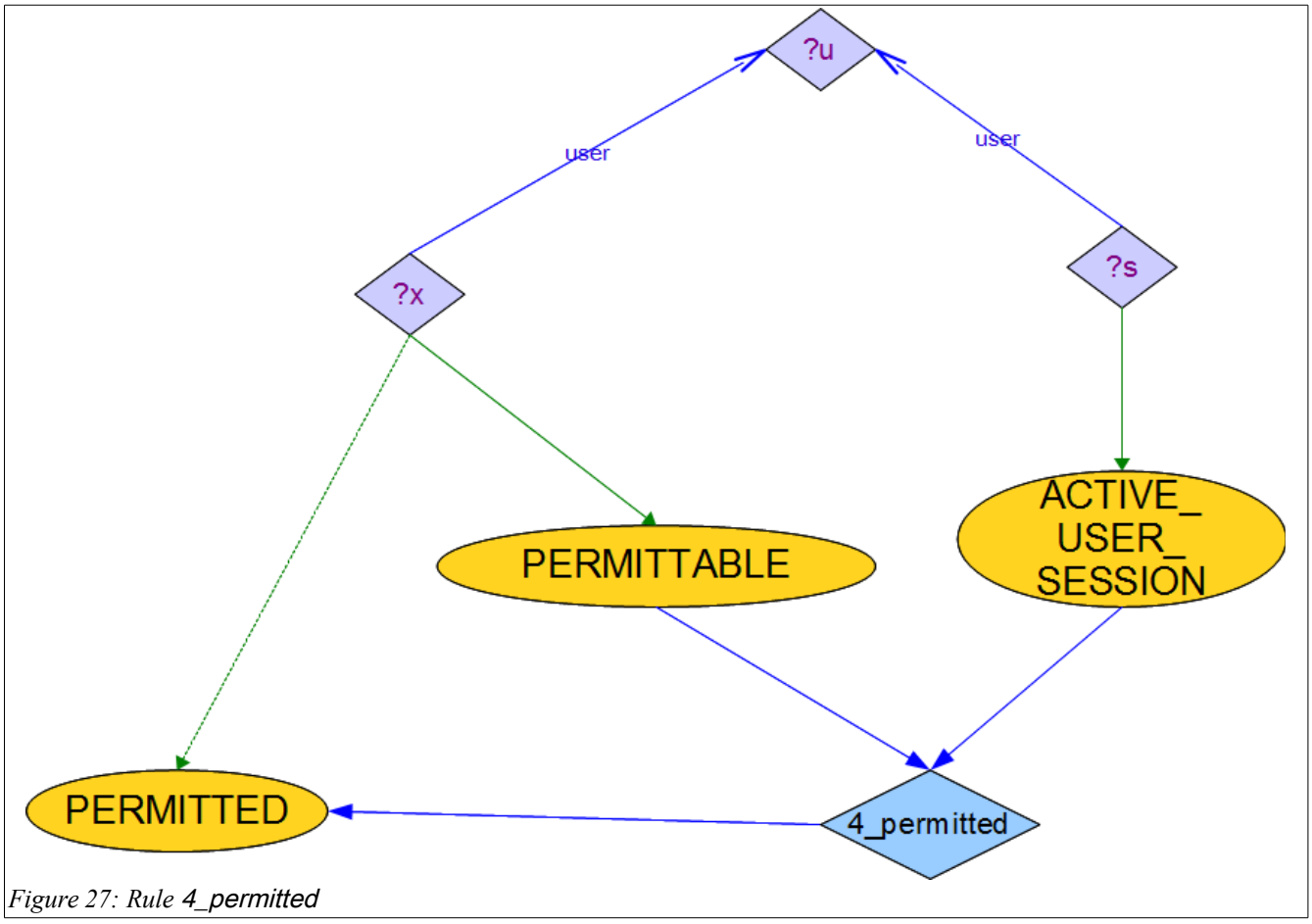

Figure 27: Rule 4_permitted

The second rule in Step 4, given in Figure 27, is called 4_permitted. This rule moves an individual in PERMITTABLE to PERMITTED if the individual has a user that is also a user in an active user session, as given by an individual in the class ACTIVE_USER_SESSION. The difference between PERMITTABLE and PERMITTED is that PERMITTABLE represents potential permissions, while PERMITTED represents actual permissions, as determined by active user sessions.

A formal description of the matching in rule in 4_permitted is below. PERMITTABLE instance *?x* is moved to PERMITTED if:

i) *?x* has *rbac:user ?u*, and

ii) ACTIVE_USER_SESSION user *?s* has *rbac:user ?u*.

Figure 27 is converted into SWRL syntax in Text 18 above.

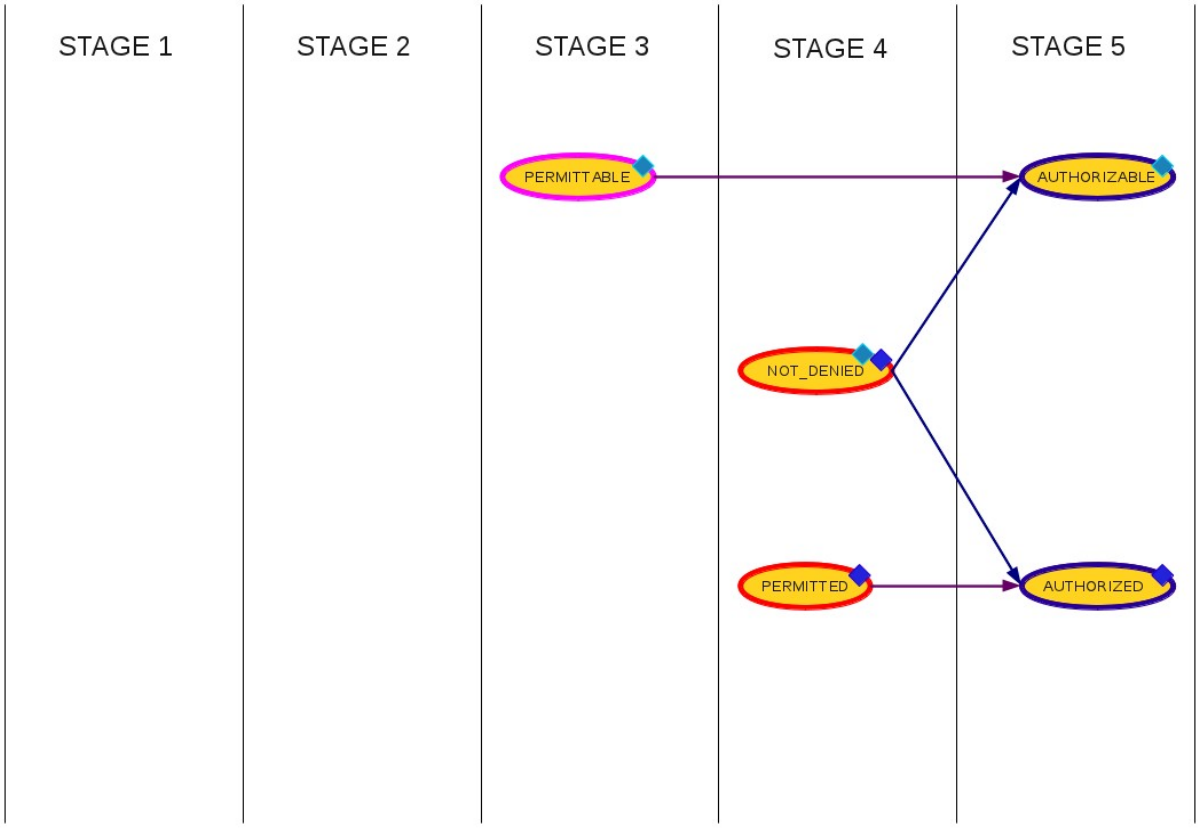

Figure 28: Diagram showing movement of individuals in Step 5 of reasoning only.

Figure 28 shows the movement of individuals in Step 5, in which AUTHORIZED is populated from PERMITTED and NOT_DENIED. AUTHORIZABLE is populated from PERMITTABLE and NOT_DENIED.

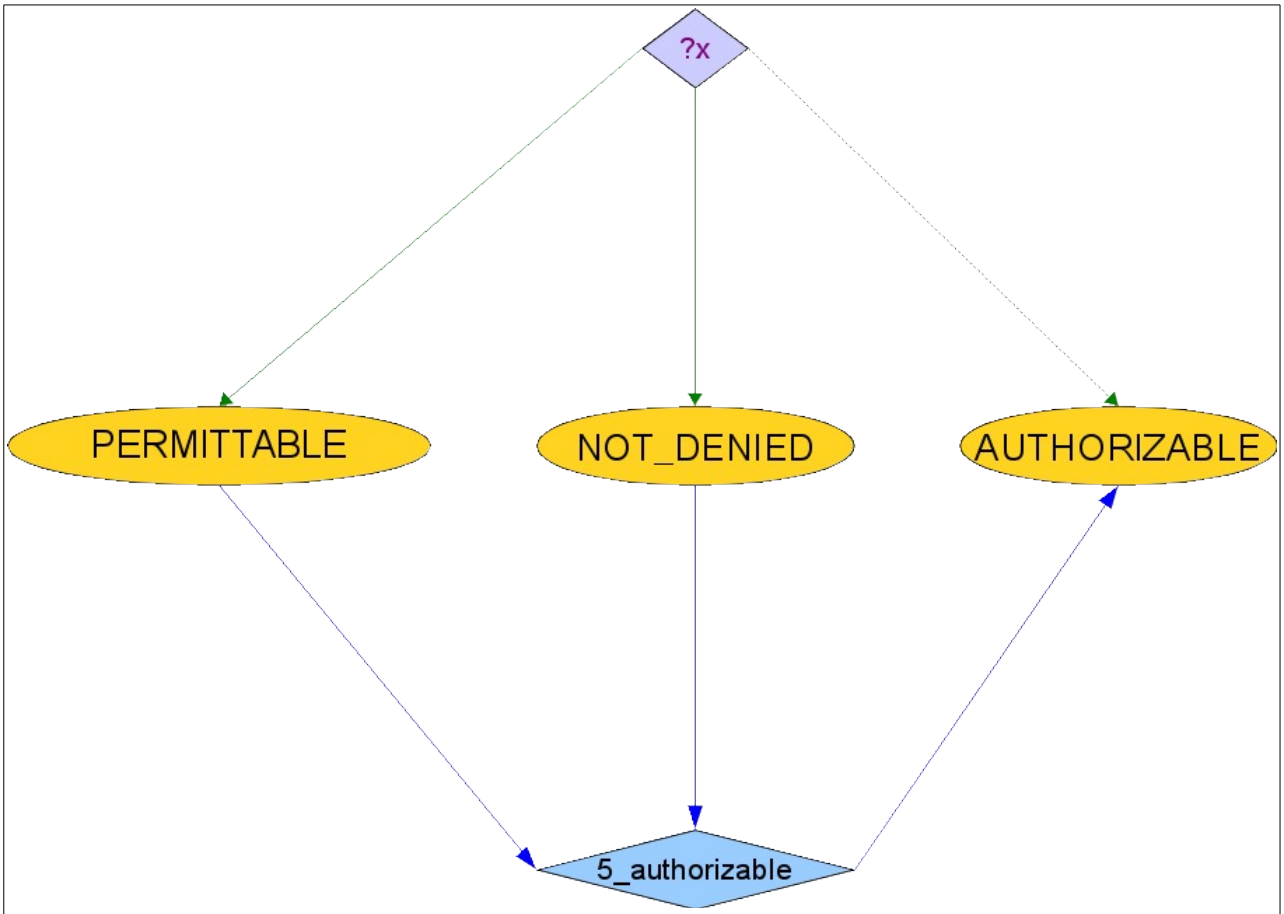

Figure 29: Rule 5_authorizable

rbac:PERMITTABLE(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZABLE(?x)

Text 19: SWRL for rule 5_authorizable

The first rule in Step 5, given in Figure 29, is called 5_authorizable. AUTHORIZABLE is defined as the intersection of PERMITTABLE and NOT_DENIED: an individual is in AUTHORIZABLE if it is in both PERMITTABLE and NOT_DENIED (Formula 5).

AUTHORIZABLE = PERMITTABLE ∩ NOT_DENIED

Formula 5: Definition of AUTHORIZABLE.

PERMITTABLE instance *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED. Note that this means that the same actual instance *?x* has to be in both PERMITTABLE and NOT_DENIED (not different instances with the same object properties).

Figure 29 is converted into SWRL syntax in Text 19 above.

rbac:PERMITTED(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZED(?x)

Text 20: SWRL for rule 5_authorized

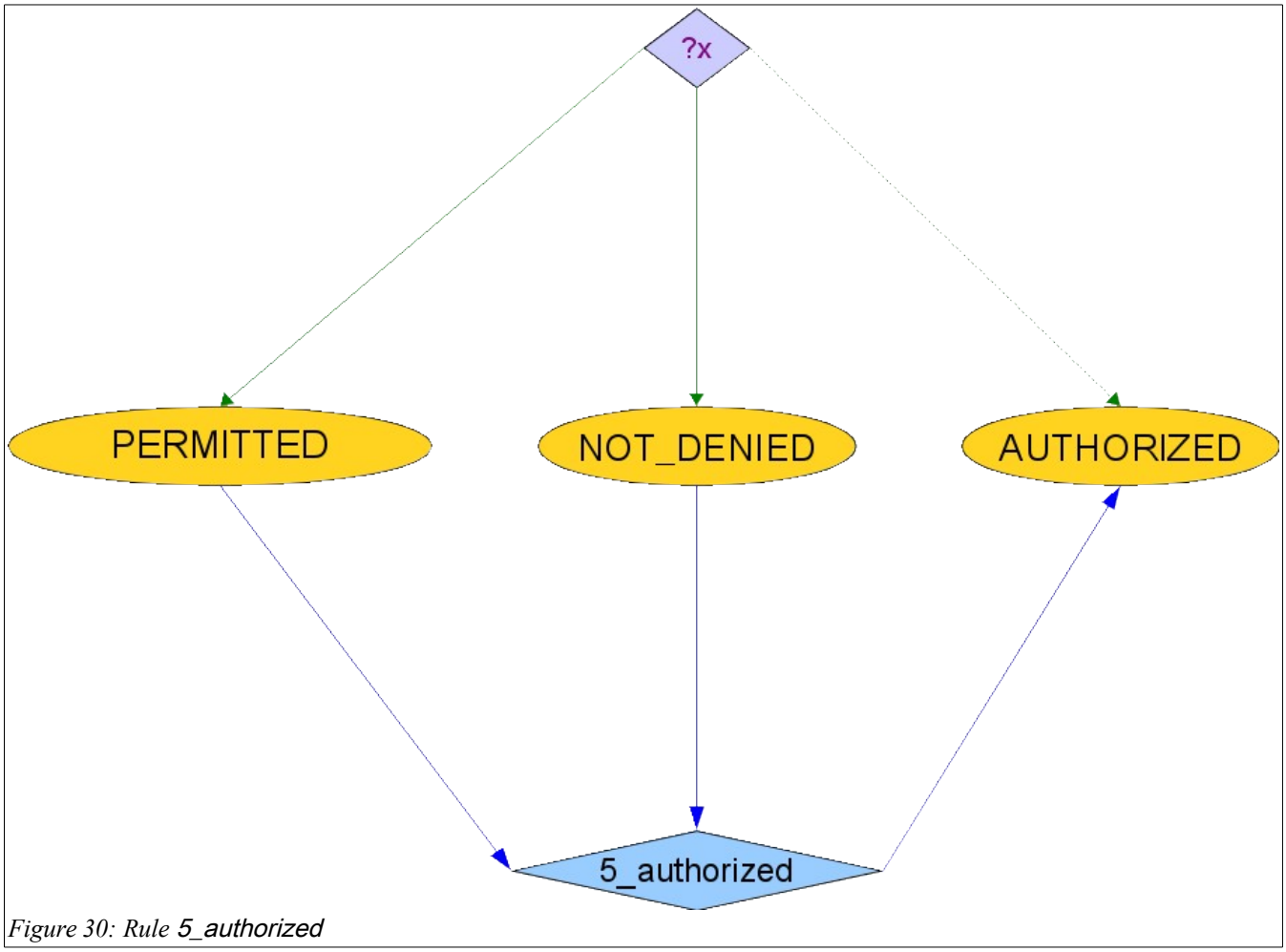

Figure 30: Rule 5_authorized

The second rule in Step 5, given in Figure 30, is called 5_authorized. This defines AUTHORIZED as the intersection of PERMITTED and NOT_DENIED: an individual is in AUTHORIZED if it is in both PERMITTED and NOT_DENIED (Formula 6).

AUTHORIZED = PERMITTED ∩ NOT_DENIED

Formula 6: Definition of AUTHORIZED.

PERMITTED instance *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED.

Figure 30 is converted into SWRL syntax in Text 20 above.

5.3 SO-RBAC Process

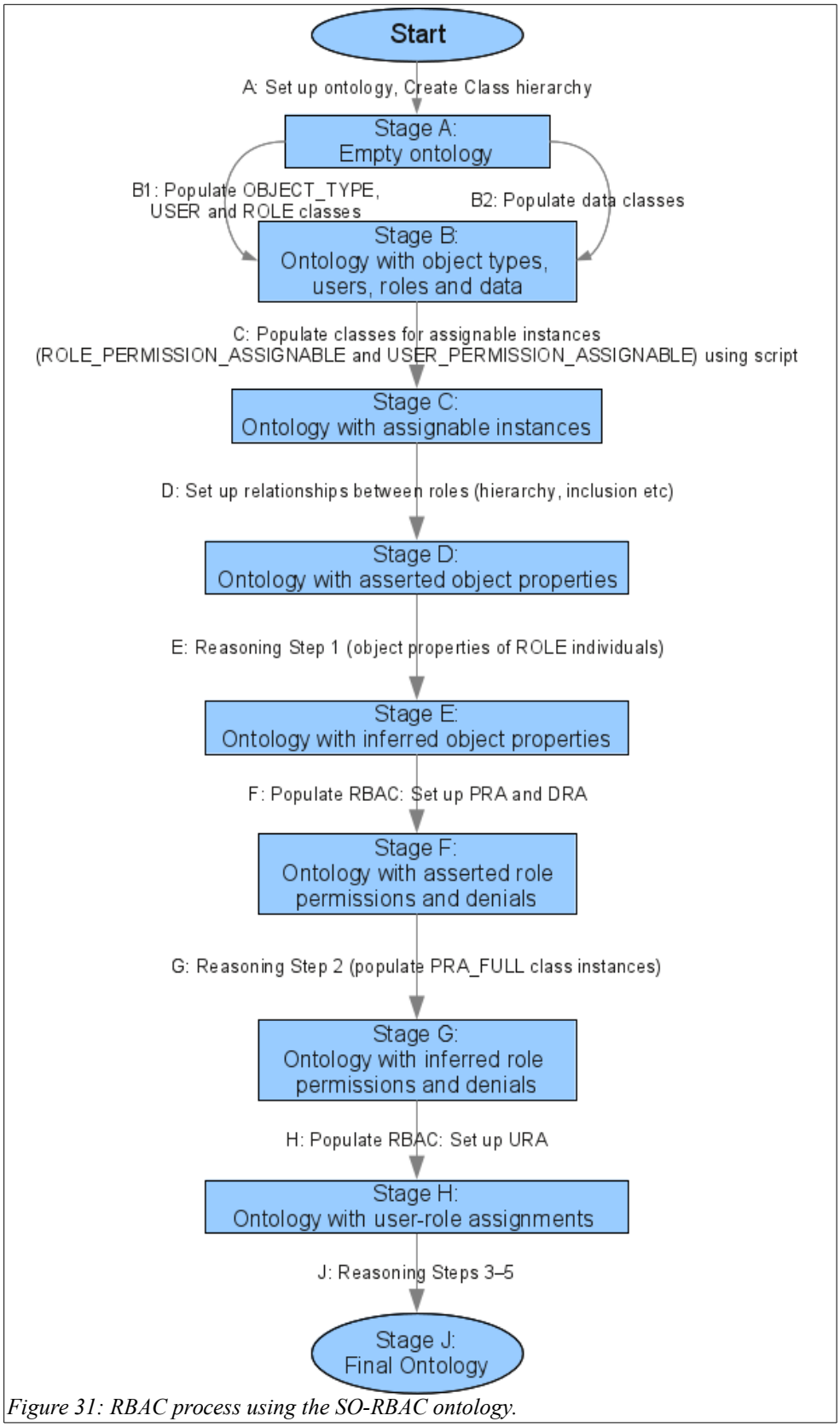

Figure 31: RBAC process using the SO-RBAC ontology.

```
domain := "rbac";
domain_uri := "http://www.cgce.net/Ontology/RBAC";

class: = "ROLE_PERMISSION_ASSIGNABLE";
for each role
 for each action
  for each object_type
   id := "role_action_object_type"
   print "  <domain:class rdf:ID=\"id\">";
   print "    <domain:action rdf:resource=\"#action\"/>";
   print "    <domain:role rdf:resource=\"#r_role\"/>";
   print "    <domain:object_type rdf:resource=\"#o_$object_type\"/>";
   print "  </$domain:$class>";
  next
 next
next

class := "USER_PERMISSION_ASSIGNABLE";
for each action
 for each object_instance
  for each user
   id = user_action_object_instance";
   print "  <domain:class rdf:ID=\"id\">";
   print "    <domain:action rdf:resource=\"#action\"/>";
   print "    <domain:object_instance rdf:resource=\"#object_instance\"/>";
   print "    <domain:user rdf:resource=\"#user\"/>";
   print "  </domain:class>";
  next
 next
next
```

Text 21: Pseudocode for step C

```
rbac:ROLE(?r) ∧ rbac:directly_senior_to(?_, ?r) → rbac:senior_to(?r, ?r)

rbac:ROLE(?r) ∧  rbac:directly_senior_to(?r, ?_) → rbac:senior_to(?r, ?r)

rbac:directly_senior_to(?r1, ?r2) ∧ rbac:senior_to(?r2, ?r3) → rbac:senior_to(?r1, ?r3)

rbac:ROLE(?r) → rbac:included_in(?r, ?r)

rbac:is_a(?r1, ?r2) ∧ rbac:included_in(?r2, ?r3) → rbac:included_in(?r1, ?r3)

rbac:ROLE(?r) → rbac:inherits_pra(?r, ?r)

rbac:senior_to(?r1, ?r2) ∧ rbac:senior_to(?r3, ?r4) ∧ rbac:senior_to(?r3, ?r4)  ∧ rbac:inherits_pra_path(?r1, ?r4) → rbac:inherits_pra(?r2, ?r3)
```

Code 54: SWRL Rules for Step E.

rbac:DRA(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:senior_to(?r1, ?r2) ∧ rbac:ROLE_PERMISSION_ASSIGNABLE(?z) ∧ rbac:role(?z, ?r2) ∧ rbac:action(?z, ?a) ∧ rbac:object_type(?z, ?o) → rbac:DRA_FULL(?z)

rbac:PRA(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:senior_to(?r2, ?r1) ∧ rbac:ROLE_PERMISSION_ASSIGNABLE(?z) ∧ rbac:role(?z, ?r2) ∧ rbac:action(?z, ?a) ∧ rbac:object_type(?z, ?o) ∧ rbac:inherits_pra(?r2, ?r1) → rbac:PRA_FULL(?z)

Code 55: SWRL Rules for Step G.

Figure 31 (page 89) shows a flowchart of the process for setting up a SO-RBAC and populating it with all the user permissions and denials on objects on that apply at a point in time.

Each potential *role* permission or denial to perform an action on an object is represented by an individual in the SO-RBAC class ROLE_PERMISSION_ASSIGNABLE. The process moves individuals representing role permissions and denials to the class PRA_FULL (indicating a permission) or DRA_FULL (indicating a denial).

Each potential *user* permission or denial to perform an action on an object is represented by an individual in the SO-RBAC class USER_PERMISSION_ASSIGNABLE. The process moves individuals representing role permissions and denials to the classes PERMITTABLE or PERMITTED (indicating a permission) or DENIED (indicating a denial). Finally, an individual representing a user permission that is not also a denial is moved to AUTHORIZABLE or AUTHORIZED. All this is done according to rules in the SO-RBAC process.

The first step, step A, is to set up the class hierarchy in the SO-RBAC ontology. This includes setting up the data classes under OBJECT_INSTANCE, to which the data that the SO-RBAC model governs access, and the classes relevant to the SO-RBAC model itself, under RBAC class. OBJECT_INSTANCE hierarchies are always domains specific, but RBAC sub-hierarchies are likely to remain the same across domains.

Step B populates the ontology with the base information. It has two parts, B1 and B2, which can be run in parallel. This is because they are independent of each other.

In step B1, the individuals representing types of data (class OBJECT_TYPE), users (class USER) and roles (sub-classes of class ROLE) are initialised by populating the classes, USER and ROLE (and their sub-classes as appropriate).

In B2, the data classes (sub-classes of OBJECT_INSTANCE) are populated with individuals representing data for which access is to be granted by the SO-RBAC model.

In step C, the classes ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE are populated, with all possible combinations of hypothetical role and user permission assignments. Due to the exponentially increasing number of combinations with increasing size of ontology, this is most likely to be done using a program or script, according to the pseudocode in Text 21.

In step D, the asserted relationships between ROLE individuals (*directly_senior_to*, *is_a*, *inherits_pra_path*) are set up, to define the relationships between roles in the RBAC role hierarchy.

In step E, the SWRL rules are run to infer the object properties that depend on the properties asserted in step D, namely *senior_to*, *included_in* and *inherits_pra*, which are respectively dependent on *directly_senior_to*, *is_a* and *inherits_pra_path*. The seven SWRL rules are described in Section 5.2.3.1, and are summarised in Code 54.

In step F, the PRA and DRA classes are populated to set up role permissions and denials, because the individuals in these classes are base information for reasoning in the RBAC model. This can be done after step E because the information about role permissions and denials is not needed for inferring relationships between roles.

In Step G, we populate the PRA_FULL and DRA_FULL classes with individuals through inference by running the following two SWRL rules in Code 55 (*cf.* Section 5.2.3.2).

In step H, the user-role relationships are set up, i.e. URA is populated with individuals. Again, this is essential information needed for reasoning in the RBAC model, but it is not needed for inferring either relationships between roles or assignment of permissions or denials to roles.

rbac:PRA_FULL(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:included_in(?r2, ?r1) ∧ rbac:instance_of(?oi, ?o) ∧ rbac:USER_PERMISSION_ASSIGNABLE(?z) ∧ rbac:action(?z, ?a) V rbac:object_instance(?z, ?oi) ∧ rbac:user(?z, ?u) ∧ rbac:URA(?y) ∧ rbac:role(?y, ?r2) ∧ rbac:user(?y, ?u)
→ rbac:PERMITTABLE(?z)

rbac:DRA_FULL(?x) ∧ rbac:role(?x, ?r1) ∧ rbac:action(?x, ?a) ∧ rbac:object_type(?x, ?o) ∧ rbac:included_in(?r2, ?r1) ∧ rbac:instance_of(?oi, ?o) ∧ rbac:USER_PERMISSION_ASSIGNABLE(?z) ∧ rbac:action(?z, ?a) ∧ rbac:object_instance(?z, ?oi) ∧ rbac:user(?z, ?u) ∧ rbac:URA(?y) ∧ rbac:role(?y, ?r2) ∧ rbac:user(?y, ?u) → rbac:DENIED(?z)

rbac:USER_PERMISSION_ASSIGNABLE(?x) ∧rbac:DENIED(?y) ° sqwrl:makeSet(?d, ?y) ° sqwrl:notElement(?x, ?d) → rbac:NOT_DENIED(?x)

rbac:PERMITTABLE(?x) ∧ rbac:user(?x, ?u) ∧ rbac:ACTIVE_USER_SESSION(?s) ∧ rbac:user(?s, ?u) → rbac:PERMITTED(?x)

rbac:PERMITTABLE(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZABLE(?x)

rbac:PERMITTED(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZED(?x)

Code 56: SWRL Rules for Step J.

Finally, in step J, the remaining reasoning steps (3–5) are performed. We run 6 SWRL rules (Code 56) which ultimately populated DENIED or AUTHORIZED classes (*cf.* Section 5.2.3.2).

At each stage the SO-RBAC ontology is in a state where the process can be run from the following step onwards. In other words, it is not necessary to always re-run the SO-RBAC process from the beginning.

5.4 Contrasting SO-RBAC with Prolog

Most SWRL rules in SO-RBAC are directly translated from the Prolog rules in Code 53 (page 60). However, there are two ways in which some SWRL definitions differ from those in Prolog.

5.4.1 Property inheritance

As noted above, some properties are super-properties of others, and so inherit the relationships defined with them. This reduces the number of rules that need to be defined in SWRL.

```
1 senior_to(R,R) :- directly_senior_to(R,_).
2 senior_to(R,R) :- directly_senior_to(_,R).
3 senior_to(R1,R2) :- directly_senior_to(R1,R2).
4 senior_to(R1,R3) :- directly_senior_to(R1,R2), senior_to(R2,R3).
```

Code 57: `senior_to` in Prolog.

For example, `senior_to` is defined in Prolog using the following 4 rules (Code 57).

Rules 1 and 2 define a role as being senior to itself, but only if it participates in a `directly_senior_to` relationship. Rule 3 states that role R1 is senior to R2 if it is `directly_senior_to` R2. Rule 4 creates the recursion.

In SO-RBAC, Rule 3 is achieved by making ***directly_senior_to*** a sub-property of ***senior_to***.

$$\forall\, a, b;\ a\ \mathbb{Q}\ b;\ \mathbb{Q} \text{ is sub-property of } \mathbb{P} \Rightarrow a\ \mathbb{P}\ b$$

Formula 7: Inferences from sub-properties.

In OWL, if property $\mathbb{P}$ is a sub-property of property $\mathbb{Q}$, then all axioms asserted for $\mathbb{Q}$ are inferred for $\mathbb{P}$. This means that, given individuals *a* and *b*, and properties $\mathbb{P}$ and $\mathbb{Q}$, Formula 7 applies.

```
rbac:ROLE(?r) ∧ rbac:directly_senior_to(?_, ?r) → rbac:senior_to(?r, ?r)
rbac:ROLE(?r) ∧ rbac:directly_senior_to(?r, ?_) → rbac:senior_to(?
rbac:directly_senior_to(?r1, ?r2) ∧ rbac:senior_to(?r2, ?r3) →
rbac:senior_to(?r1, ?r3)
```

*Text 22: **senior_to** in SWRL.*

Therefore, any ***directly_senior_to*** relationship between any two individuals infers a ***senior_to*** relationship between the same pair of individuals. This eliminates the need for a SWRL for Rule 3. Rules 1, 2 and 4 are defined as in Text 22.

Note that in Prolog, the consequent appears at the start of a clause, while in SWRL, it appears at the end.

Similarly, making ***is_a*** a sub-property of ***included_in*** implements the 2nd `included_in` definition from Prolog.

In Step 1, object properties *senior_to*, *included_in* and *inherits_pra* are inferred from other object properties and membership of the class ROLE.

i) *senior_to* is inferred from membership of ROLE and from *directly_senior_to*. (Rules: 1_senior_to_1, 1_senior_to_2 and 1_senior_to_4)

ii) *included_in* is inferred from membership of ROLE and *is_a*. (Rules: 1_included_in _1, 1_included_in_3)

iii)*inherits_pra* is inferred from membership of ROLE, *inherits_pra_path* and *senior_to*. (Rules: 1_inherits_pra_1, 1_inherits_pra_3)

1_senior_to_3, 1_included_in_2 and 1_inherits_pra_2 do not exist.

5.4.2 Negation and Transitivity

```
authorized(U,A,O) :- permitted(U,A,O),
                     not(denied(U,A,O)).
```

Code 58: `authorized` in Prolog.

`authorized` (`authorizable`) are defined as being `permitted` (`permittable`) and `not denied`. This is simple to express, and quick to run, in Prolog (Code 58).

However, the implementation in SO-RBAC is more complex, because OWL handles negation differently from Prolog. Prolog uses classical negation, also called "negation as failure". This means that if the truth of a query cannot be inferred, then the query is assumed to be false (if a query fails, then its negation succeeds). This is also called the 'closed wo .lnbrld assumption'. In contrast, OWL implements an 'open world' assumption: a fact that cannot be inferred is not necessarily false. Since SWRL is a tool for querying ontologies, it does not have a negation operation. However, it is possible to simulate classical negation in SWRL using SQWRL operators.

rbac:USER_PERMISSION_ASSIGNABLE(?x) ∧ rbac:DENIED(?y) ˚ sqwrl:makeSet(?d, ?y) ˚ sqwrl:notElement(?x, ?d) → rbac:NOT_DENIED(?x)

Text 23: NOT_DENIED in SWRL.

As noted earlier, SO-RBAC defines a class NOT_DENIED, for USER_PERMISSION_ASSIGNABLE individuals that do not appear in DENIED. To populate this class, the SQWRL operators makeSet and notElement are used. makeSet makes a set consisting of a list of previously defined individuals. element and notElement check whether a given individual is a member of a set. notElement enables negation-as-failure to be used with OWL and SWRL. Text 23 defines NOT_DENIED.

[In the SWRL syntax used in Protégé, both ∧ and ˚ mean logical AND.] For this to work, there has to be at least one individual in DENIED. Therefore, DENIED must be initialized with a dummy individual, with none of its properties defined.

NOT_DENIED is defined as a class because the above rule takes a long time to run, and its result set is used more than once. It is much quicker to run this rule once and store its results, than to run it each time it is needed.

rbac:PERMITTABLE(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZABLE(?x)

rbac:PERMITTED(?x) ∧ rbac:NOT_DENIED(?x) → rbac:AUTHORIZED(?x)

Text 24: AUTHORIZABLE and AUTHORIZED in SWRL.

AUTHORIZABLE and AUTHORIZED are then defined as in Text 24.

These rules look similar to the equivalent Prolog rules, but they use the class NOT_DENIED rather than a negation of the DENIED class.

In theory, some of the relationships that are defined using recursive SWRL rules could be defined using transitivity. *senior_to*, *junior_to* and *is_a* are defined as transitive; however, Protégé does not infer any relationships from transitivity, so defining it has no effect. Therefore, the recursive rules that are defined in Prolog also have to be defined in SWRL, despite the transitivity. However, Protégé does infer from inversity: each asserted *senior_to* relationship thus has a corresponding *junior_to* relationship.

5.5 Implementing SO-RBAC based on a hospital environment

The SO-RBAC implementation is illustrated through a scenario with roles, permissions, denials, seniority relationships, inclusion relationships and inheritance paths.

The following individuals were defined in class ROLE, reflecting a simplified hospital scenario.

- *r_admin*, *r_clerk*, *r_manager*
- *r_doctor*, *r_specialist_doctor*, *r_consultant*, *r_junior_staff_doctor*, *r_junior_staff_doctor_day*, *r_junior_staff_doctor_night*, *r_senior_staff_doctor*, *r_senior_staff_doctor_day*, *r_senior_staff_doctor_night*
- *r_technician*, *r_junior_technician*, *r_senior_technician*
- *r_nurse*, *r_senior_nurse*, *r_specialist_nurse*, *r_staff_nurse*, *r_staff_nurse_day*, *r_staff_nurse_night*, *r_student_nurse*, *r_student_nurse_day*, *r_student_nurse_night*
- *r_day_duty, r_night_duty*

The following individuals representing permission and denial assertions were created.

- PRA: *junior_staff_doctor_read_patient*, *junior_staff_doctor_read_room*, *junior_staff_doctor_read_vital_sign*, *junior_staff_doctor_read_ward*, *senior_staff_doctor_write_patient*, *senior_staff_doctor_write_room*, *senior_staff_doctor_write_vital_sign*, *consultant_write_vital_sign*, *consultant_read_computer*, *specialist_doctor_write_computer*, *student_nurse_read_patient*, *staff_nurse_read_room*, *staff_nurse_read_ward*, *staff_nurse_write_patient*, *senior_nurse_read_vital_sign*, *senior_nurse_write_ward*, *specialist_nurse_read_computer*, *specialist_nurse_write_room*, *specialist_nurse_write_vital_sign*, *specialist_nurse_write_computer*
- DRA: *consultant_read_room*, *consultant_write_ward*, *senior_nurse_read_ward*, *senior_staff_doctor_read_computer*, *staff_nurse_write_patient*

Seniority relationships were defined using *directly_senior_to* axioms to indicate the following hierarchies:

1. *r_doctor*: *r_junior_staff_doctor* → *r_senior_staff_doctor* → *r_consultant* → *r_specialist_doctor*
2. *r_nurse*: *r_student_nurse* → *r_staff_nurse* → *r_senior_nurse* → *r_specialist_nurse*
3. *r_technician*: *r_junior_technician* → *r_senior_technician*
4. *r_admin*: *r_clerk* → *r_manager*
5. *r_junior_staff_doctor: r_junior_staff_doctor_day, r_junior_staff_doctor_night*
6. *r_senior_staff_doctor: r_senior_staff_doctor_day, r_senior_staff_doctor_night*
7. *r_student_nurse: r_student_nurse_day, r_student_nurse_night*
8. *r_staff_nurse: r_staff_nurse_day, r_staff_nurse_night*

The following path inheritance axioms were defined.

```
inherits_pra_path(r_specialist_doctor, r_junior_staff_doctor)

inherits_pra_path(r_specialist_nurse, r_student_nurse)
```

Table 8: Numbers of users of each role defined in ontologies.

Role	*Small*	*Large*
clerk	1	2
manager	1	1
junior_staff_doctor	3	4
senior_staff_doctor	3	4
consultant	1	2
specialist_doctor	1	1
student_nurse	3	4
staff_nurse	3	4
senior_nurse	1	2
specialist_nurse	1	1
junior_technician	1	2
senior_technician	1	1

Two ontologies were defined, small and large, varying by the numbers of users and data items defined. Table 8 shows the numbers of users in the small and large ontologies.

One or more USER individuals for each ROLE was created (Table 8), except for the roles *r_admin*, *r_doctor*, *r_technician* and *r_nurse*, as these are intended as super-class roles to allow permissions to be defined for a particular type of user generically.

User individuals are named simply as *<role>_<n>*, where *<n>* is a number. The roles with day and night sub-roles defined each had 3 or 4 users defined, named for the main role. For example, *junior_staff_doctor_1* was assigned directly to *r_junior_staff_doctor*; *junior_staff_doctor_2* to *r_junior_staff_doctor_day*, and *junior_staff_doctor_3* to *r_junior_staff_doctor_night.* No personalized data were defined for any of these users, because they are not relevant in this static RBAC model.

The users were linked to roles using URA individuals, with one URA individual defined for each role in which a user could be defined.

Instances were created for object types *o_computer*, *o_patient*, *o_room*, *o_vital_sign* and o_ward. One instance of each type was created for the small scenario, and three of each type for the large scenario. The instances were named *<object_name>_n*, e.g. *patient_1*.

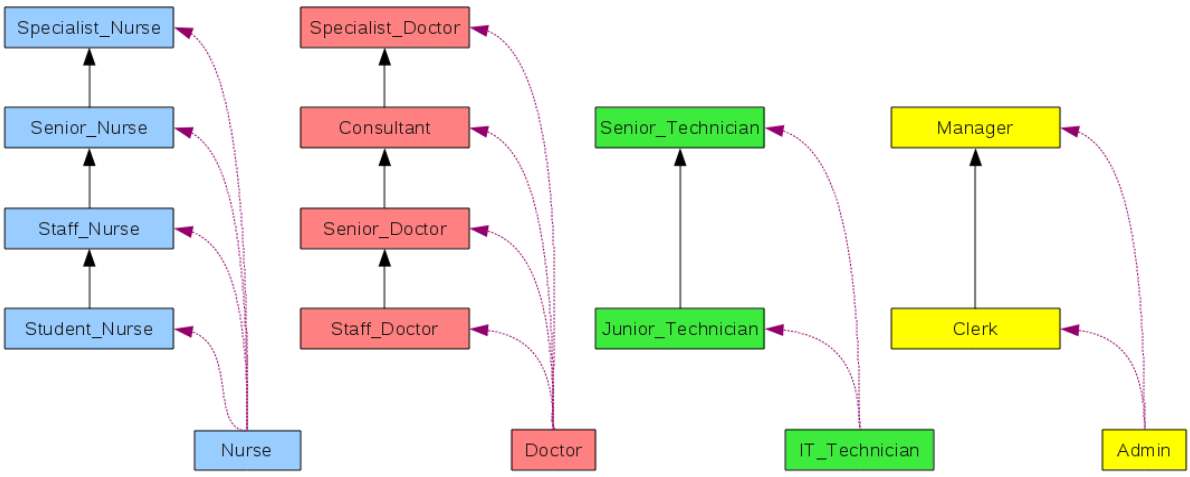

Figure 32: RBAC Model used to demonstrate SO-RBAC, excluding night and day duties. Solid (black) lines represent seniority (`d_s`) relationships. Dashed (purple) lines represent `is_a` relationships. Arrows show direction of inheritance of positive authorizations (permissions).

Figure 32 shows the full RBAC hierarchy.

5.6 Results of Implementation

The ontological model was implemented using the Protégé Ontology Editor, using the Protégé-OWL plugin. The Pellet Reasoner Inspector [114] was used to test the consistency of the ontology's classes, properties and instances, and to compute inferred class memberships. The SWRL rules were implemented through the SWRLTab plugin in Protégé [110], and executed using the Jess Rule Engine [115]. Jess is also built into Protégé via the SWRLJessTab, [116] which is a plug-in to the SWRLTab.

The classes ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE were populated with individuals representing all possible permutations of roles, users and permissions using a Perl script, which also added to PRA and DRA the individuals listed above.

The SWRL rules were run using Jess in the SWRLJessTab. The resulting ontology was then saved in a new file. The output from the SWRL tabs was copied and pasted into plain text files. The same rules and model were run in both Prolog and OWL+SWRL, and both produced the same results.

The numbers of unique axioms were obtained by copying and pasting the axioms from the SWRLJessTab into plain text files, and analyzing these using the Unix shell tools `sort` and `uniq`.

The number of SWRL rules exported is simply the number of SWRL rules that are run (i.e. are ticked in the SWRL tab). All OWL classes in the ontology are exported when a set of SWRL rules is run. However, only the individuals in classes and properties mentioned in the exported SWRL rules are exported, along with the axioms that relate them. The axioms inferred are the results of running the SWRL rules. The same relationship may be inferred

many times, resulting in some non-unique axioms, as found in Steps 2 and 4 in this experiment. In Step 2, two individuals were inferred twice as members of PRA_FULL (*consultant_write_vital_sign* and *specialist_nurse_read_computer*), resulting in two non-unique inferences. This is the same for both ontologies, which differ in the numbers of users and user-role assignments, and not in the number of roles or role-permission assignments. Step 4 produces a very large number of non-unique axioms. This is because sqwrl:notElement compares each individual *?x* with each individual *?y* in the set *?d*, to check for non-membership of *?x* in *?d*, resulting in each NOT_DENIED(?x) axiom being inferred as many times are there are *?y* individuals in DENIED.

Table 9: Numbers of rules, classes, individuals and axioms reported by SWRL for the small ontology.

	Step 1	***Step 2***	***Step 3***	***Step 4***	***Step 5***
SWRL rules exported to Jess	7	2	2	2	2
OWL classes exported to Jess	75	75	75	75	75
OWL individuals exported to Jess	24	731	388	232	160
OWL axioms exported to Jess	41	935	937	211	0
OWL axioms inferred	127	64	131	5,350	98
Unique triples created	127	62	131	166	98

Table 10: Numbers of rules, classes, individuals and axioms reported by SWRL for the large ontology.

	Step 1	***Step 2***	***Step 3***	***Step 4***	***Step 5***
SWRL rules exported to Jess	7	2	2	2	2
OWL classes exported to Jess	75	75	75	75	75
OWL individuals exported to Jess	24	731	820	654	528
OWL axioms exported to Jess	31	935	2,210	633	0
OWL axioms inferred	127	64	423	59,214	306
Unique triples created	127	62	423	-	30

Table 9 shows the results of running SWRL for the small ontology. The 5,350 axioms inferred in Step 4 include 5,328 NOT_DENIED axioms. As described above, sqwrl:notElement makes 180×37=6,660 comparisons between individuals in USER_PERMISSION_ASSIGNABLE and DENIED, resulting in 144×37=5,328 inferred NOT_DENIED axioms. The other 22 inferred axioms were unique PERMITTED axioms.

Table 10 shows the results of running SWRL for the large ontology. In Step 4, it was not possible to paste the 59,214 axioms into a plain text file, due to memory limitations (and this step took an extremely long time to run to completion).

The numbers of triples of affected classes and properties at each stage were determined by exporting the OWL files as *n*-triple files, and analyzing these using the Unix shell tool `grep`.

These should correspond to the numbers of unique OWL axioms inferred in each step.

Table 11: Numbers of triples at stage 1.

Property	***Small***	***Large***
senior_to	26	26
included_in	61	61
inherits_pra	40	40
Total	***127***	***127***

Table 12: Numbers of triples at stage 2.

Class	***Small***	***Large***
PRA_FULL	49	49
DRA_FULL	13	13
Total	***62***	***62***

Table 13: Numbers of triples at stage 3.

Class	***Small***	***Large***
PERMITTABLE	95	300
DENIED	37	124
Total	***132***	***424***

Table 14: Numbers of triples at stage 4.

Class	***Small***	***Large***
NOT_DENIED	144	477
PERMITTED	22	66
Total	***166***	***543***

Table 15: Numbers of triples at stage 5.

Class	***Small***	***Large***
AUTHORIZABLE	79	249
AUTHORIZED	19	57
Total	***98***	***306***

The same numbers of triples were found for both ontologies, because Step 1 only operates on roles, and both have the same roles (Table 11). Note that OWL did not infer the 26 *junior_to* axioms, which were instead inferred in Protégé as a result of *junior_to* being defined as inverse to *senior_to*.

Two individuals were inferred twice as members of PRA_FULL (*consultant_write_vital_sign* and *specialist_nurse_read_computer*), resulting in two non-unique inferences (Table 12). The number of triples in the affected classes is one more than the number of inferences made in Step 3, due to the dummy individual in DENIED on initialization (Table 13).

Finding 543 triples and 477 NOT_DENIED triples in the large ontology (Table 14) is consistent with 59,214 axioms inferred by the SWRL rules, as this number is 124×477+66 (DENIED × NOT_DENIED + PERMITTED). The AUTHORIZABLE and AUTHORIZED classes were populated (Table 15).

5.7 Results of SO-RBAC Process in Protégé

This section displays screen shots (Figs. 33–56) captured using the Protégé OWLViz tab [117] at various stages of reasoning. All screenshots are taken from the small ontology.

5.7.1 Classes and Individuals

5.7.1.1 General

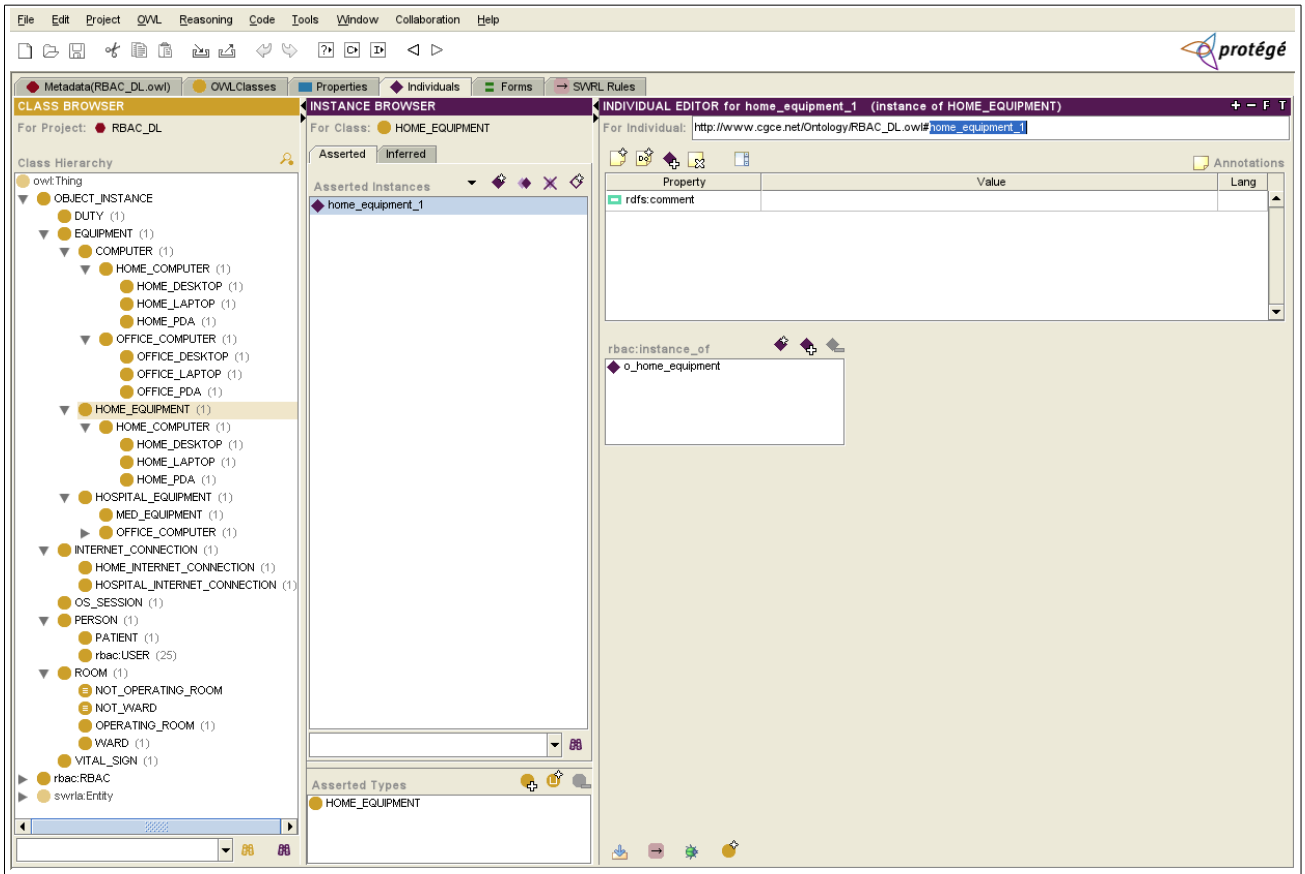

Figure 33: The OBJECT_INSTANCE hierarchy in our example.

Figure 33 shows the hierarchy of object classes under the OWL class OBJECT_INSTANCE. The cursor is focused on one class, HOME_EQUIPMENT, which contains one member, *home_equipment_1*, which is a member of class and is related to OBJECT_TYPE individual *o_home_equipment* via property *instance_of*. This can be expressed as the triple (home_equipment_1) (rbac:instance_of) (o_home_equipment).

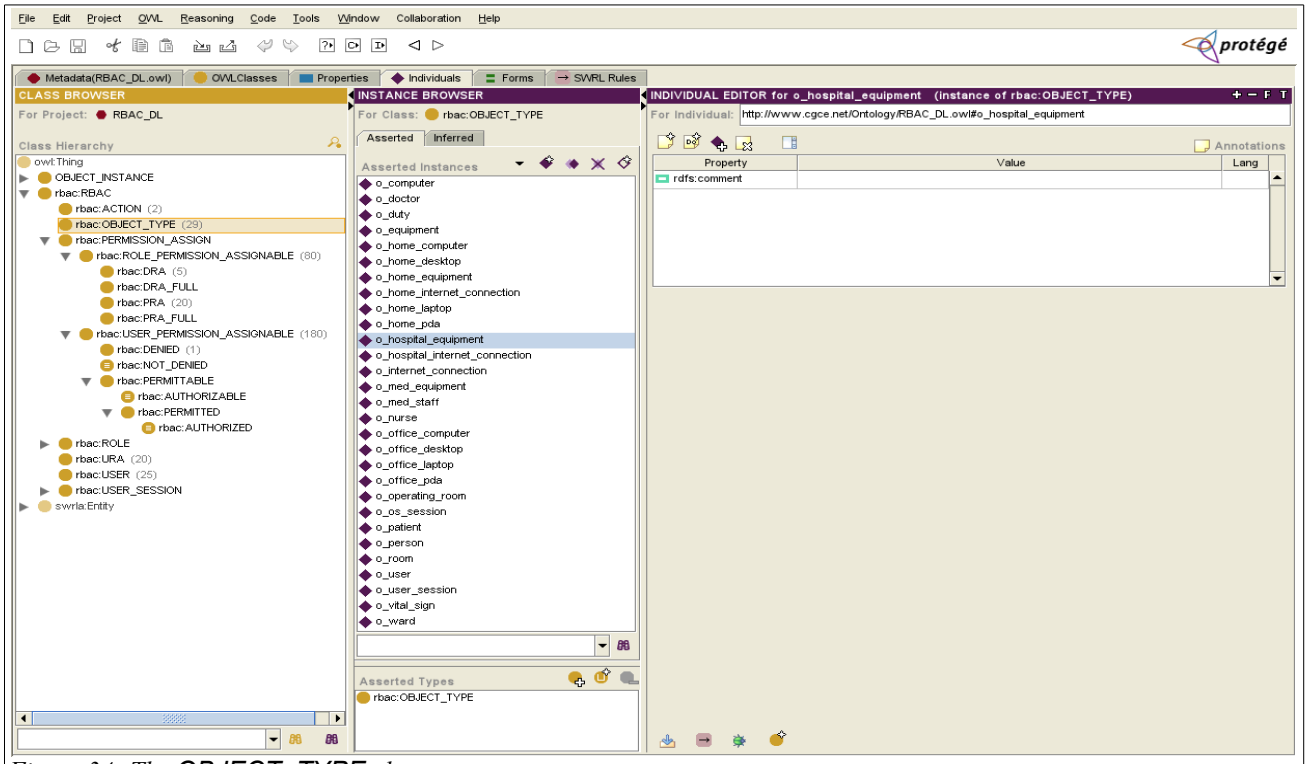

Figure 34: The OBJECT_TYPE class.

Figure 34 shows the OBJECT_TYPE class, which contains individuals representing types of objects. These are not the same as the individuals for the objects themselves, which are in the OBJECT_INSTANCE class.

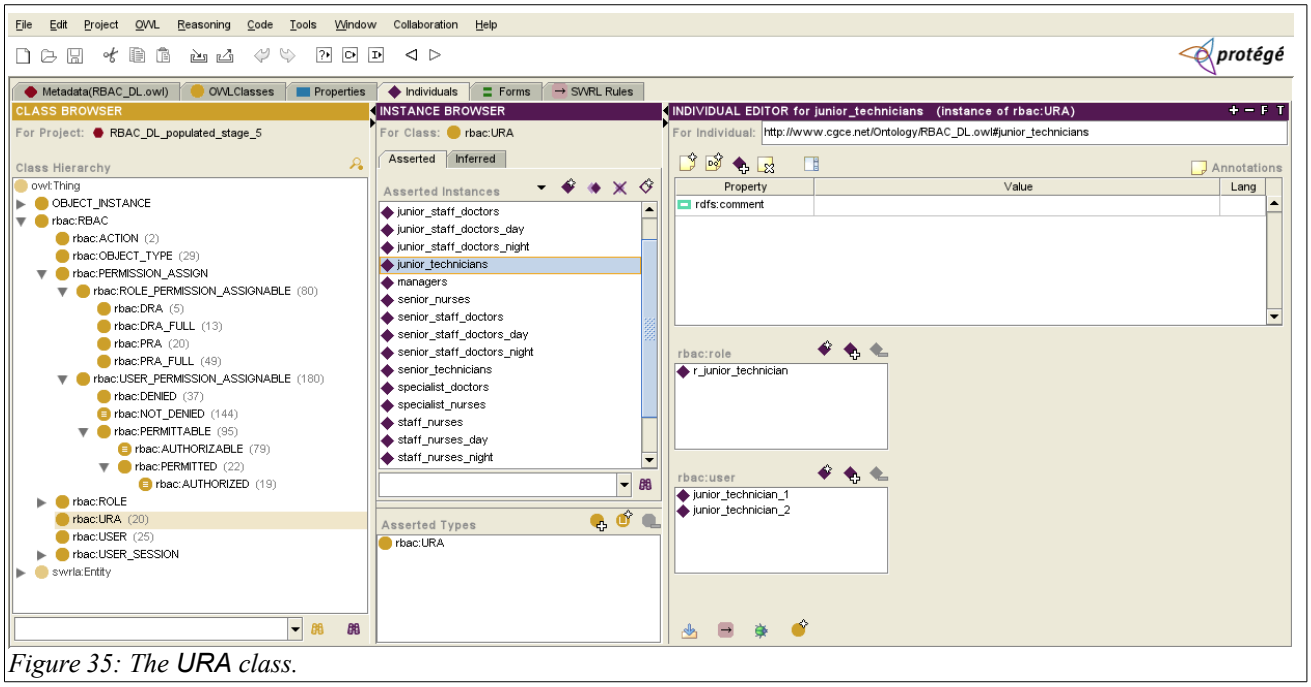

Figure 35: The URA class.

Figure 35 shows the URA class, focusing on the individual *junior_technicians*. junior_technicians is related to *r_junior_technician* via *role*, and to *junior_technician_1* and *junior_technician_2* via *user*. This is expressed as the following triples:

junior_technicians rbac:role r_junior_technician
junior_technicians rbac:user junior_technician_1
junior_technicians rbac:user junior_technician_2

It expresses the following URA relationships:

```
ura(junior_technician_1, r_junior_technician)
ura(junior_technician_2, r_junior_technician)
```

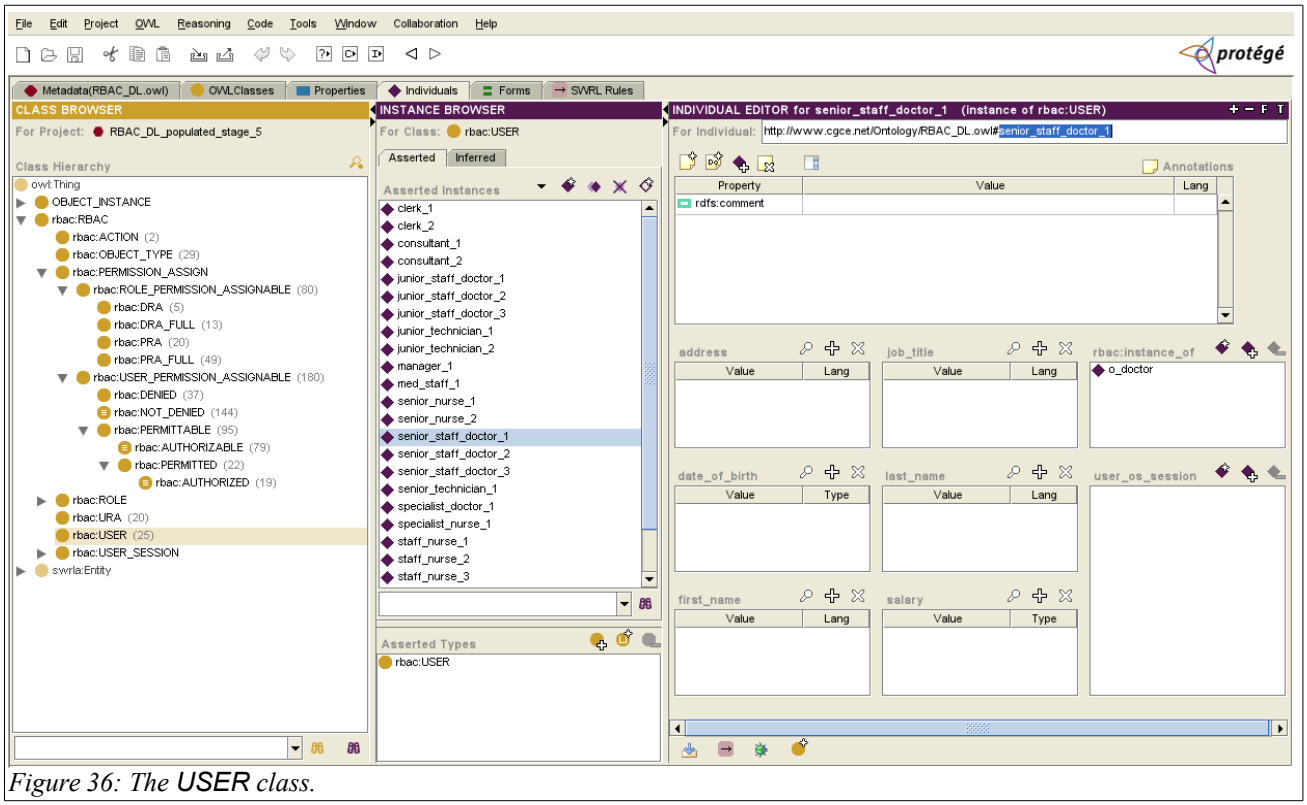

Figure 36: The USER class.

Figure 36 shows the USER class, focusing on the individual *senior_staff_doctor_1*. Although various properties that could be relevant to the personal characteristics of users been defined, they have been left empty in this model, as they are not relevant to the running of this static RBAC model. However, the USER individual represents an object (USER is also a sub-class of OBJECT_INSTANCE). Therefore, the triple (senior_staff_doctor_1) (rbac:instance_of) (o_doctor) is defined. Note that this is *not* a role membership assertion (*o_doctor* is not a ROLE, but an OBJECT_TYPE): that would be in the URA class, as described in Figure 35 above.

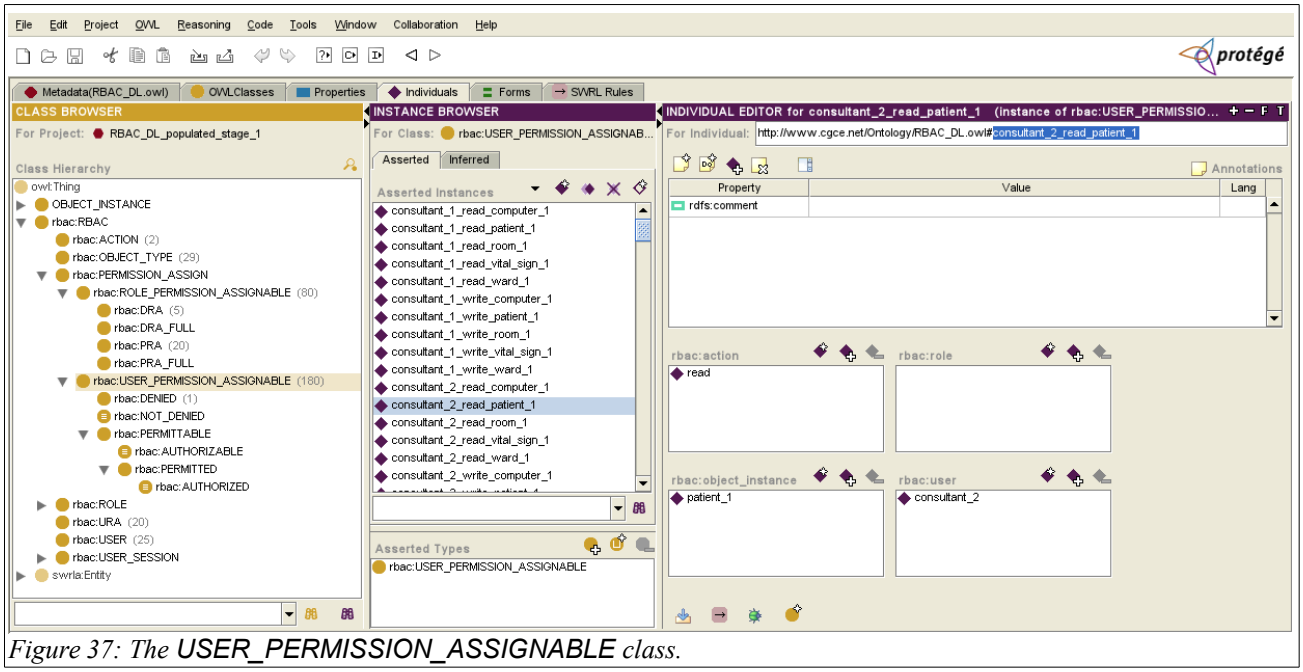

Figure 37: The USER_PERMISSION_ASSIGNABLE class.

USER_PERMISSION_ASSIGNABLE (Figure 37) is the class containing all potential assignments of permissions and denials to users.

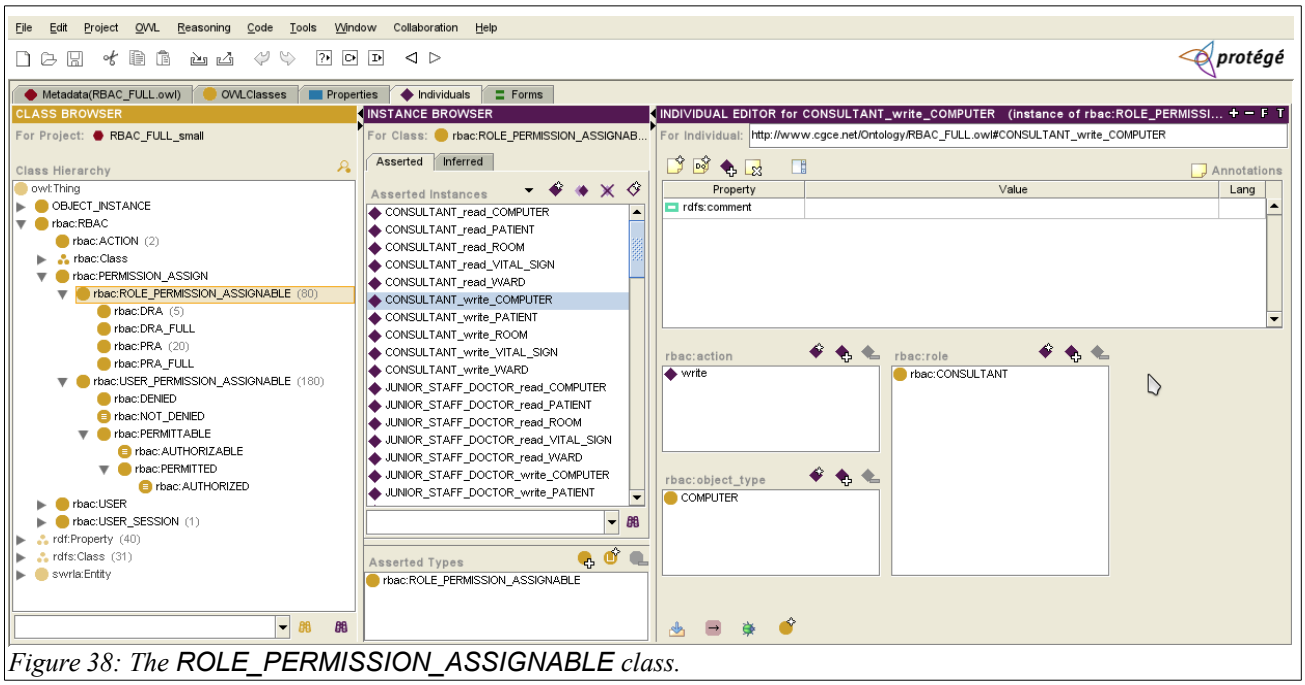

Figure 38: The ROLE_PERMISSION_ASSIGNABLE class.

ROLE_PERMISSION_ASSIGNABLE (Figure 38) is the class containing all potential assignments of permissions and denials to roles.

5.7.1.2 Initialization

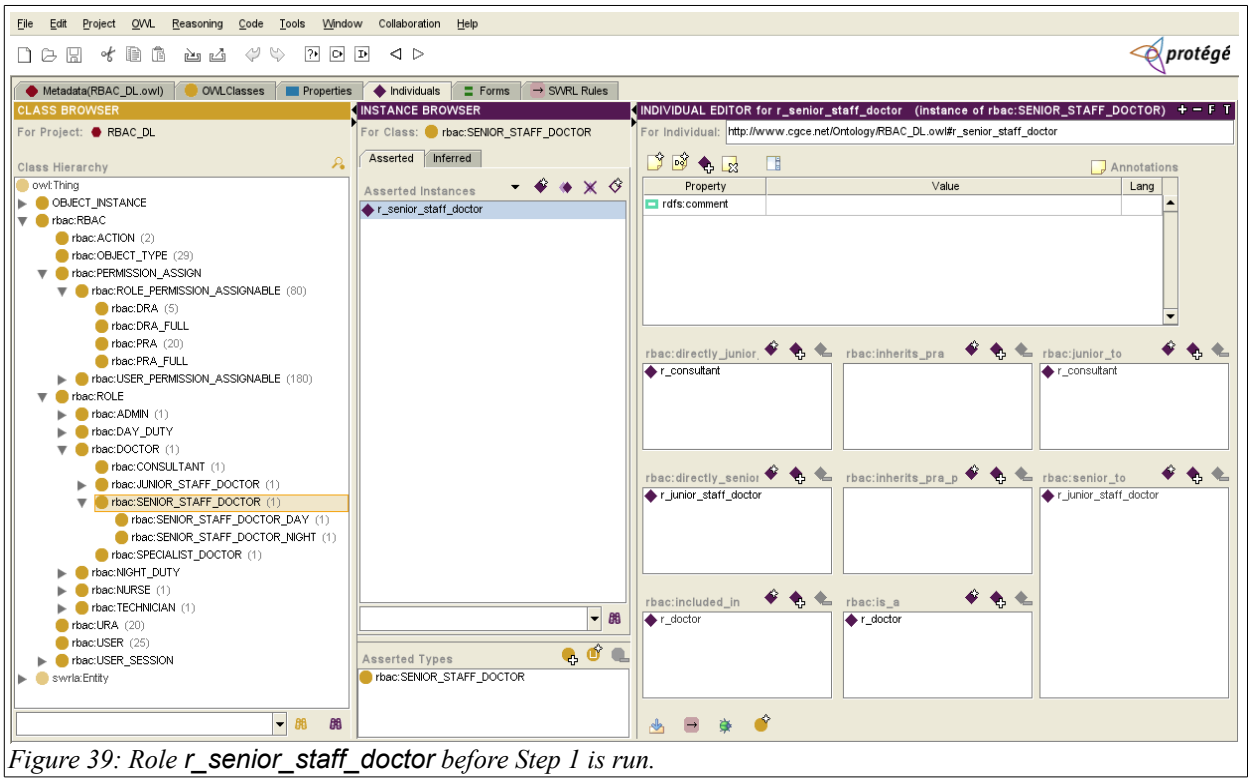

Figure 39: Role r_senior_staff_doctor before Step 1 is run.

Figure 39 shows the role *r_senior_staff_doctor* in the hierarchy under ROLE before Step 1 is run. *r_senior_staff_doctor* is a member of SENIOR_STAFF_DOCTOR, and can therefore be inferred to be a member of ROLE through sub-classing when the SWRL rules in Step 1 are run.

Of the property relationships shown in Figure 39, only two ((r_senior_staff_doctor) (senior_to) (r_senior_staff_doctor) and (r_senior_staff_doctor) (is_a) (r_doctor)) are explicitly asserted in the model. The remaining relationships are inferred through inversity and sub-properties, as explained earlier.

5.7.2 Reasoning

5.7.2.1 Stage 1

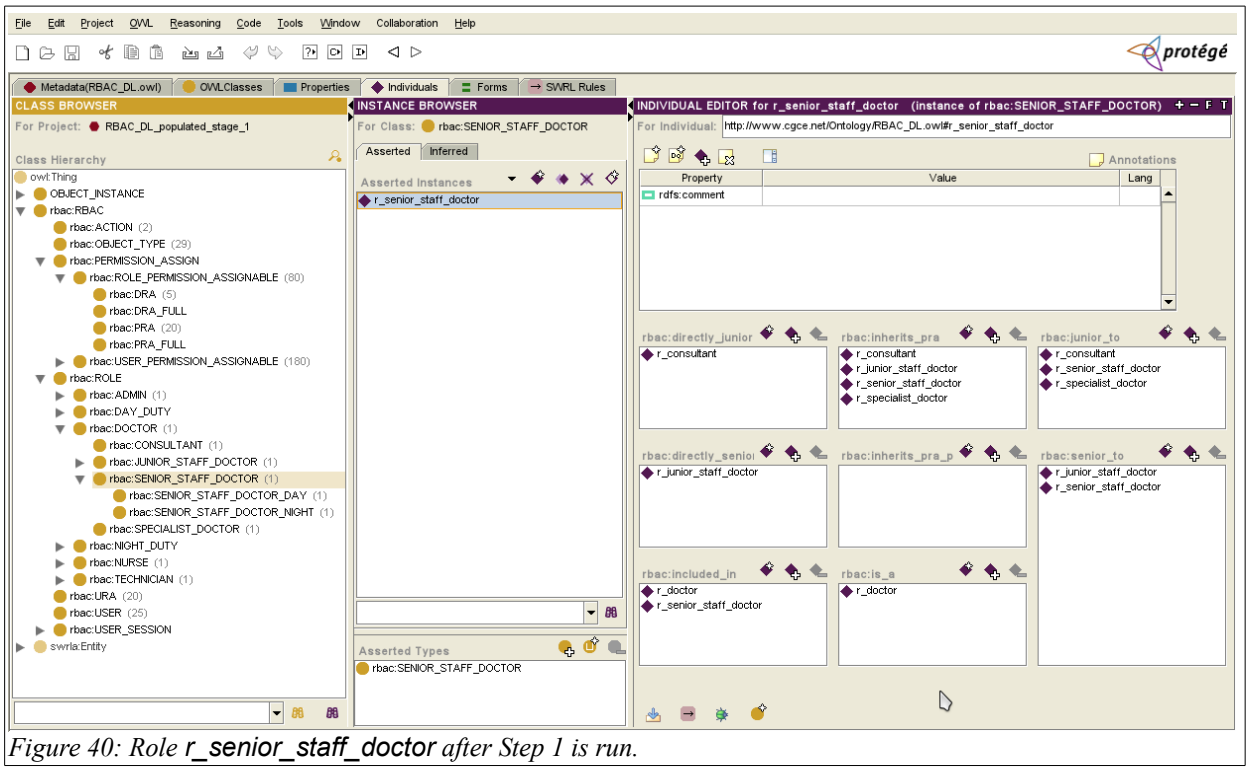

Figure 40: Role r_senior_staff_doctor after Step 1 is run.

After running Step 1, additional property relationships for individuals in sub-classes of ROLE are inferred, based on the rules, and are added to the model, as shown in Figure 40).

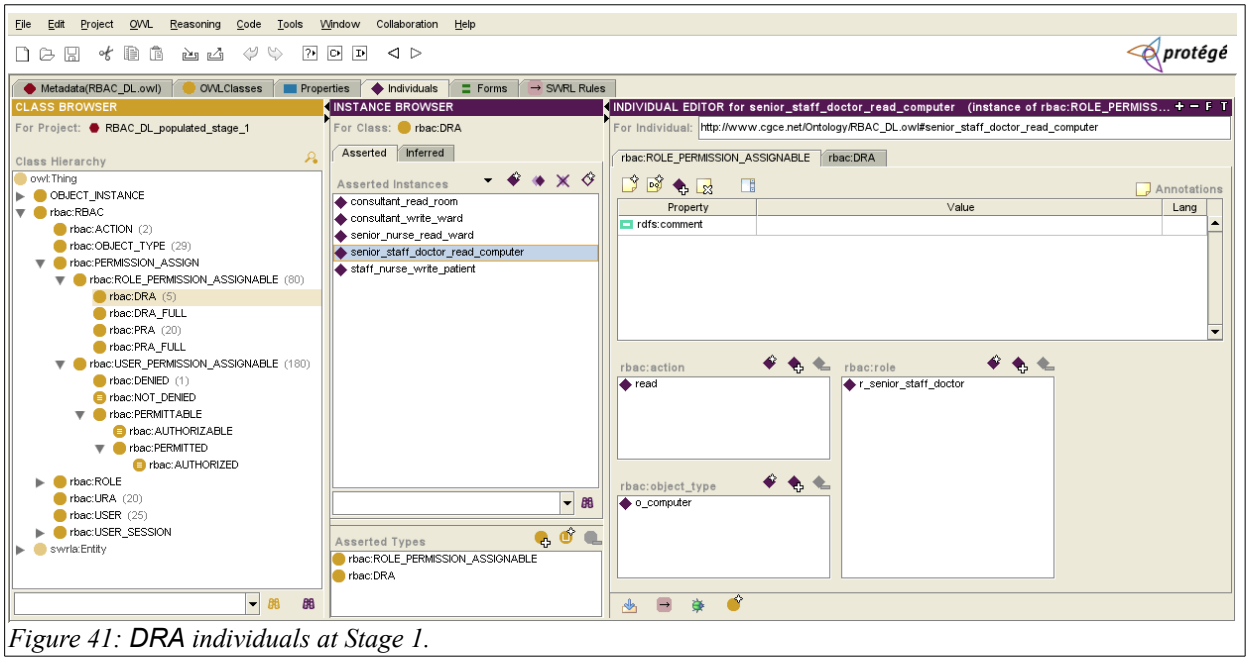

Figure 41: DRA individuals at Stage 1.

Figure 41 shows DRA at Stage 1, containing the individuals with which it is initialized.

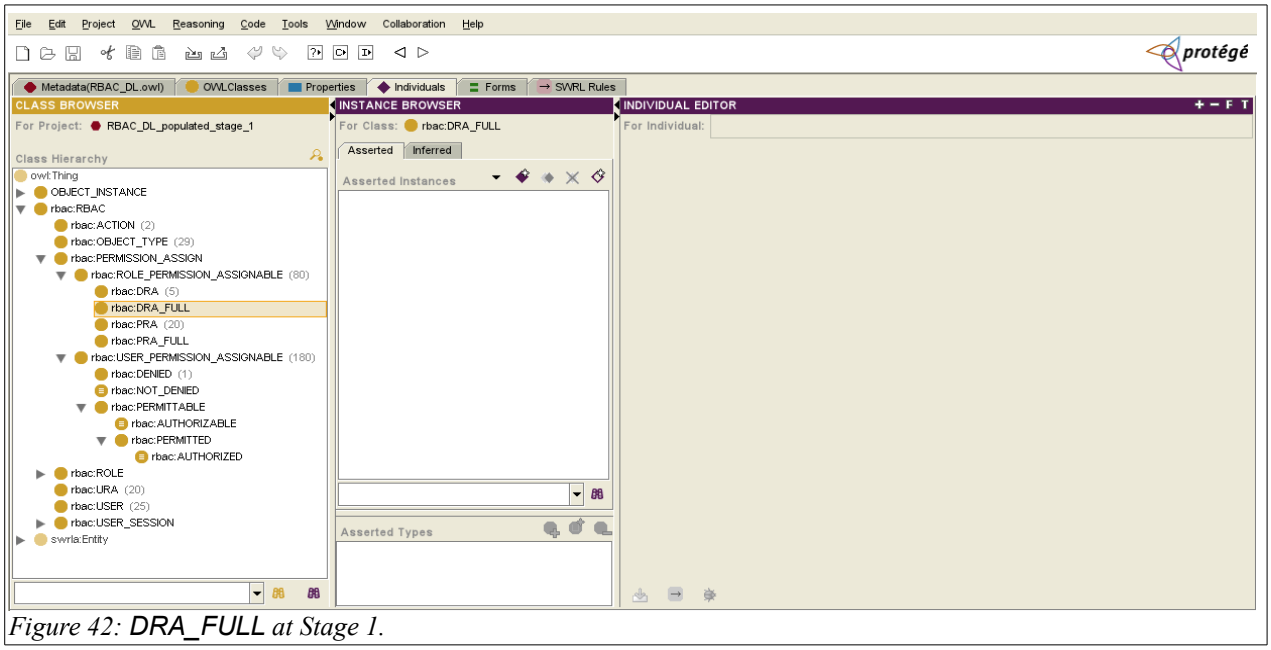

Figure 42: DRA_FULL at Stage 1.

Figure 42 shows DRA_FULL at Stage 1. This class is empty because it has not yet been populated in Step 2.

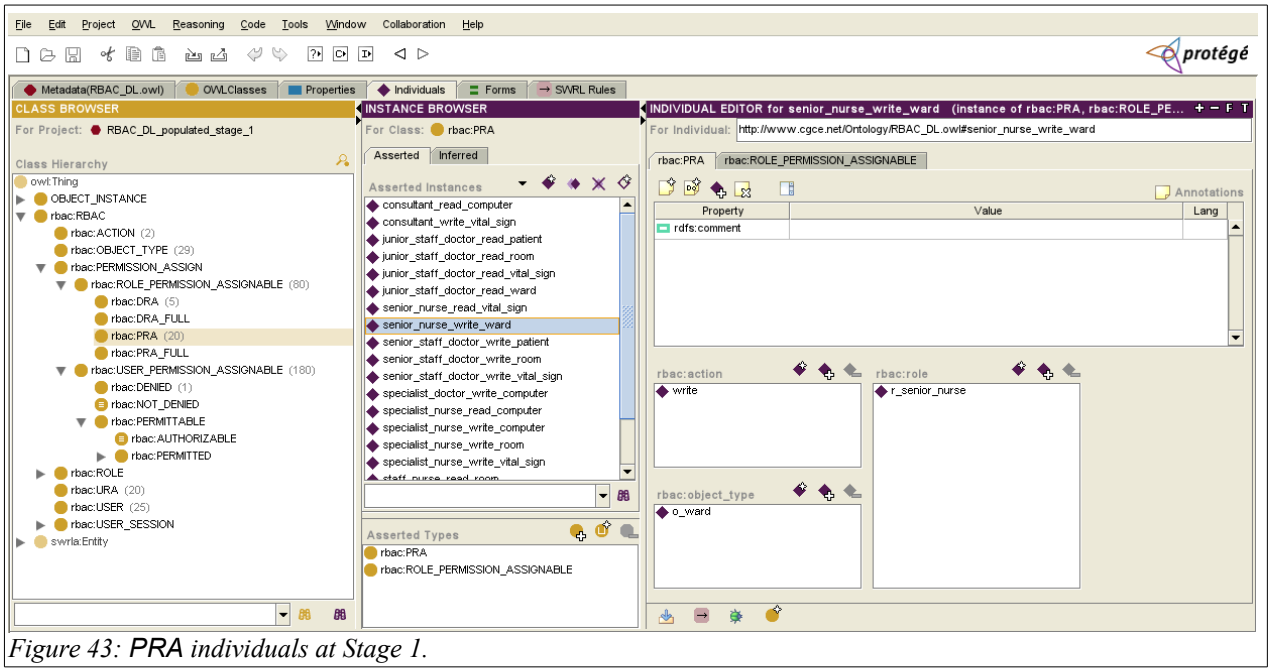

Figure 43: PRA individuals at Stage 1.

Figure 43 shows PRA at Stage 1, containing the individuals with which it is initialized.

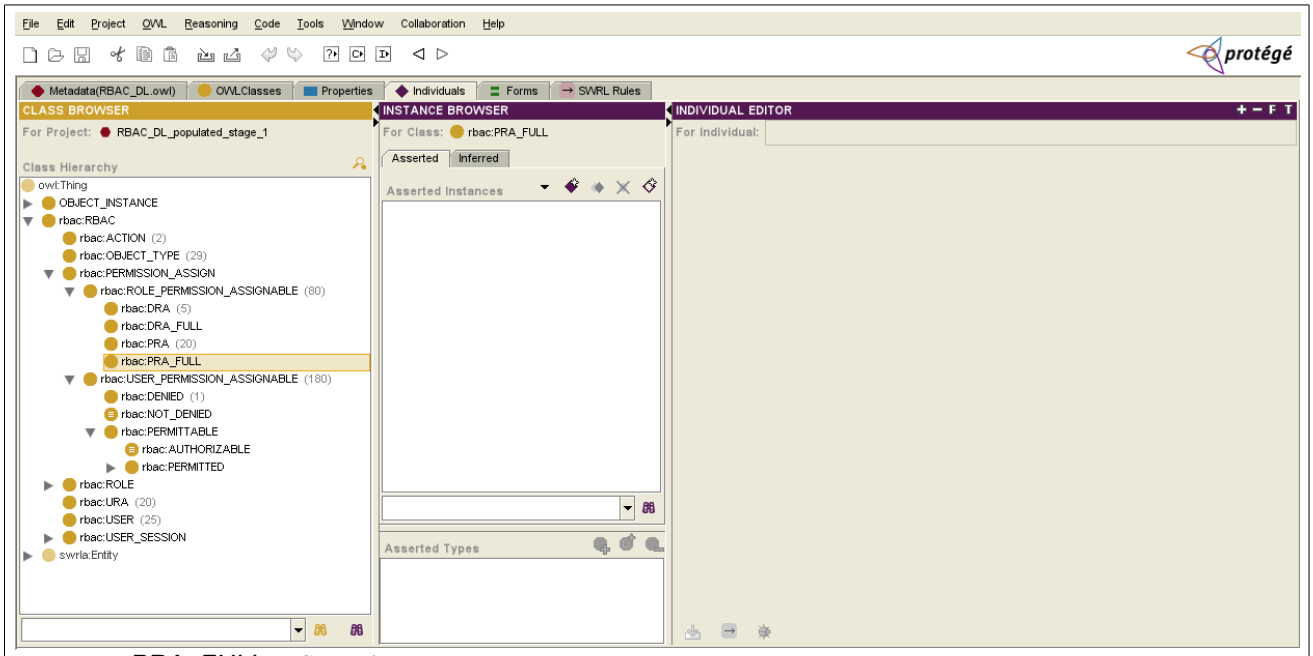

Figure 44: PRA_FULL at Stage 1.

Figure 44 shows PRA_FULL at Stage 1. This class is empty because it has not yet been populated in Step 2.

5.7.2.2 Stage 2

Figure 45 shows DENIED at Stage 2. The only individual present is the dummy individual *DENIED_1*, which is needed for 3_not_denied to work.

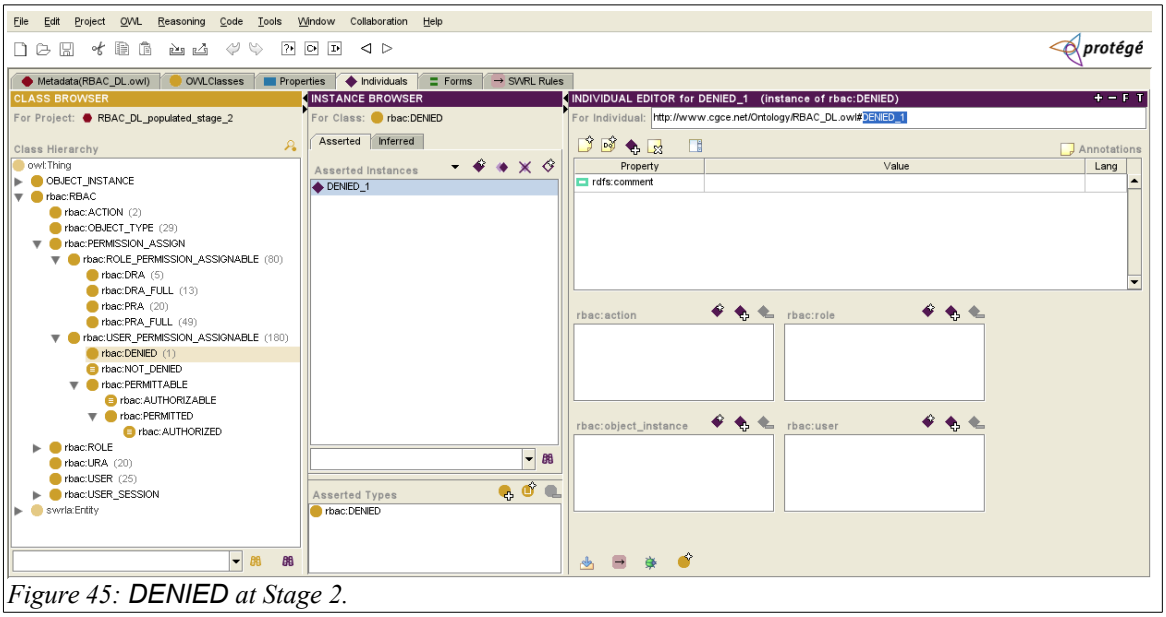

Figure 45: DENIED at Stage 2.

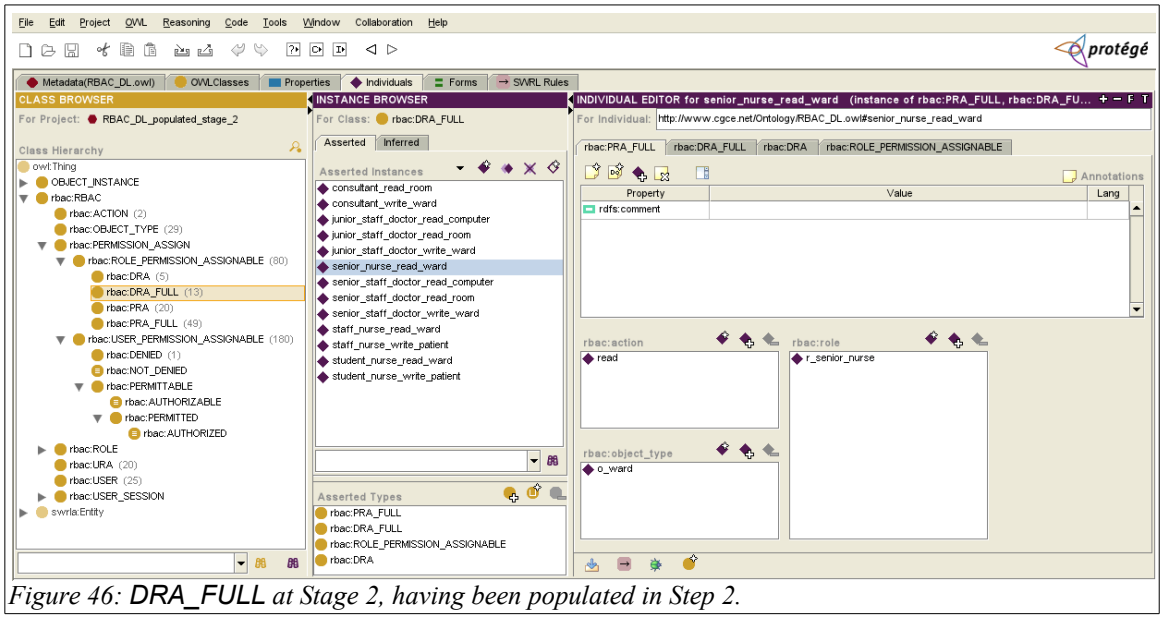

Figure 46: DRA_FULL at Stage 2, having been populated in Step 2.

Figure 46 shows DRA_FULL after it has been populated in Step 2. The individual *senior_nurse_read_ward* is highlighted. Notice the entries in Asserted Types, which lists all the classes of which an individual is a member. Naturally, DRA_FULL appears; so does DRA, as this particular DRA_FULL member is directly inferred to be in DRA_FULL as a result of being in DRA (see Figure 4). It is also in PRA_FULL: Step 2 infers this individual as representing a role permission as well as a denial. [This means that USER individuals will be both PERMITTABLE and DENIED in later steps; this conflict is resolved in AUTHORIZABLE by having denials over-ride permissions.] Notice also that the individual is a member of ROLE_PERMISSION_ASSIGNABLE; membership of this is essential for the rules in Step 2 to work.

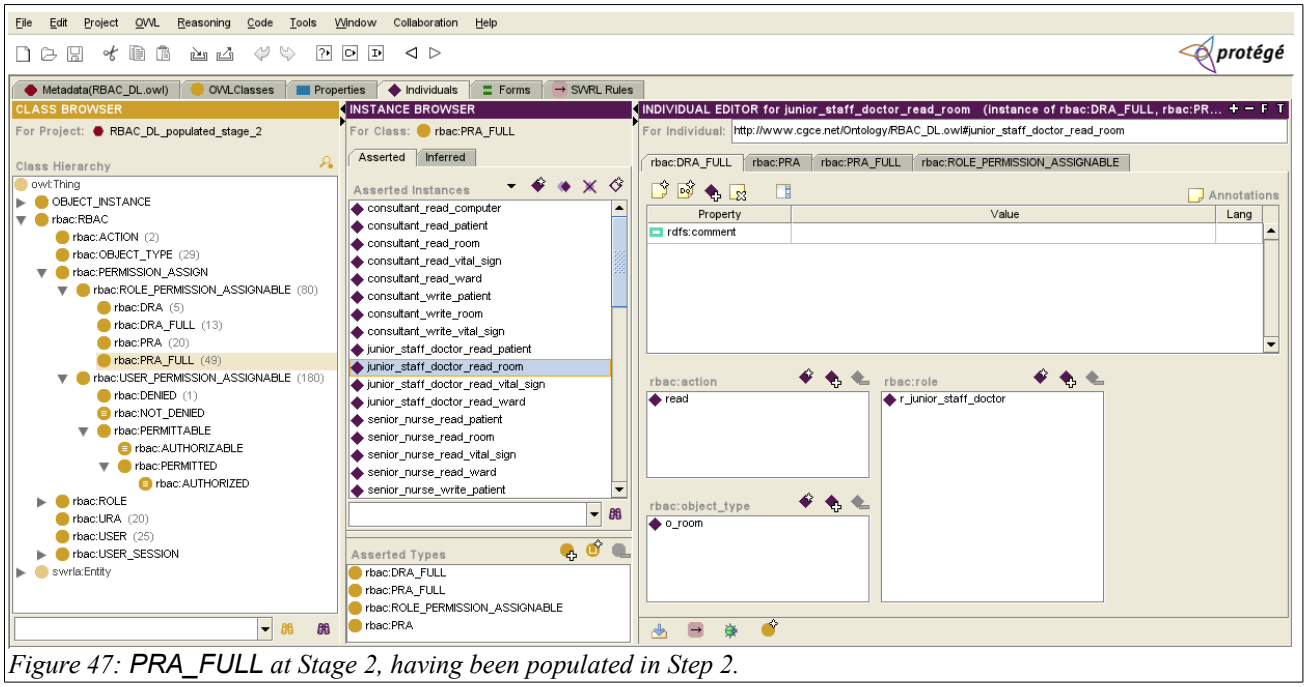

Figure 47: PRA_FULL at Stage 2, having been populated in Step 2.

Figure 47 shows PRA_FULL after Step 2 has run. This class is analogous to DRA_FULL.

5.7.2.3 Stage 3

Figure 48 shows DENIED after Step 3 has run. Note that the individual highlighted also belongs to USER_PERMISSION_ASSIGNABLE (as it has to for this step to run) and PERMITTABLE. Thus we have:

PERMITTABLE(staff_nurse_3_read_ward_1)

DENIED(staff_nurse_3_read_ward_1)

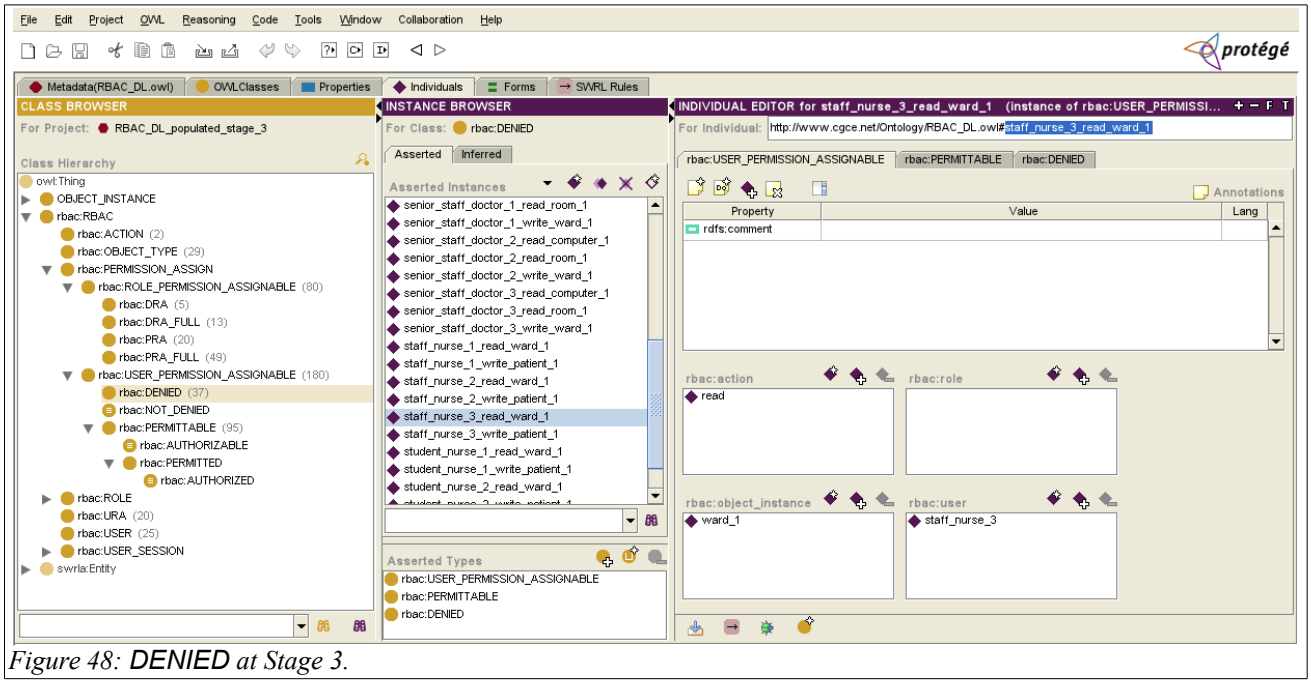

Figure 48: DENIED at Stage 3.

Given the property relationships of this individual, this is like saying:

```
permittable( staff_nurse_3, read, ward_1).
denied( staff_nurse_3, read, ward_1)
```

In Step 5, this conflict will be resolved by the denial over-riding the permission.

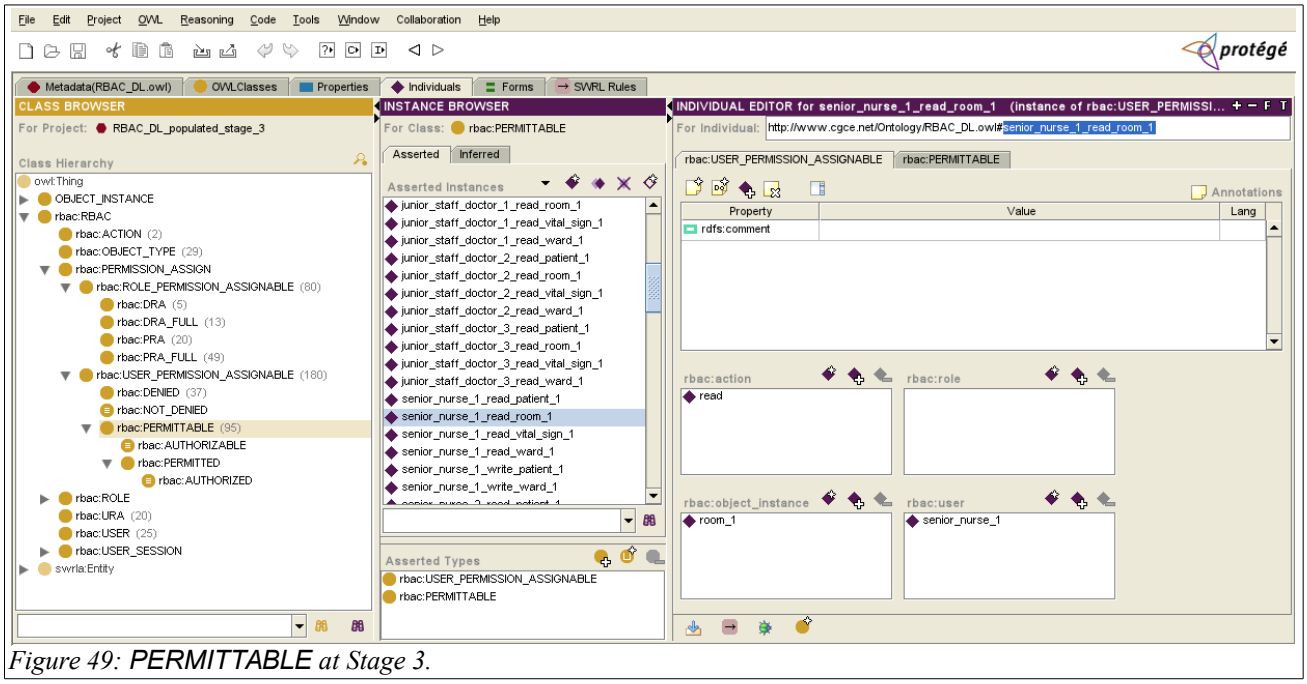

Figure 49: PERMITTABLE at Stage 3.

Figure 49 shows the PERMITTABLE class after Step 3 is run. Again, the individual is also a member of USER_PERMISSION_ASSIGNABLE, but is not a member of DENIED. Thus we only have

```
permittable(senior_nurse_1, read, room_1)
```

5.7.2.4 Stage 4

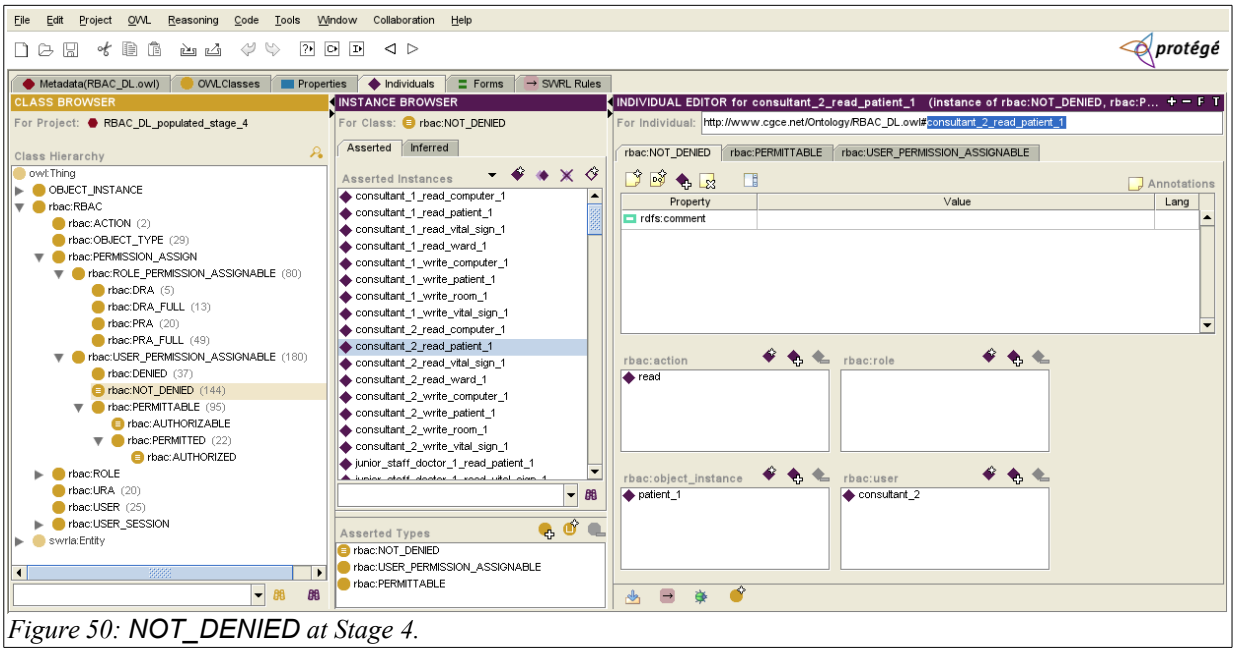

Figure 50: NOT_DENIED at Stage 4.

Figure 50 shows the results of populating NOT_DENIED in Step 4. Although each individual's membership of this class is defined many times due to the way the populating rule runs (as discussed earlier) each individual still appears only once in the Protégé window.

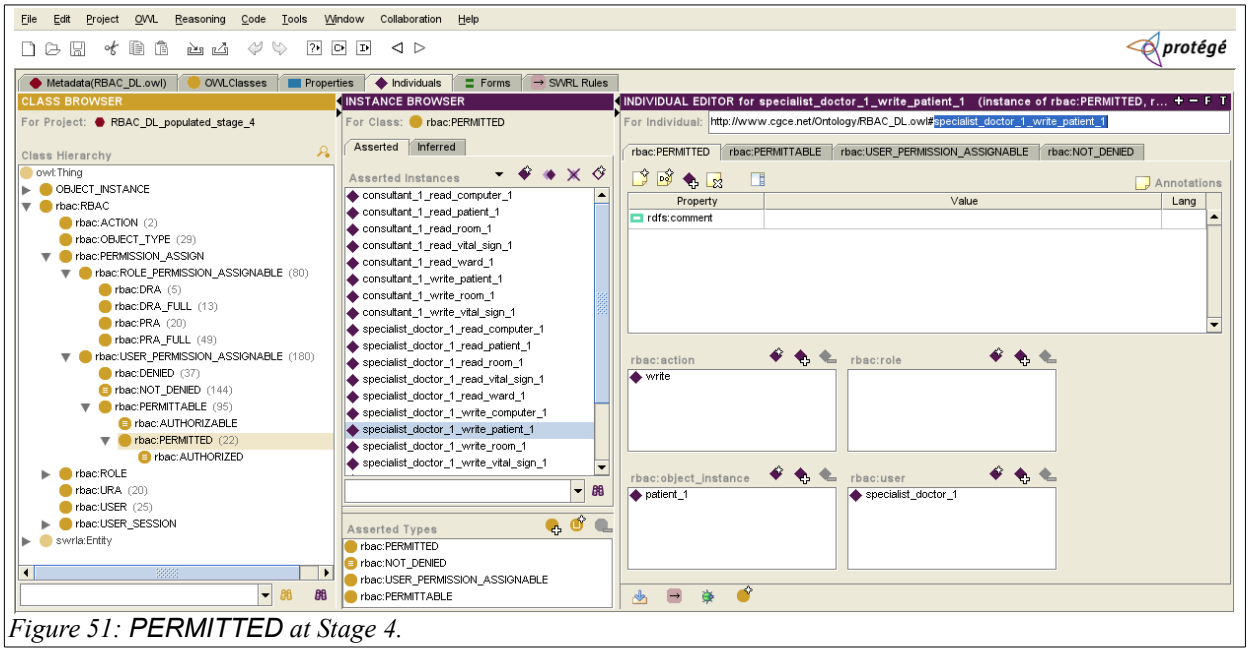

Figure 51: PERMITTED at Stage 4.

Figure 51 shows the results of populating PERMITTED in Step 4. As well as being a member of PERMITTABLE and USER_PERMISSION_ASSIGNABLE (as is necessary for membership of PERMITTED), the highlighted individual also belongs to NOT_DENIED. At Stage 4, every individual in USER_PERMISSION_ASSIGNABLE, PERMITTABLE and PERMITTED will belong to either DENIED or NOT_DENIED.

5.7.2.5 Stage 5

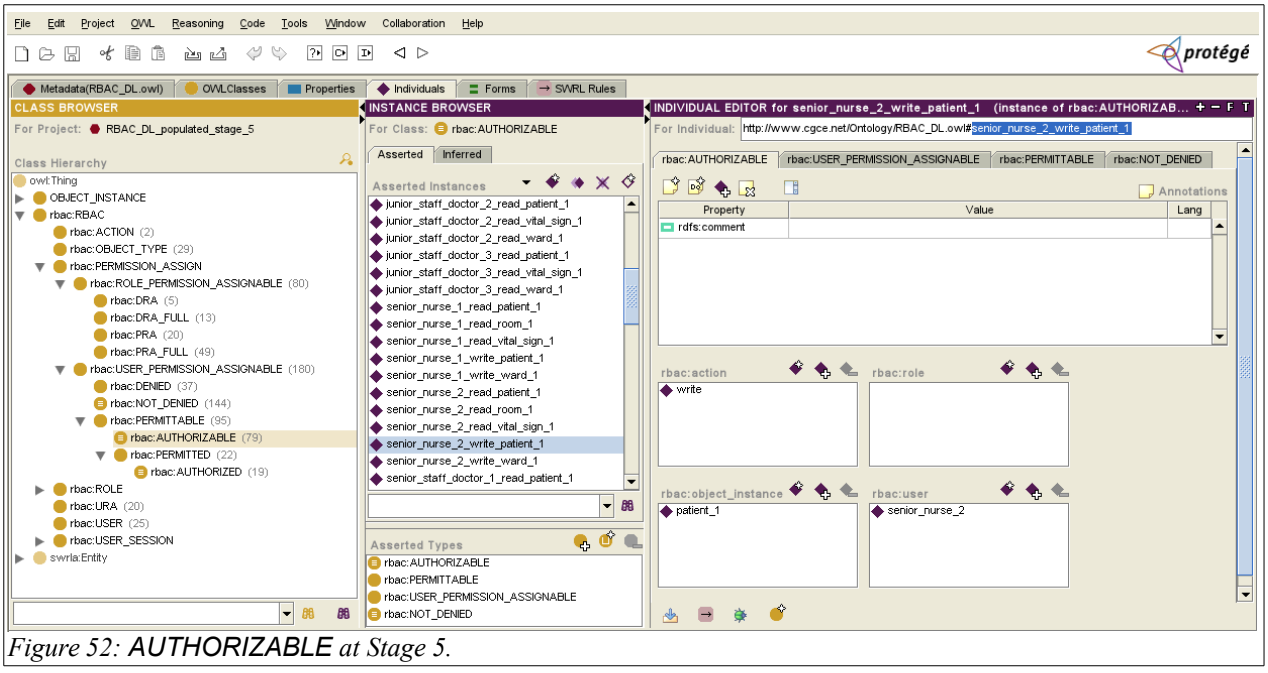

Figure 52: AUTHORIZABLE at Stage 5.

Figure 52 shows AUTHORIZABLE after Step 5. All individuals belonging to AUTHORIZABLE must by definition belong to the other three types listed for this individual (PERMITTABLE, USER_PERMISSION_ASSIGNABLE and NOT_DENIED).

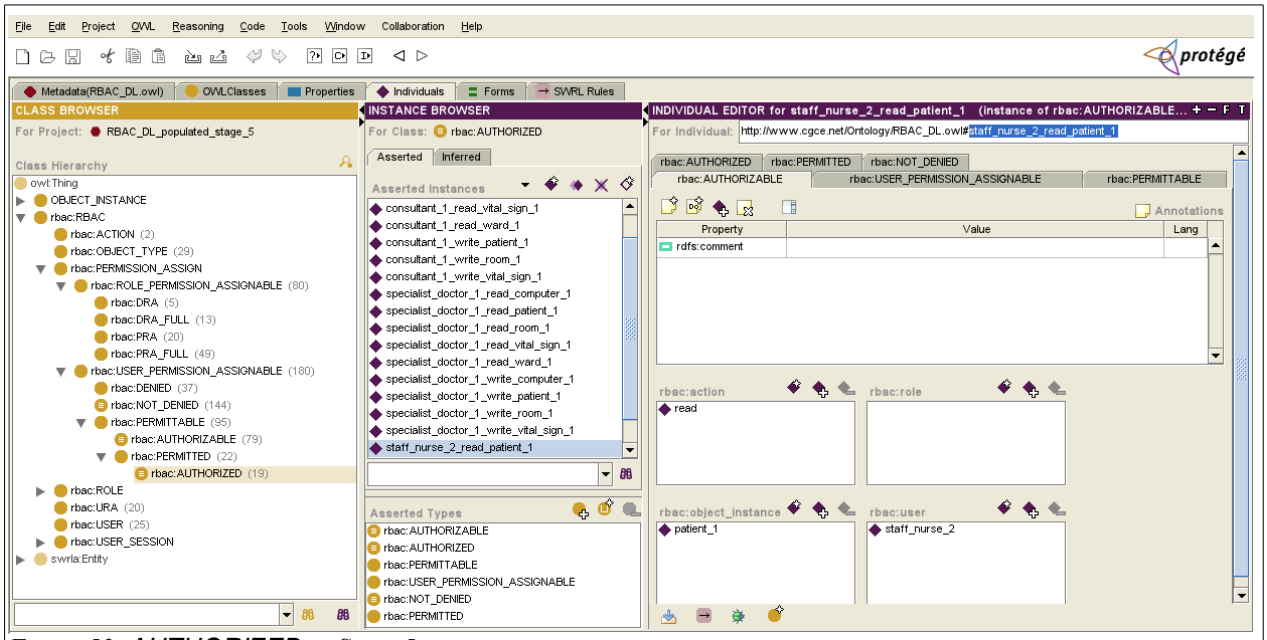

Figure 53: AUTHORIZED at Stage 5.

Figure 53 shows AUTHORIZED after Step 5. Again, any individual in AUTHORIZED must be a member of the other 5 classes listed here.

5.7.3 SWRL Rules Tab

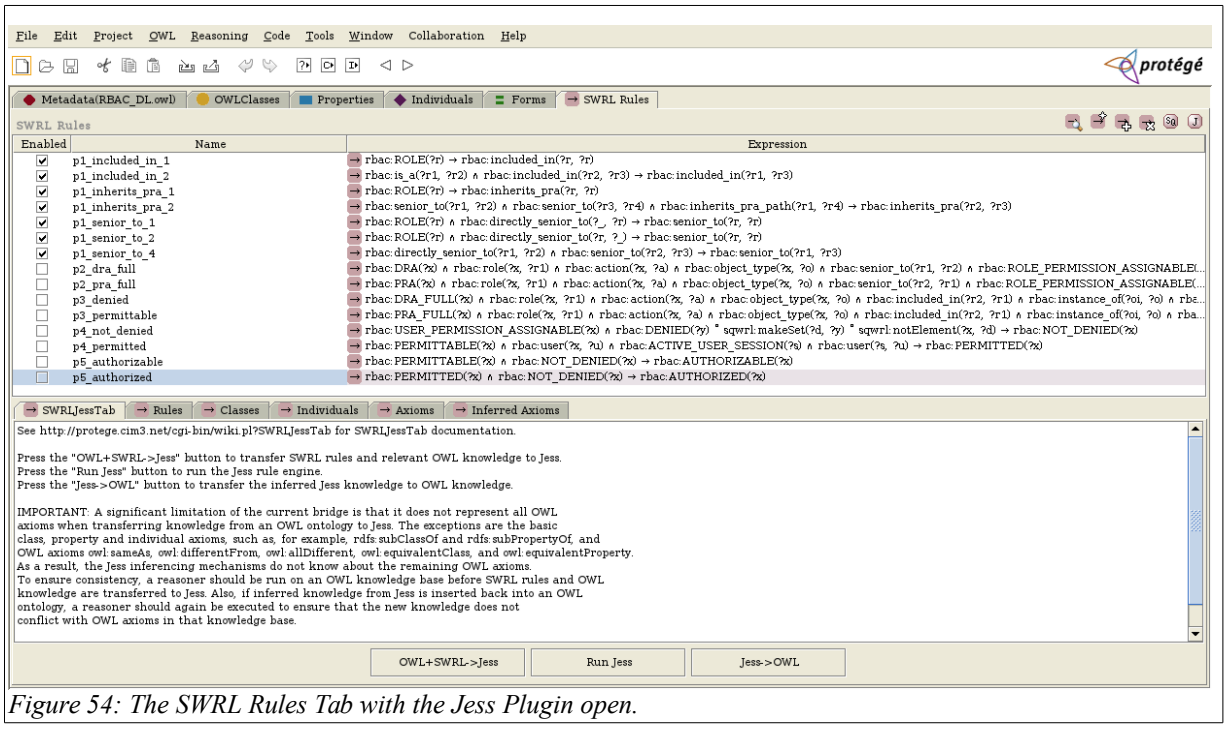

Figure 54: The SWRL Rules Tab with the Jess Plugin open.

Figure 54 shows the SWRL Rules Tab with the Jess Plugin open. Note that the seven Step 1 rules are ticked, indicating that they will be fired when OWL+SWRL→Jess button is clicked.

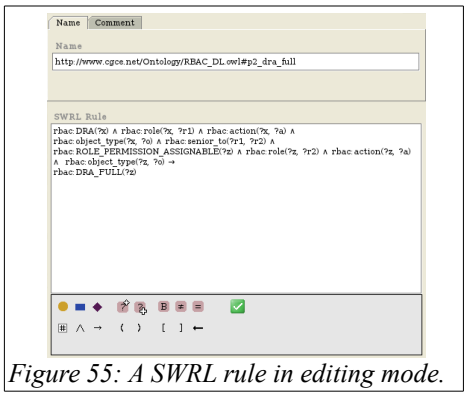

Figure 55: A SWRL rule in editing mode.

Figure 55 shows a SWRL rule (in this case 2_dra_full) in edit mode.

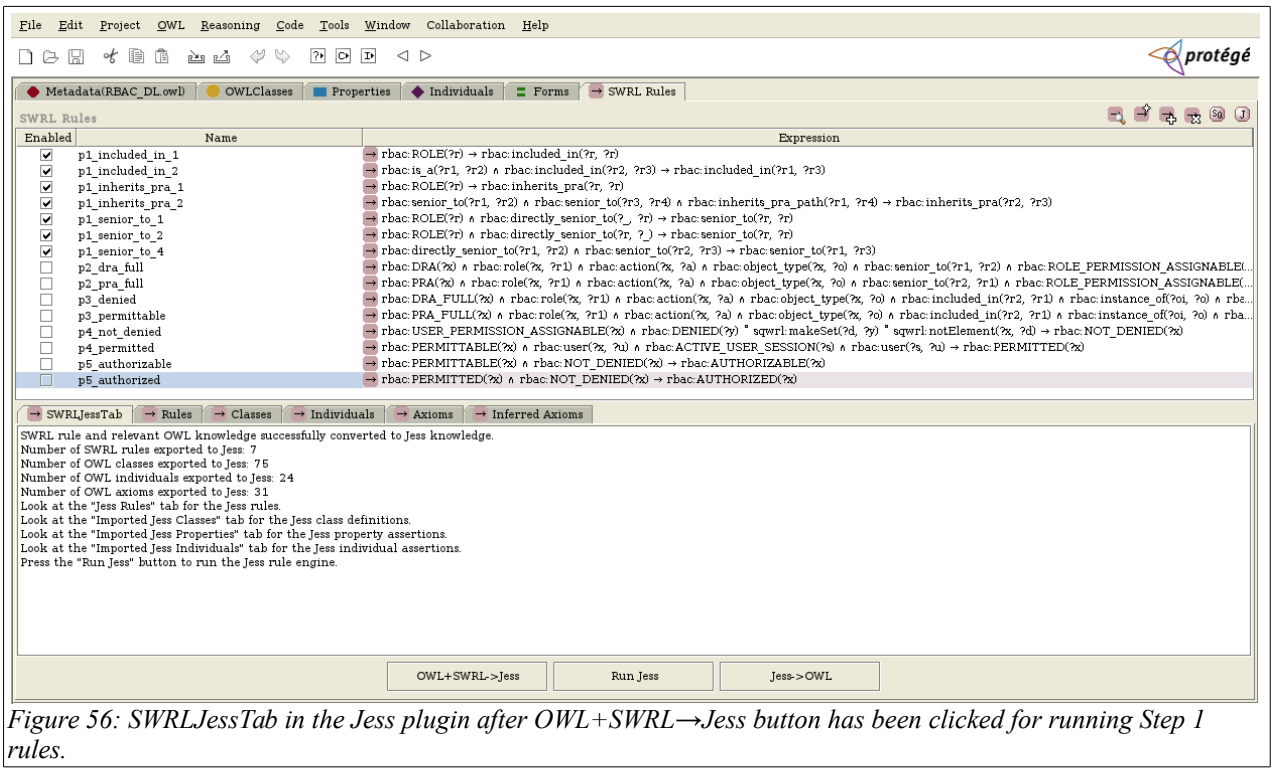

Figure 56: SWRLJessTab in the Jess plugin after OWL+SWRL→Jess button has been clicked for running Step 1 rules.

Figure 56 shows the SWRLJessTab in the Jess plugin after OWL+SWRL→Jess button has been clicked for running Step 1 rules. This screen shows the numbers of SWRL rules and OWL classes, individuals and axioms exported to Jess.

5.8 Conclusion

In this chapter we have created and tested SO-RBAC, which is an RBAC ontological model and process written in OWL-DL and SWRL, based on earlier RBAC access models written in predicate logic. Although OWL-DL is easy to understand and widely supported, it does not allow a full exploitation of the power of the Semantic Web. Therefore, SO-RBAC closely follows the semantics of the original Prolog-based RBAC model, and therefore retains many of the drawbacks and complexity of this model. In particular, it was not possible to make full use of the hierarchical nature of

object classes in SO-RBAC, as doing this requires classes to be linked to each other via properties. This requires classes to be treated as individuals, which is not possible in OWL-DL. Therefore, the predicates defining an is-a hierarchy of roles in the Prolog implementation were directly imported into SO-RBAC as properties linking objects, when a representation in terms of the OWL class hierarchy would be more naturally suited to this definition. Another apparent failure of OWL was that defining a property as transitive did not work. Transitivity should eliminate the need to define recursive rules in most cases. However, this appears to be a failure in Protégé, and a working environment would run this properly.

The main differences between the Prolog/relational model and SO-RBAC follow from an important difference between predicate logic and OWL, namely that OWL is monotonic. This has two major implications for modelling in SO-RBAC.

The first is in the handling of negation. Predicate logic uses a 'closed world' assumption, in which a fact that is not explicitly defined and cannot be inferred from other facts is assumed to be false. This makes negation a very straightforward operation. However, OWL uses an 'open world assumption', in which a fact has to be explicitly defined as false. Therefore, it was necessary to use a complex series of functions to simulate negation in SO-RBAC, and this process was very time consuming due to the exponentially large number of individuals that have to be compared to each other.

Additionally, predicate logic can run a rule on a dataset, and automatically returns all axioms that apply to it based on the stored facts. In contrast, OWL can only move individuals that already exist. Where possible, facts are defined in SO-RBAC using object properties, but this is only possible for binary relationships. Other relationships need to be defined using individuals held in classes, and it is necessary to define an individual for every potential relationship. This again results in a time-consuming reasoning process due to the need to compare large numbers of individuals.

Although dynamic RBAC could be implemented using SO-RBAC, we have not attempted to do so. The purpose of SO-RBAC is to prove the feasibility of the principle of building an RBAC model based on the Semantic Web. The test results indicate that SO-RBAC successfully does this, producing results that are consistent with the equivalent model written in predicate logic. The next chapter will consider a purely ontological RBAC model that uses OWL-Full, and thus fully exploits the power of the Semantic Web in reasoning, giving many advantages over SO-RBAC.

6 The Proposal (Continued): Enhanced Semantic and Ontology-based RBAC (ESO-RBAC)

6.1 Introduction

This chapter discusses **E**nhanced **S**emantic and **O**ntology-based **R**ole-**B**ased **A**ccess **C**ontrol (ESO-RBAC), which models roles as classes, so that RBAC role hierarchies can be represented naturally using ontological class hierarchies.

The ESO-RBAC ontology uses OWL-Full. Reasoning is performed using Jena [118], an open source Semantic Web framework for Java [119], which supports OWL. Jena is used because SWRL cannot handle certain aspects of OWL-Full semantics used in ESO-RBAC, such as class-individual duality. [120]

Most previous ontologies for access control have used OWL-DL. Although this is widely supported and easy to understand, it was found to be inflexible. ESO-RBAC uses OWL Full so that classes, as well as instances, can be used as the Domain and Range of properties. This increases flexibility in defining properties, and allows the use of OWL's native class hierarchy in defining roles in an object-oriented fashion. Therefore, roles need to be defined as classes, not as individuals. However, some properties in the ontology take roles as their domains and/or ranges. Unlike OWL-DL, OWL Full permits the use of classes as property parameters, allowing properties to be defined this way.

The definition of "roles" as classes also allows users to be defined directly as instances of their roles. Since some roles have role-specific properties (e.g. only subclasses of DOCTOR would have a *consults* property), this allows users to be defined with precisely the properties they need (all users have USER properties, but only doctors have DOCTOR properties). Note also that the USER class is multiply inherited: it is a subclass not only of RBAC, but also of PERSON (which is a subclass of data). Another subclass of PERSON is PATIENT. Some properties (those relating to personal details) apply to any PERSON, but USER and PATIENT classes also have specific properties. Additionally, permissions may be defined at any level in the RBAC class hierarchy. That is, a permission or context constraint might apply to any DOCTOR or SPECIALIST role, or it might apply specifically to SPECIALIST_DOCTOR.

Section 6.2 demonstrates the ESO-RBAC ontological model and reasoning. The section is divided into three subsections.

Section 6.2.1 defines the ESO-RBAC ontological model through three distinctive steps:

(a) Definition of OWL classes and their hierarchies

(b) Definition of Necessary & Sufficient conditions and

(c) Definitions of object properties.

Step (a) above is not sufficient to mirror the semantic of RBAC within OWL, *i.e.* the semantics stored in OWL class hierarchies must be strengthen through object properties and necessary and sufficient conditions to achieve both, successful reasoning upon OWL individuals and consistency of ESO-RBAC ontology.

Section 6.2.2 describes the way of populating ESO-RBAC classes with individuals by assertion. That section explains exactly which classes must be populated before the reasoning process starts and why. Consequently, a portion

of ESO-RBAC ontological classes will remain ‘empty’ until a reasoning process determines which individuals from the asserted classes will be ‘moved’ (or copied) into ESO-RBAC classes which were empty on ESO-RBAC initialisation.

Section 6.2.3 explains the purpose and the outcome of the reasoning process upon ESO-RBAC concepts using Jena. ESO-RBAC has two types of reasoning. The first reasoning step, in described in 6.2.3.1, uses Jena for creating a set of new object properties which use existing object properties defined in step (c). All of the object properties for which this is done have ROLE_SET meta-class as both domain and range, as the purpose of this step is to set up all the relationships between roles in the RBAC model. The second step, described in Section 6.2.3.2, performs reasoning to move individuals across ESO-RBAC in order to determine permission or denials in particular request imposed by a user, who has a ‘role’ and would like to perform an ‘activity’ upon set of “objects”.

Section 6.3 describes the ESO-RBAC process and explains its steps, which are based on the model and reasoning introduced in Section 6.2.

Section 6.4 describes the reasoning rules used for running dynamic RBAC in the ESO-RBAC model.

Section 6.5 contrasts the proposed ESO-RBAC solution with the SO-RBAC model described in the previous chapter.

Section 6.6 gives a particular scenario of RBAC in terms of defining which individuals may populate one of ESO-RBAC instances. The healthcare domain and a medical database is used to demonstrate the implementation of ESO-RBAC.

Section 6.7 describes the implementation of ESO-RBAC reasoning and the deployment of the ESO-RBAC process. The ESO-RBAC ontology is modelled in OWL-Full using Protégé. The reasoning rules, written in Jena, were run using a Java command-line tool. The model was initialized using a Perl script to create the initial instances.

Section 6.8 shows screen shots from Protégé of the implementation and testing of ESO-RBAC.

Section 6.9 draws conclusions.

6.2 Ontological Model and Reasoning

This section describes the ESO-RBAC model in terms of OWL and Jena.

6.2.1 Definition of ESO-RBAC Ontological Model

6.2.1.1 OWL classes and their hierarchies

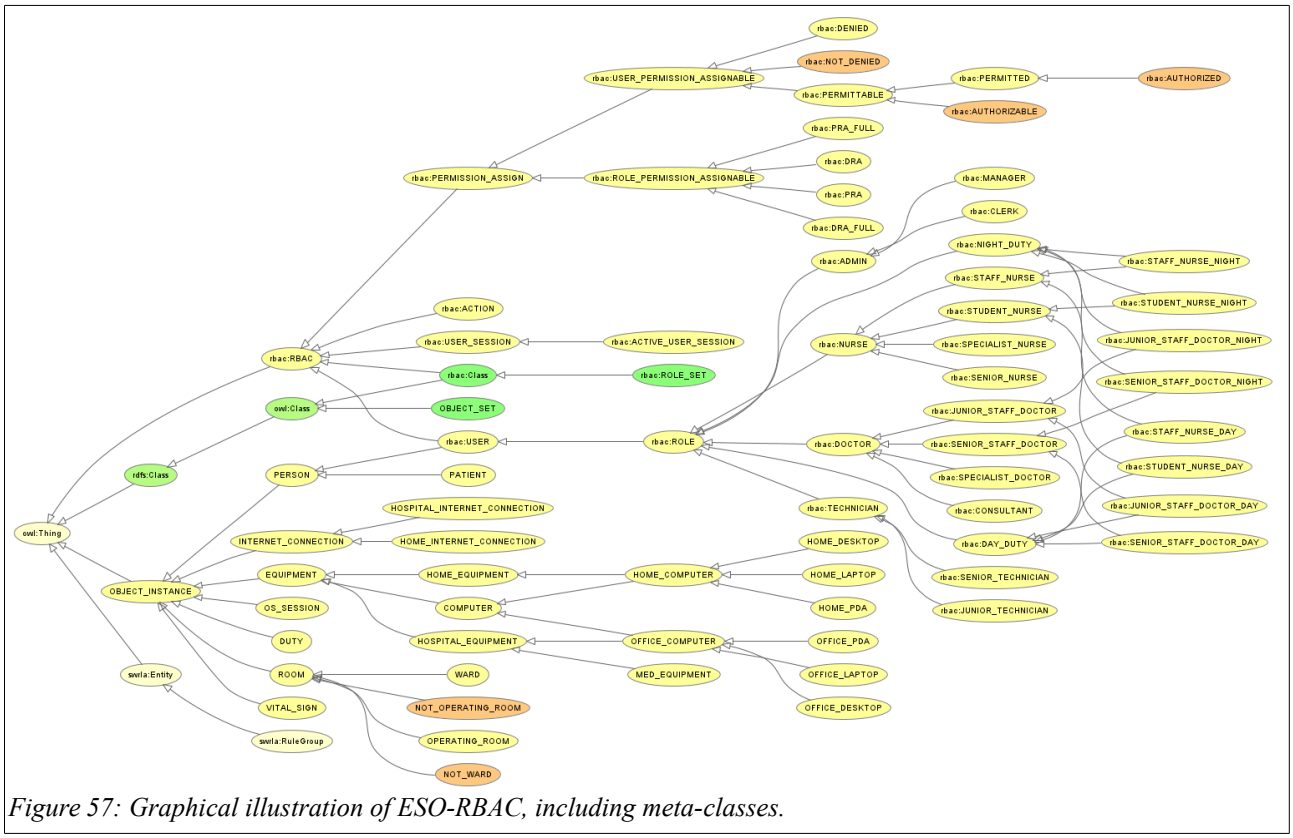

Figure 57: Graphical illustration of ESO-RBAC, including meta-classes.

Figure 57 shows a graphical illustration of ESO-RBAC. At the top level, the ontology is again divided into two abstract super-classes called OBJECT_INSTANCE and RBAC.

Whereas SO-RBAC represents *is-a* relationships using object properties, ESO-RBAC represents them directly using the class hierarchy. For instance, the information represented in Prolog by the fact `is_a(senior_doctor,doctor)` is represented in ESO-RBAC by defining SENIOR_ DOCTOR as a subclass of DOCTOR. Similarly, user-role assignments are handled in ESO-RBAC by directly assigning instances of users to subclasses of ROLE. For example, ESO-RBAC would represent the information corresponding to the Prolog fact `ura(claire,senior_doctor)` by assigning the user instance *Claire* to the role class SENIOR_DOCTOR.

In this ESO-RBAC model, the subclasses in the OBJECT_INSTANCE class are the same as those in SO-RBAC.

The ontology for ESO-RBAC is given in Text 2, with a graphical illustration in Figure 57. Class RBAC has mostly the same sub-classes in ESO-RBAC as in SO-RBAC, but ESO-RBAC has the following differences:

- URA is missing, as its semantics are represented by assigning individuals to sub-classes of ROLE.
- ROLE is modelled as a sub-class of USER.

- OBJECT_TYPE is missing, as object types are represented by sub-classes of OBJECT_INSTANCE. The relationship between object instances and types is represented by assigning individuals representing object instances to object classes.

ROLE identifies all roles. The `is_a` (role inclusion) hierarchy is represented through the ontological class hierarchy. This is because a role defined as an instance of another role is defined as inheriting all the super-role's permissions and denials. This is the natural behaviour of inheritance in a class hierarchy, making it an intuitive way of representing `is_a` relationships.

Thus, as stated earlier, the information represented in Prolog by the fact `is_a(senior_doctor,doctor)` is represented in ESO-RBAC by defining SENIOR_DOCTOR as a subclass of DOCTOR. Inheritance in the seniority hierarchy is different, since permissions and denials are inherited in opposite directions. Therefore, `d_s` and `senior_to` are still represented explicitly by the four object properties *senior_to*, *junior_to*, *directly_senior_to* and *directly_junior_to*.

In an OWL-Full ontology, all classes belong, as individuals, to the meta-class owl:Class. To define a class as a parameter of a property, the domain or range of the property needs to be set to this meta-class. However, ESO-RBAC defines an additional meta-class ROLE_SET, containing ROLE and all of its sub-classes as individuals. This allows properties to be defined that can have only ROLE sub-classes as their domains and ranges. Note that there is no hierarchy in ROLE_SET: all ROLE-class individuals are asserted directly under ROLE_SET.

The super class RBAC defines concepts that are relevant to RBAC, which should be stored in a separate super-class from OBJECT_INSTANCE because it is conceptually different from other information, and is typically stored separately in other systems. For example, a relational DBMS would store the RBAC information as meta-data, which is not usually queried directly by users.

Sub-classes of the OBJECT_INSTANCE class are:

- EQUIPMENT: represents all machines, both computers and medical equipment (and possibly others) to which a user might be logged in. There are various sub-classes of EQUIPMENT, and multiple inheritance is used.
- INTERNET_CONNECTION: represents Internet settings of computers. This class is sub-classed into HOME_INTERNET_CONNECTION and HOSPITAL_INTERNET_CONNECTION.
- OS_SESSION represents operating system login settings of computers.
- PERSON represents all individuals with information stored about them. This includes users, so the class USER is a sub-class of this as well as of RBAC. The other sub-class of PERSON in this example is PATIENT.
- ROOM represents all rooms in a hospital, and is sub-classed into OPERATING_ROOM and WARD.
- VITAL_SIGNS represents vital signs recorded for patients.

Sub-classes of the RBAC class are as follows.

The USER sub-class defines the set of users of the system. However, USER also inherits from PERSON, which is a subclass of the OBJECT_INSTANCE class. On a superficial level, this is because user information might be stored both as ordinary data and as meta-data in a relational database. On a practical level, it is because the USER class, describing a user, contains information about users that is used in either ordinary information-retrieval situations or in RBAC processing, or both.

ROLE sub-class, as discussed above a sub-class of USER in ESO-RBAC, contains a complex hierarchy of sub-classes, defining roles to which users and permissions may be assigned. The hierarchy of classes under ROLE represents sub-divisions of roles by type (not by seniority). The RBAC administrator is free to sub-class this class according to the domain. In this example, it is sub-classed according to roles that might be found in a hospital. The main sub-classes of ROLE in this example are DOCTOR, NURSE, ADMIN, TECHNICIAN, DAY_DUTY and NIGHT_DUTY. These sub-classes are further sub-classed, including multiple inheritance.

USER_SESSION defines user login sessions. Its sub-class ACTIVE_USER_SESSION defines user login sessions that are active, and thus give permissions to users.

ACTION class defines actions that can be performed on objects, such as *read* and *write*.

PERMISSION_ASSIGN is a sub-class consisting of all classes that relate to permission assignments. However, it is also an abstract class in ESO-RBAC, *i.e.* it never contains any instances directly assigned to it. It is defined to provide the *role* and *action* properties to all permission-assignment classes in ESO-RBAC. Its sub-classes are ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE. They define permission assignments between users and objects, and between roles and objects. ROLE_PERMISSION_ASSIGNABLE defines permissions and denials assigned to roles, either explicitly or computationally by ESO-RBAC. USER_PERMISSION_ASSIGNABLE defines permissions, authorizations and denials assigned to users by ESO-RBAC computations.

The sub-classes of USER_PERMISSION_ASSIGNABLE are DENIED, NOT_DENIED, PERMITTABLE, AUTHORIZABLE, PERMITTED and AUTHORIZED. All these sub-classes, except NOT_DENIED, are equivalent to the similarly-named Prolog predicates. NOT_DENIED is the complement of DENIED. PERMITTED is defined as a sub-class of PERMITTABLE, because it can only contain individuals that are also in this.

The sub-classes of ROLE_PERMISSION_ASSIGNABLE are DRA, DRA_FULL, PRA and PRA_FULL, all of which are equivalent to the similarly-named Prolog predicates. PRA defines explicit role-permission assignments. PRA_FULL defines role-permission assignments that are inferred when the ESO-RBAC model is run. Similarly, DRA defines explicit role-denial assignments, and DRA_FULL defines inferred role-denial assignments.

6.2.1.2 Necessary & Sufficient conditions

Table 16: Necessary & Sufficient conditions imposed on ESO-RBAC classes

Class	***Necessary & Sufficient condition***
NOT_DENIED	USER_PERMISSION_ASSIGNABLE ⊓ ¬DENIED
AUTHORIZABLE	PERMITTABLE ⊓ ¬DENIED
AUTHORIZED	PERMITTED ⊓ ¬DENIED

As with SO-RBAC, a few Necessary & Sufficient conditions were imposed on some ESO-RBAC classes in order to guarantee consistency of ESO-RBAC when populating classes with individuals. In other words Necessary & Sufficient conditions are imposed on NOT_DENIED, AUTHORIZABLE and AUTHORIZED (see Table 16). If a class has a Necessary & Sufficient condition imposed on it, then populating the class in a way that violates this condition

makes the ontology inconsistent. The ESO-RBAC reasoning process populates these classes in a way that would always be consistent with the conditions.

In Figure 57 (page 117), the graphical illustration of ESO-RBAC, OWL classes are in yellow, except classes bound by Necessary & Sufficient conditions, which are in amber.

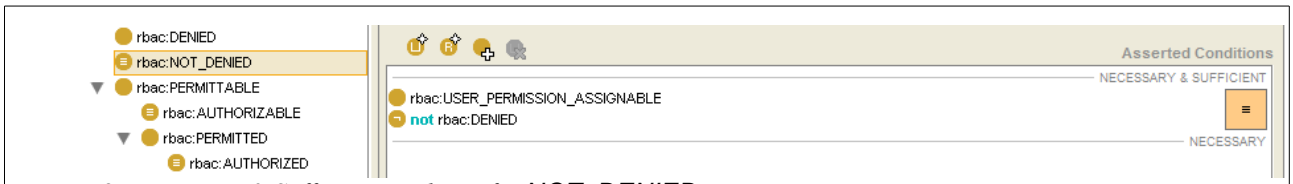

Figure 58: Necessary & Sufficient condition for NOT_DENIED.

Figure 58 shows how a Necessary & Sufficient condition appears in Protégé. As this figure shows, these Necessary & Sufficient conditions cause AUTHORIZABLE to become a sub-class of PERMITTABLE, and AUTHORIZED to become a sub-class of PERMITTED.

6.2.1.3 Object property relationships

We have already mentioned above that PERMISSION_ASSIGN provides the *role* and *action* properties to all permission-assignment classes in ESO-RBAC. Therefore its full description must include object properties it holds.

Naturally, PERMISSION_ASSIGN has object properties *role* and *action*. Just as all permission assignment predicates in the Prolog RBAC model described in Section 3.2.1 have role and action as arguments, so do all analogous classes in ESO-RBAC. However, the hierarchical nature of ontologies makes it much easier to define a series of related classes with the same properties in an ontology than it is to define predicates with similar arguments in Prolog. In OWL, property inheritance can be used to define a super-class with certain properties, and define sub-classes representing related predicates that inherit its object properties. Accordingly, PERMISSION_ASSIGN sub-classes ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE, both inherit properties *role* and *action*, as well as defining other object properties. ROLE_PERMISSION_ASSIGNABLE has the additional property *object_type*. Since DRA, DRA_FULL, PRA and PRA_FULL are sub-classes of ROLE_PERMISSION_ASSIGNABLE, all have the object properties *role*, *action* and *object_type*. *role* and *action*, are inherited from PERMISSION_ASSIGN (their grandparent super-class), while *object_type* is inherited directly from ROLE_PERMISSION_ASSIGNABLE.

USER_PERMISSION_ASSIGNABLE defines permissions, authorizations and denials assigned to users by ESO-RBAC computations. As well as inheriting *role* and *action* from PERMISSION_ASSIGN, it also has the object properties *user* and *object_instance*.

Table 17: Object properties in ESO-RBAC.

Domain	***Property***	***Description***	***Range***
rbac:PERMISSION_ASSIGN (sub-classes: rbac:USER_PERMISSION_ASSIGNABLE and sub-classes, rbac:ROLE_PERMISSION_ASSIGNABLE and sub-classes)	*rbac:action*	Actions involved in role and user permission assignments.	rbac:ACTION
rbac:PERMISSION_ASSIGN and sub-classes	*rbac:role*	Roles involved in role and user permission assignments.	rbac:ROLE_SET

Domain	*Property*	*Description*	*Range*
rbac:USER_PERMISSION_ASSIGNABLE (sub-classes: rbac:DENIED, rbac:NOT_DENIED, rbac:PERMITTABLE, rbac:AUTHORIZABLE, rbac:PERMITTED, rbac:AUTHORIZED)	*rbac:object_instance*	Object instance to which a user is permitted, authorized or denied access.	OBJECT_INSTANCE and sub-classes
rbac:USER_PERMISSION_ASSIGNABLE (rbac:DENIED, rbac:NOT_DENIED, rbac:PERMITTABLE, rbac:AUTHORIZABLE, rbac:PERMITTED, rbac:AUTHORIZED)	*rbac:user*	Users involved in user permission/denial/authorization assignments.	rbac:USER
rbac:ROLE_PERMISSION_ASSIGNABLE (rbac:DRA_FULL, rbac:DRA, rbac:PRA_FULL, rbac:PRA)	*rbac:object_type*	Object types associated with PRA and DRA relationships.	rbac:OBJECT_TYPE
rbac:USER_SESSION	*rbac:user*	A user attached to a session.	rbac:USER
rbac:ROLE_SET	*rbac:directly_junior_to*	Inverse of *directly_senior_to*. Sub-property of *junior_to*.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:directly_senior_to*	Assertions of direct seniority relationships. Sub-property of *senior_to*.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:included_in*	Direct and indirect inclusion relationships, inferred from OBJECT sub-classing.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:inherits_pra*	Roles that participate in inheritance paths, inferred from *inherits_pra_path*.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:inherits_pra_path*	Assertions of ends of inheritance paths.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:junior_to*	Inverse of *senior_to*.	rbac:ROLE_SET
rbac:ROLE_SET	*rbac:senior_to*	Direct and indirect seniority relationships, inferred from *senior_to*.	rbac:ROLE_SET

Table 17 lists ALL object properties with their Domains and Ranges, and includes both asserted and inherited object properties. Most of the same object properties from SO-RBAC are used in ESO-RBAC. The following properties are not used: *rbac:instance_of* (represented by OBJECT_INSTANCE subclass membership) and *is_a* (represented by ROLE sub-classing). Additionally, the class URA is not defined (as its semantics are represented by ROLE sub-class membership). Properties that have ROLE as the domain and range in SO-RBAC have ROLE_SET in ESO-RBAC.

Most object properties are named after their Ranges. These properties may have different functions depending on the Domain: each function of a property is listed separately in the table.

Object properties not named after their Ranges are the properties that have ROLE_SET as both Domain and Range (*directly_junior_to*, *directly senior_to*, *included_in*, *inherits_pra*, *inherits_pra_path*, *junior_to*, *senior_to*).

It is important to note that some object properties from Table 17 are *asserted* and some of them are *inferred*. For example, the object properties *action*, *user*, *role* and *object_instance* are *asserted* between USER_PERMISSION_ASSIGNABLE and OBJECT_INSTANCE, ACTION, USER and ROLE_SET (and its sub-classes), but *inferred* between all subclasses of USER_PERMISSION_ASSIGNABLE and these classes.

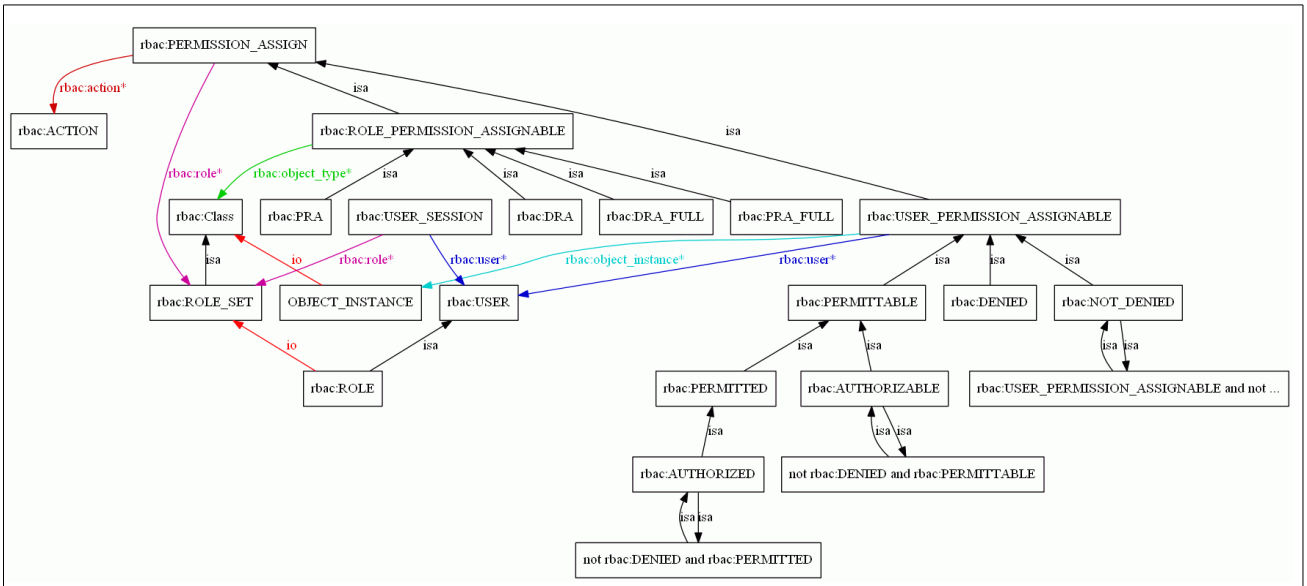

*Figure 59: Property map of all ESO-RBAC properties except those that have **ROLE** as both domain and range.*

```
+ OBJECT_INSTANCE
- RBAC
  - ACTION
  - OBJECT_TYPE
  + ROLE
  = PERMISSION_ASSIGN {action ACTION, role ROLE_SET}
    - ROLE_PERMISSION_ASSIGNABLE {object_type owl:Class}
      - DRA
      = DRA_FULL
      - PRA
      = PRA_FULL
    = USER_PERMISSION_ASSIGNABLE {object_instance OBJECT_INSTANCE, user USER}
      - DENIED
      ≡ NOT_DENIED {USER_PERMISSION_ASSIGNABLE ⊓ ¬DENIED}
      = PERMITTABLE
        ≡ AUTHORIZABLE {PERMITTABLE ⊓ ¬DENIED}
        = PERMITTED
   ≡ AUTHORIZED {PERMITTED ⊓ ¬DENIED}
  - USER
  - USER_SESSION {user USER}
    - ACTIVE_USER_SESSION
  + rdf:Property
  - rdfs:Class
    - owl:Class
      - Class
        - ROLE_SET {directly_junior_to ROLE_SET, directly_senior_to ROLE_SET, included_in
ROLE_SET, inherits_pra ROLE_SET, inherits_pra_path ROLE_SET, is_a ROLE_SET}
```

Text 25: ESO-RBAC Ontology (some classes are collapsed).

Similarly, the object properties ***action***, ***object_type*** and ***role*** are *asserted* between ROLE_PERMISSION_ASSIGNABLE and ACTION, OBJECT_TYPE and ROLE_SET classes, but *inferred* between all subclasses of ROLE_PERMISSION_ASSIGNABLE and these classes.

```
+ COLLAPSED_CLASS
- CLASS
   - SUB-CLASS {object_property_1 CLASS, object_property_2 CLASS}
   - ABSTRACT_CLASS
= SWRL-INFERRED_CLASS
   ≡ N&S_BOUND_CLASS {N&S CONDITION (⊓: and; ¬: not}
```

Text 26: Legend for ESO-RBAC Ontology.

Text 25 (page 122) illustrates a collapsed version of the ontology from Figure 57, and highlights main ESO-RBAC classes involved in ontological reasoning. It is important to note that Text 25 should be read in conjunction with Text 26.

Object properties are listed, with their ranges, in grey text in curly brackets after the classes that have them as their domains. Classes that contain inferred individuals as the result of our ontological reasoning are listed in blue and preceded by the = symbol. Classes on which Necessary & Sufficient conditions are imposed are in green and preceded by the ≡ symbol.

Figure 59 (page 122) graphically illustrates all object properties defined in Table 17, except those that have ROLE as both domain and range. The label 'isa' in Figure 59 refers to the sub-class–super-class relationship: a sub-class 'isa' super-class. It has nothing to do with the *is_a* property used in ESO-RBAC. The label 'io' ('instance of') is used to denote a class that belongs, as an individual, to another class (e.g. from ROLE to ROLE_SET). In this diagram, each property is distinguished by colour: where the same property appears several times, it is shown in the same colour. However, these colours are not used anywhere else.

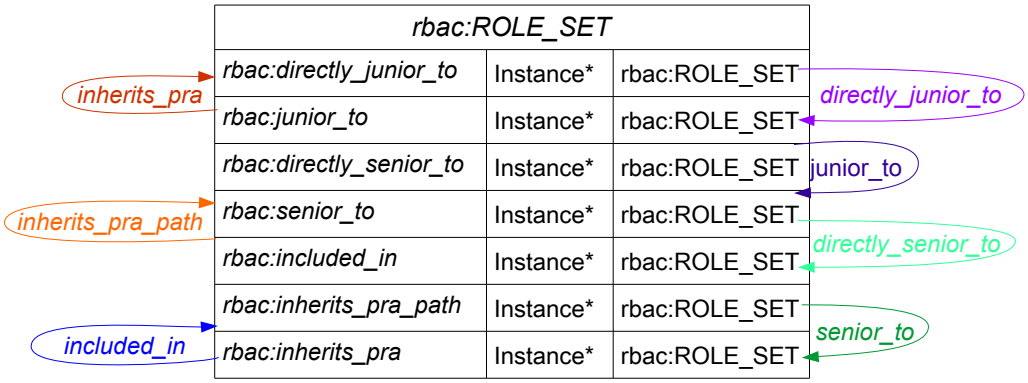

Figure 60: Property map of all ESO-RBAC properties with the meta-class ROLE_SET as both domain and range.

There are several ways in which instances of the meta-class ROLE_SET can be related affecting user-permission assignment in RBAC. These need separate attention.

Figure 60 depicts all object properties in ESO-RBAC that have the meta-class ROLE_SET as both domain and range. These properties are *directly_junior_to*, *directly_senior_to*, *inherits_pra*, *inherits_pra_path*, *junior_to*, *senior_to* and *included_in* (ESO-RBAC does not have *is_a*). These separate properties represent different relationships between ROLE_SET instances (sub-classes of ROLE), as described in Table 20. Each object property relating instances of the same class is indicated by an arrow from the node representing the ROLE class and pointing back to this box. For clarity, these object properties are also listed in the node. The box in Figure 60 signifies that a ROLE class (instance of meta-class ROLE_SET) (represented by the ROLE at the top of the box) has can be linked to any instances of ROLE_SET via any of the properties listed.

Note that all object properties in Figure 60 apply to the same ROLE_SET meta-class. Therefore, they appear in Figure 60 twice: in the first column of the figure and as coloured labels of arcs which graphically illustrate these object properties defined upon meta-class ROLE_SET.

6.2.2 Populating ESO-RBAC classes by assertion

Classes populated in this stage are classified into two types. Note that ROLE and PERMISSION_ASSIGN are abstract classes, which contain no asserted individuals.

i. Auto-populated on initialization: ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE are populated on initialization with individuals representing possible role and user permission assignments. Individuals asserted under these classes are not active: they have to be moved to sub-classes of these classes to be active in ESO-RBAC.

ii. Populated according to RBAC model on initialization: USER, ACTION, ROLE, OBJECT_TYPE, USER_SESSION, ACTIVE_USER_SESSION, DRA, PRA and all classes under ROLE and OBJECT_INSTANCE are populated, by the RBAC administrator and application, with individuals that define the RBAC rules and environment.

ESO-RBAC lacks the OBJECT_TYPE class found in SO-RBAC, since it identifies the type of an object instance by its class.

Classes in the meta-class ROLE_SET specify RBAC roles.

On initialization, it is populated with individuals representing all possible relationships between roles, actions and object types. These are then moved in the reasoning step into any of the sub-classes.

The sub-classes of ROLE_PERMISSION_ASSIGNABLE are DRA, DRA_FULL, PRA and PRA_FULL. DRA and PRA are populated on initialization with explicit denial and permission assertions, respectively, copied from the super-class. They are exactly equivalent to `dra` and `pra` assertions in the Prolog model.

USER_PERMISSION_ASSIGNABLE is populated on initialization with individuals representing all possible combinations of user assignments of access to perform actions on object instances.

The sub-classes of ROLE are populated by individuals representing users who are assigned the roles that they represent. In the example RBAC, ROLE is an abstract class, and has a hierarchy below this indicating types of role such as DOCTOR and NURSE. Unlike in SO-RBAC, ESO-RBAC directly uses the position of a sub-class of ROLE in the class hierarchy to represent role inclusion, eliminating the need for the property *is_a*.

The USER_SESSION class is populated with user login sessions. It contains sub-class ACTIVE_USER_SESSION, which represents active user sessions.

6.2.3 Reasoning in ESO-RBAC using Jena

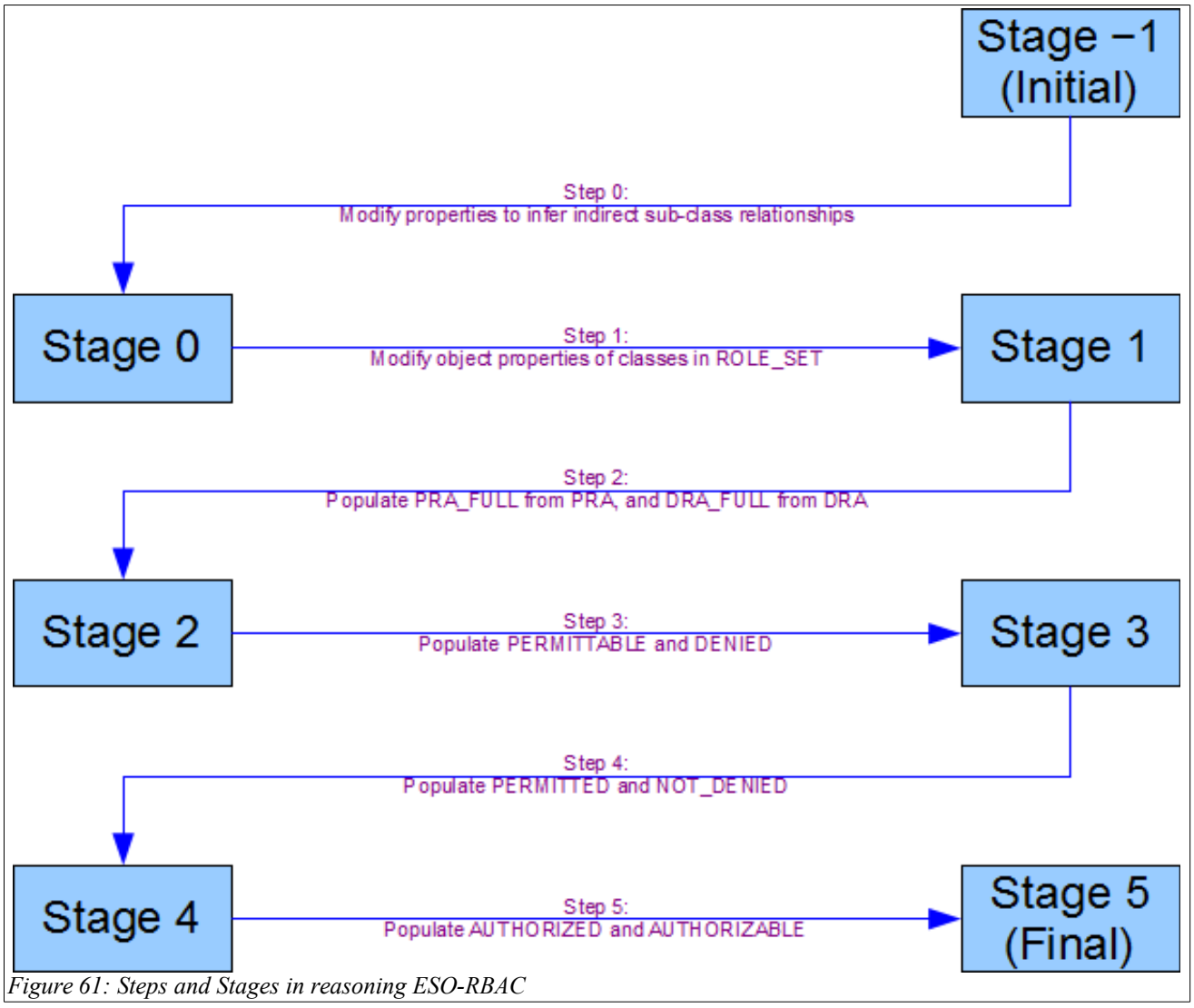

Figure 61: Steps and Stages in reasoning ESO-RBAC

Figure 61 shows the six steps of reasoning, which in terms of overview are the same as in SO-RBAC, except for the addition of Step 0. Step 1 significantly differs from the others because it uses Jena for inferring more object properties. In other words, Step 1 modifies object properties in sub-classes of ROLE (as instances of ROLE_SET) for the purpose of determining all the relationships between roles within RBAC.

A new step, Step 0, is added to the beginning of the reasoning process to run reasoning rules that infer indirect sub-classes and class membership. This is because Jena does not infer these on its own. Step 0 would not be required when using a fully-functioning OWL-Full reasoner.

Steps 2–5 of the reasoning process infer individuals in ESO-RBAC classes according to strictly defined matching of SO-RBAC sub-classes. The final result of our reasoning through Jena and ontological matching will be shown in Stage 5, when certain individuals will be moved into ESO-RBAC classes AUTHORIZABLE and AUTHORIZED.

Steps 0 and 1 are shown in Sub-section 6.2.3.1, and Steps 2–5 are shown in Sub-section 6.2.3.2.

The steps are designed such that each stage populated the ontology with all axioms that may be required for the immediately following stage (except that Step 1 creates all object property relationships).

The reader should be aware that if more than one rule affects the same class or property, then the relationship between the rules is a logical OR. Although the syntax of Jena (unlike that of SWRL) does allow representation of logical OR relationships in a single rule, it was decided to use separate rules in ESO-RBAC, to maintain the link with the SRWL rules used in SO-RBAC.

The Jena rules were named according to the following conventions:

- The rules are numbered according to the step in which they are executed when rule chaining. There are six steps, 0–5 (Figure 61).

The Jena rules are named according to the convention `s_relation[_n]`, where `s` is the step number, `relation` is the class or property affected by the rule, and `n` is a sequence number (if there is more than one rule relating to the same relation in the same stage).

Jena can be written either in the standard Prolog syntax, as below, with the consequent at the head (as in Prolog), or with the consequent at the tail (as in SWRL). In ESO-RBAC, they are written with the consequent at the head.

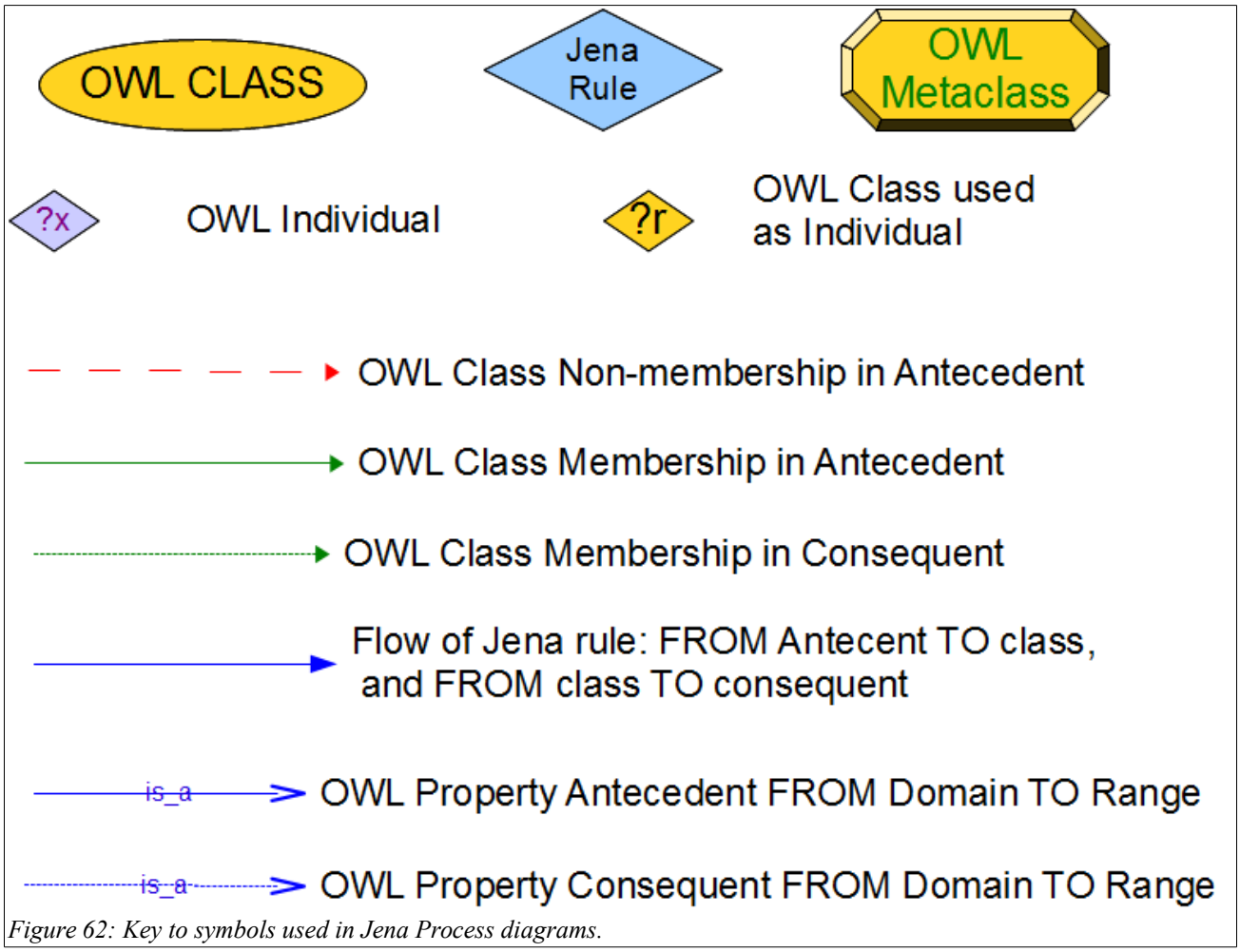

Figure 62: Key to symbols used in Jena Process diagrams.

Figure 62 shows the key to the symbols used in the diagrams in Figures 63–93 showing the inference processes for object properties.

6.2.3.1 Defining new object properties

We define new object properties in Steps 0 and 1, from Figure 62. These definitions are based on previously defined object properties, where the ROLE_SET meta-class is the Range and Domain. Step 0 consists of 4 Jena rules, named as `0_inferred_subClassOf_1`, `0_inferred_subClassOf_2`, `0_inferred_type_1` and `0_inferred_type_2`. Step 1 consists of 8 Jena rules, named as `1_senior_to_1`, `1_senior_to_2`, `1_senior_to_4`, `1_junior_to_1`, `1_junior_to_2`, `1_junior_to_4`, `1_inherits_pra_1` and `1_inherits_pra_3`. `1_junior_to_1`, `1_junior_to_2` and `1_junior_to_4`, are the inverse rules to `1_senior_to_1`, `1_senior_to_2` and `1_senior_to_4`, respectively.

```
[inferred_subClassOf_1: (?c1
rdfs:inferred_subClassOf ?c2)
            <-
            (?c1 rdfs:subClassOf ?c2)
]
```

Text 27: Jena for rule `0_inferred_subClassOf_1`.

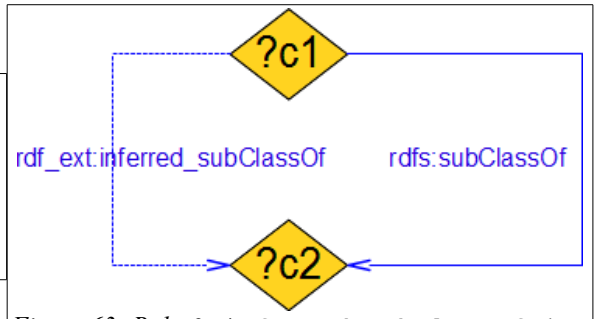

Figure 63: Rule `0_inferred_subClassOf_1`.

The first rule in Step 0, given in Figure 63, is called `0_inferred_subClassOf_1`. It defines a class as being an inferred sub-class if it is a sub-class of that class. Figure 63 is converted into Jena syntax in Text 27 above.

```
[inferred_subClassOf_2: (?c1
rdfs:inferred_subClassOf ?c3)
            <-
            (?c1 rdfs:subClassOf ?c2)
            (?c2 rdf_ext:inferred_subClassOf ?c3)
]
```

Text 28: Jena for rule `0_inferred_subClassOf_2`.

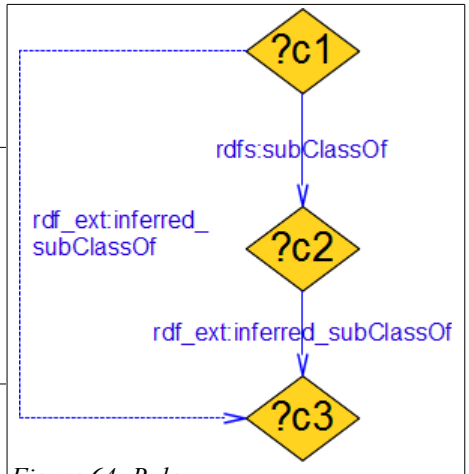

Figure 64: Rule `0_inferred_subClassOf_2`.

The second rule in Step 0, given in Figure 64, is called `0_inferred_subClassOf_2`. It defines inferred sub-classing as transitive. In other words, class *?c1* is an inferred sub-class of class *?c3* if it is a direct sub-class of another class (*?c2*) that is an inferred sub-class of class *?r3*.

Figure 64 is converted into Jena syntax in Text 28 above.

```
[inferred_type_1: (?i
rdf_ext:inferred_type ?c)
            <-
            (?i rdf:type ?c)
]
```

Text 29: Jena for rule `0_inferred_type_1`.

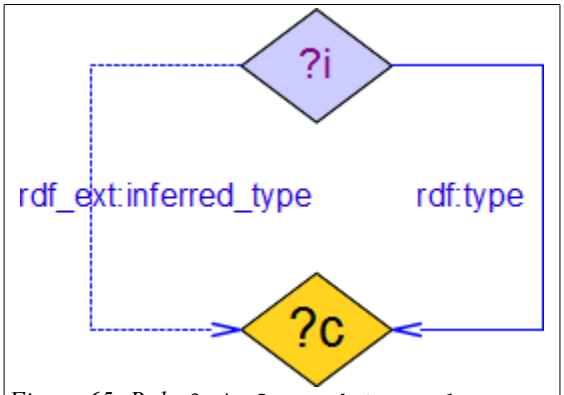

Figure 65: Rule `0_inferred_type_1`

The third rule in Step 0, given in Figure 65, is called `0_inferred_type_1`. It an individual as being an inferred type of a class if it is a type (member) of this class. Figure 65 is converted into Jena syntax in Text 29 above.

```
[inferred_type_2: (?i rdf_ext:inferred_type ?c)
            <-
            (?c1 rdf_ext:inferred_subClassOf ?c)
            (?i rdf:type ?c1)
]
```

Text 30: Jena for rule `0_inferred_type_2`.

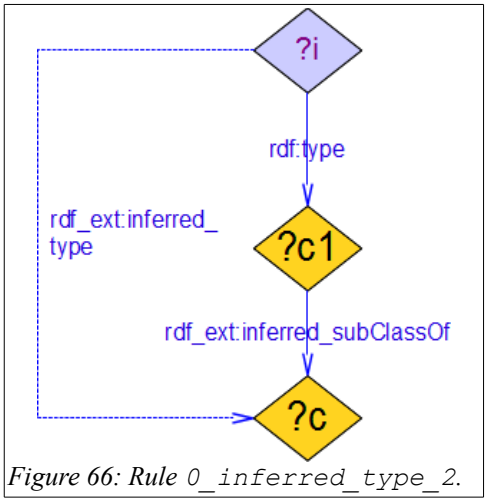

Figure 66: Rule `0_inferred_type_2`.

The fourth rule in Step 0, given in Figure 66, is called `0_inferred_type_2`. It defines an individual as an inferred type of a class if it is a member of an inferred sub-class of this class. In other words, individual *?i* is an inferred type of class *?c* if it is a member of class *?c1*, which is an inferred sub-class of class *?c*.

Figure 66 is converted into Jena syntax in Text 30 above.

```
[1_senior_to_1: (?r rbac:senior_to ?r)
    <-
   (?r rdf:type rbac:ROLE_SET)
   (?r rbac:directly_senior_to ?r1)
]
```

Text 31: Jena for rule `1_senior_to_1`.

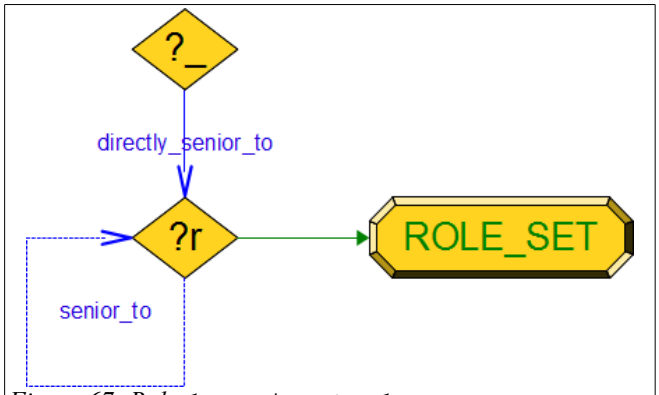

Figure 67: Rule `1_senior_to_1`

The first rule in Step 1, given in Figure 67, is called `1_senior_to_1`. It defines a role as is senior to itself if it has at least one role directly senior to it. Figure 67 is converted into Jena syntax in Text 31 above.

```
[1_senior_to_2: (?r rbac:senior_to ?r)
    <-
   (?r rdf:type rbac:ROLE_SET)
   (?r1 rbac:directly_senior_to ?r)
]
```

Text 32: Jena for rule `1_senior_to_2`.

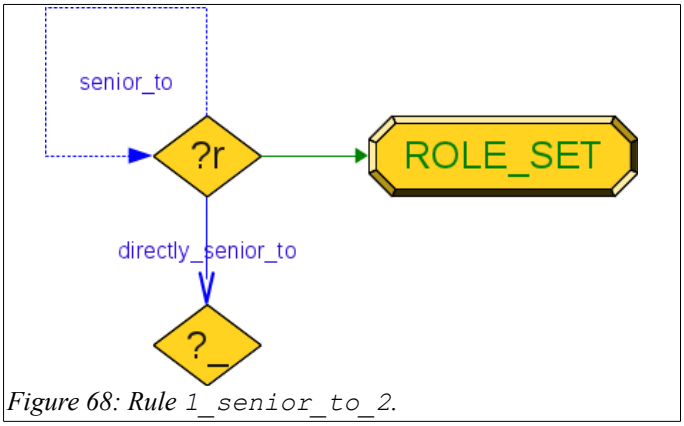

Figure 68: Rule `1_senior_to_2`.

The second rule in Step 1, given in Figure 68, is called `1_senior_to_2`. It defines a role as senior to itself if it is directly senior to at least one role. A role class is identified by its membership of the meta-class ROLE_SET. The syntax `?r rdf:type rbac:ROLE_SET` in Jena is equivalent to rbac:ROLE_SET(?r) in SWRL. In other words, class membership is queried in Jena by querying the RDF property *rdf:type*.

Figure 68 is converted into Jena syntax in Text 32 above.

```
[1_senior_to_4: (?r1 rbac:senior_to ?r3)
    <-
   (?r1 rdf:type rbac:ROLE_SET)
   (?r1 rbac:directly_senior_to ?r2)
   (?r2 rbac:senior_to ?r3)
]
```

Text 33: Jena for rule 1_senior_to_4.

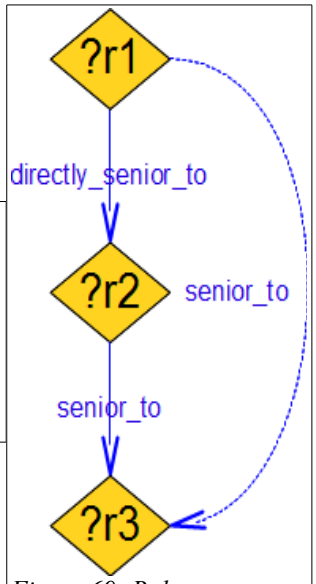

Figure 69: Rule 1_senior_to_4

The third rule in Step 1, given in Figure 69, is called `1_senior_to_4`. It defines a seniority of roles as being transitive. In other words, Role *?r1* is senior to *?r3* if it is directly senior to another role (*?r2*) that is senior to *?r3*. Figure 69 is converted into Jena syntax in Text 33 above.

```
[1_junior_to: (?r1 rbac:junior_to ?r2)
           <-
          (?r rdf:type rbac:ROLE_SET)
          (?r2 rbac:senior_to ?r1)
]
```

Text 34: Jena for rule 1_junior_to.

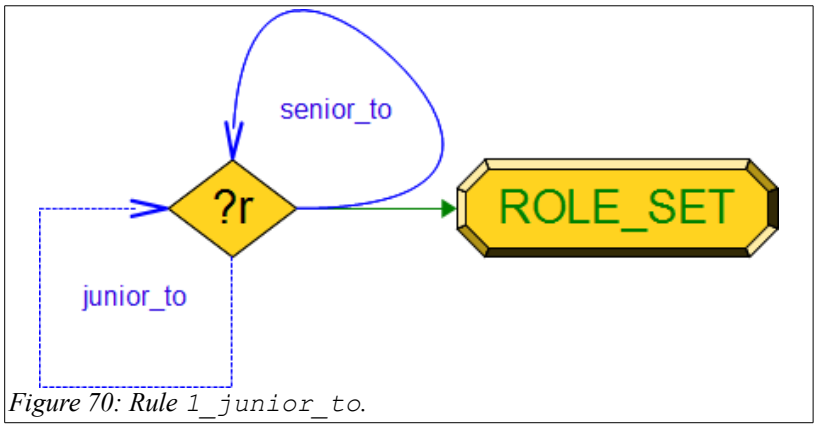

Figure 70: Rule 1_junior_to.

The fourth rule in Step 1, given in Figure 70, is called `1_junior_to`. It defines axioms for *junior_to* as the inverses of *senior_to* axioms: if role *?r2* is senior to *?r1*, then *?r1* is junior to *?r2*. This rule is defined because Jena cannot infer inverse axioms from the axioms that it has inferred, even where properties are defined in the ontology as being inverses of each other.

Figure 70 is converted into Jena syntax in Text 34 above.

The fifth rule in Step 1, given in Figure 71, is called `1_inherits_pra_1`. It defines a role as being part of an inheritance path involving itself. An inheritance path is a path along which permissions can be inherited. This rule is necessary to set up recursion when defining inheritance paths. Figure 71 is converted into Jena syntax in Text 35 below.

```
[1_inherits_pra_1: (?r
rbac:inherits_pra ?r)
    <-
    (?r rdf:type rbac:ROLE_SET)
    notEqual(?r, rbac:ROLE)
]
```

Text 35: Jena for rule `1_inherits_pra_1`.

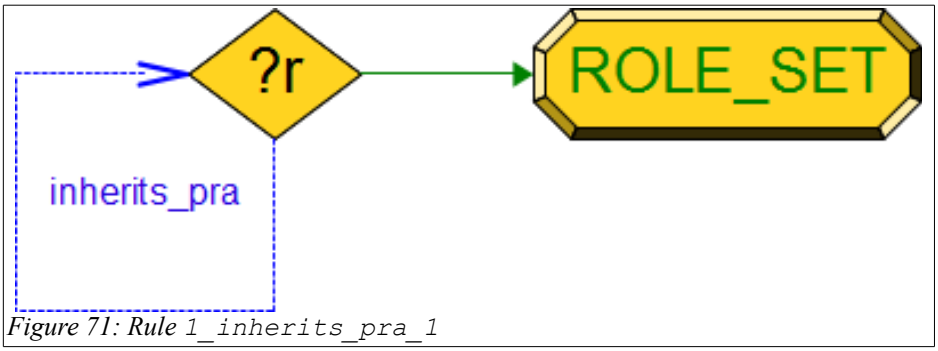

Figure 71: Rule `1_inherits_pra_1`

```
[1_inherits_pra_3: (?r2 rbac:inherits_pra ?r3)
    <-
     (?r1 rbac:senior_to ?r2)
     (?r2 rbac:senior_to ?r3)
     (?r3 rbac:senior_to ?r4)
     (?r1 rbac:inherits_pra_path ?r4)
]
```

Text 36: Jena for rule `1_inherits_pra_3`.

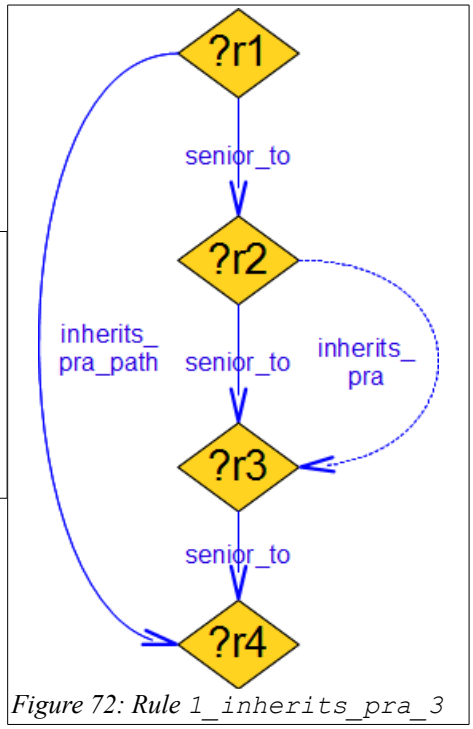

Figure 72: Rule `1_inherits_pra_3`

The sixth rule in Step 1, given in Figure 72, is called `1_inherits_pra_3`. It defines that Roles *?r2* and *?r3* are in an inheritance path, where *?r3* is the senior role, if:

iv) *?r2* has a senior role *?r1* that is at the senior end of an inheritance path, and

v) *?r3* is senior to role *?r4* that is at the junior end of an inheritance path

Figure 72 is converted into Jena syntax in Text 36 above.

6.2.3.2 Moving individuals across ESO-RBAC classes

Individuals are moved across ESO-RBAC classes according to the reasoning performed in Steps 2–5. All rules in Steps 2 and 3, and rule 2 of Step 4, match individuals according to object properties. Rule 1 of Step 4, and both rules in Step 5, match individuals by a simple set operation (set difference or intersection).

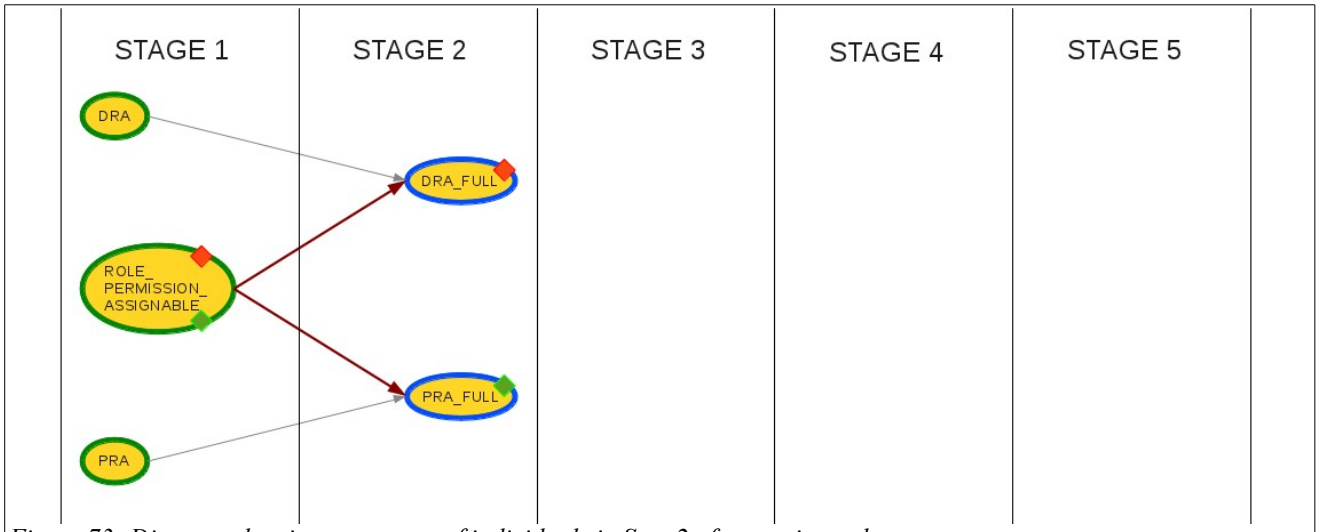

Figure 73: Diagram showing movement of individuals in Step 2 of reasoning only.

Step 2 is shown in Figure 73. It takes class ROLE_PERMISSION_ASSIGNABLE and matches its individuals with individuals of classes PRA and DRA. If individuals from ROLE_PERMISSION_ASSIGNABLE satisfy the rules for their matching, then they are moved to PRA_FULL and DRA_FULL. It is important to note that only individuals from ROLE_PERMISSION_ASSIGNABLE are being moved into PRA_FULL and DRA_FULL, according to the object properties of these and of the individuals in PRA and DRA.

```
[2_dra_full: (?z rdf:type rbac:DRA_FULL )
    <-
    (?z rdf:type rbac:ROLE_PERMISSION_ASSIGNABLE )
    (?z rbac:role ?r2 )
    (?z rbac:action ?a )
    (?z rbac:object_type ?o )
    (?r1 rbac:senior_to ?r2 )
    (?x rdf:type rbac:DRA )
    (?x rbac:role ?r1 )
    (?x rbac:action ?a )
    (?x rbac:object_type ?o )
]
```

Text 37: Jena for rule `2_dra_full`

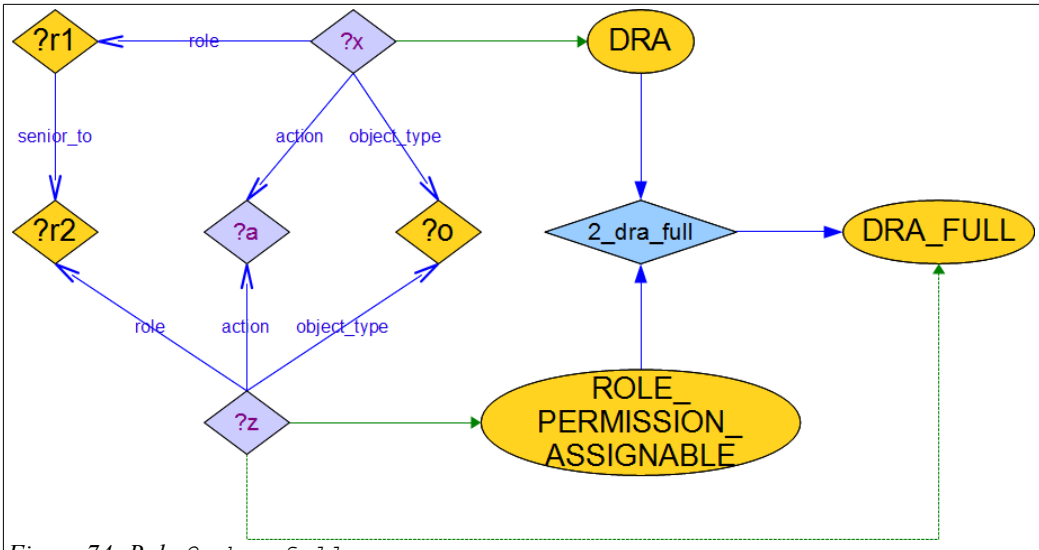

Figure 74: Rule `2_dra_full`

The first rule in Step 2, given in Figure 74, is called `2_dra_full`. This rule moves an individual from ROLE_PERMISSION_ASSIGNABLE to DRA_FULL if there exists an individual in DRA that has the same *action* and *object_type* properties as that in ROLE_PERMISSION_ASSIGNABLE, and if the role property of the individual in DRA is senior to that of the individual in ROLE_PERMISSION_ASSIGNABLE.

A formal description of the matching in rule `2_dra_full` is below. ROLE_PERMISSION_ASSIGNABLE instance *?z* represents a potential user role assignment with the following properties:

- *rbac:action ?a*;
- *rbac:role ?r2*, and
- *rbac:object_type ?o*.

?z is moved to DRA_FULL if:

i) *?z* is linked by object property *rbac:role* to *?r2*;

ii) *?r1* is senior to *?r2* (is linked to *?r1* via object property *rbac:senior_to*);

iii)DRA instance *?x* is linked by object property *rbac:role* to *?r1*, and

iv)both *?z* and *?x* have *rbac:action ?a* and *rbac:object_type ?o*.

Figure 74 is converted into Jena syntax in Text 37 above.

```
[2_pra_full: (?z rdf:type rbac:PRA_FULL )
    <-
    (?z rdf:type rbac:ROLE_PERMISSION_ASSIGNABLE )
    (?z rbac:role ?r2)
    (?z rbac:action ?a )
    (?z rbac:object_type ?o )
    (?r2 rbac:senior_to ?r1 )
    (?r2 rbac:inherits_pra ?r1)
    (?x rdf:type rbac:PRA )
    (?x rbac:role ?r1 )
    (?x rbac:action ?a )
    (?x rbac:object_type ?o )
]
```

Text 38: Jena for rule `2_pra_full`.

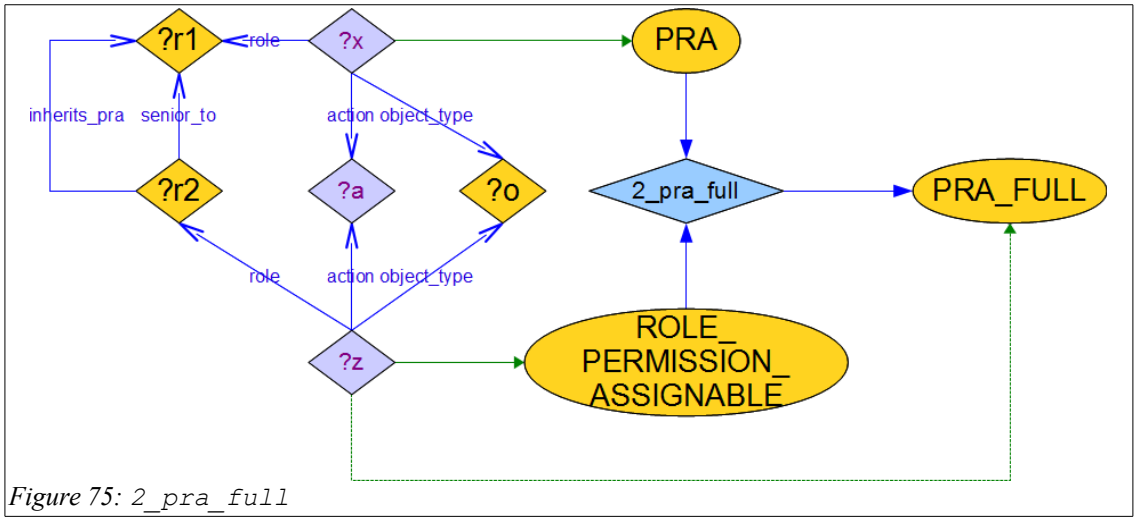

Figure 75: `2_pra_full`

The second rule in Step 2, given in Figure 75, is called `2_pra_full`. This rule moves an individual from ROLE_PERMISSION_ASSIGNABLE to PRA_FULL if there exists an individual in PRA that has the same *action* and *object_type* properties as that in ROLE_PERMISSION_ASSIGNABLE, and if the *role* property of the individual in PRA is junior to that of the individual in ROLE_PERMISSION_ASSIGNABLE.

A formal description of the matching in rule in `2_dra_full` is given below. ROLE_PERMISSION_ ASSIGNABLE instance ?z represents a potential user-role assignment with the following properties:

- *rbac:action ?a*;
- *rbac:role ?r2*, and
- *rbac:object_type ?o*

?z is moved to PRA_FULL if:

i) *?z* is linked by object property *rbac:role* to *?r2*;

ii) *?r2* is senior to *?r1* (is linked to *?r1* via object property *rbac:senior_to*);

iii)*?r2* and *?r1* are in an inheritance path (linked via object property *rbac:inherits_pra*);

iv)PRA instance *?x* is linked by object property *rbac:role* to *?r1*, and

v) both *?z* and *?x* have *rbac:action ?a* and *rbac:object_type ?o*.

Figure 75 is converted into Jena syntax in Text 38 above.

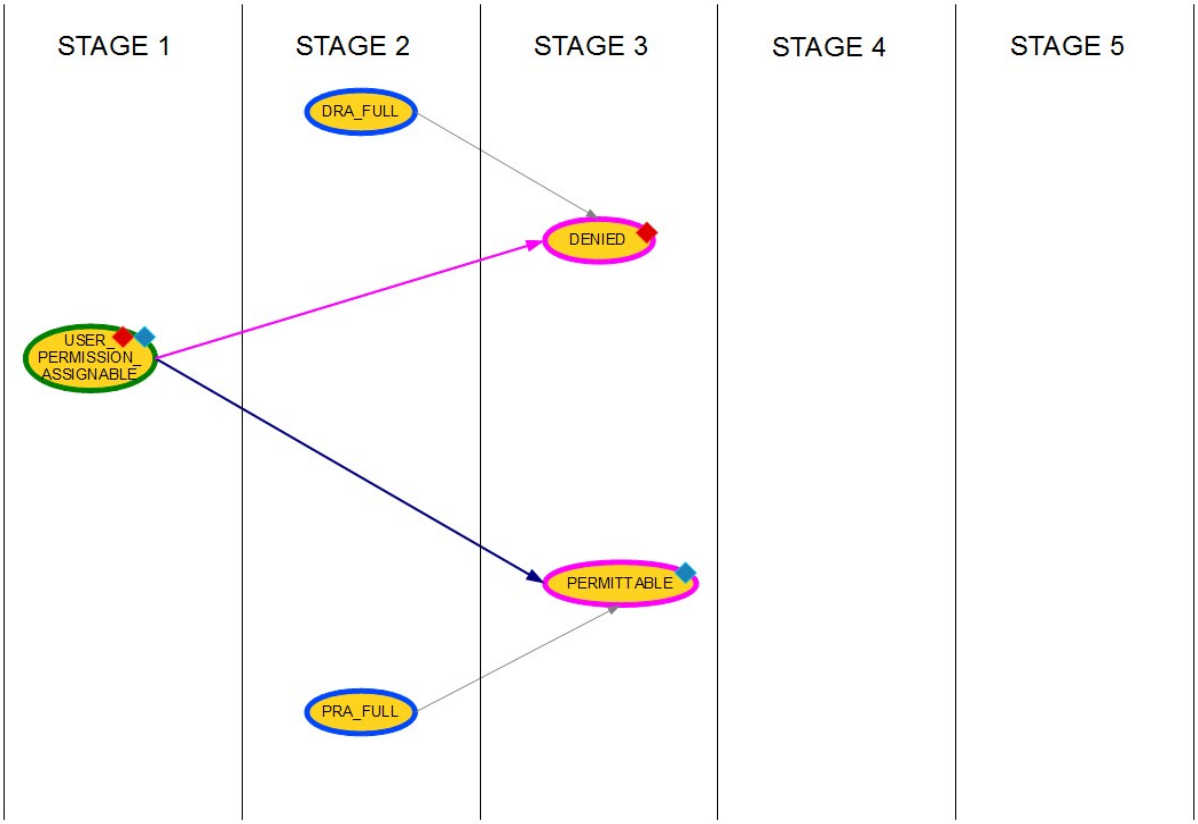

Figure 76: Diagram showing movement of individuals in Step 3 of reasoning only.

Figure 76 shows the movement of individuals in Step 3, in which PERMITTABLE and DENIED are populated from individuals in USER_PERMISSION_ASSIGNABLE, as determined by individuals in PRA_FULL and DRA_FULL, as well as relationships between roles defined by *included_in* axioms.

```
[3_permittable: (?z rdf:type rbac:PERMITTABLE )
            <-
            (?z rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
            (?x rdf:type rbac:PRA_FULL )
            (?x rbac:role ?r )
            (?x rbac:action ?a )
            (?x rbac:object_type ?o )
            (?z rbac:user ?u )
            (?z rbac:role ?r )
            (?z rbac:action ?a )
            (?z rbac:object_instance ?oi )
            (?oi rdf:type ?o )
            (?u rdf_ext:inferred_type ?r )
]
```

Text 39: Jena for rule `3_permittable`.

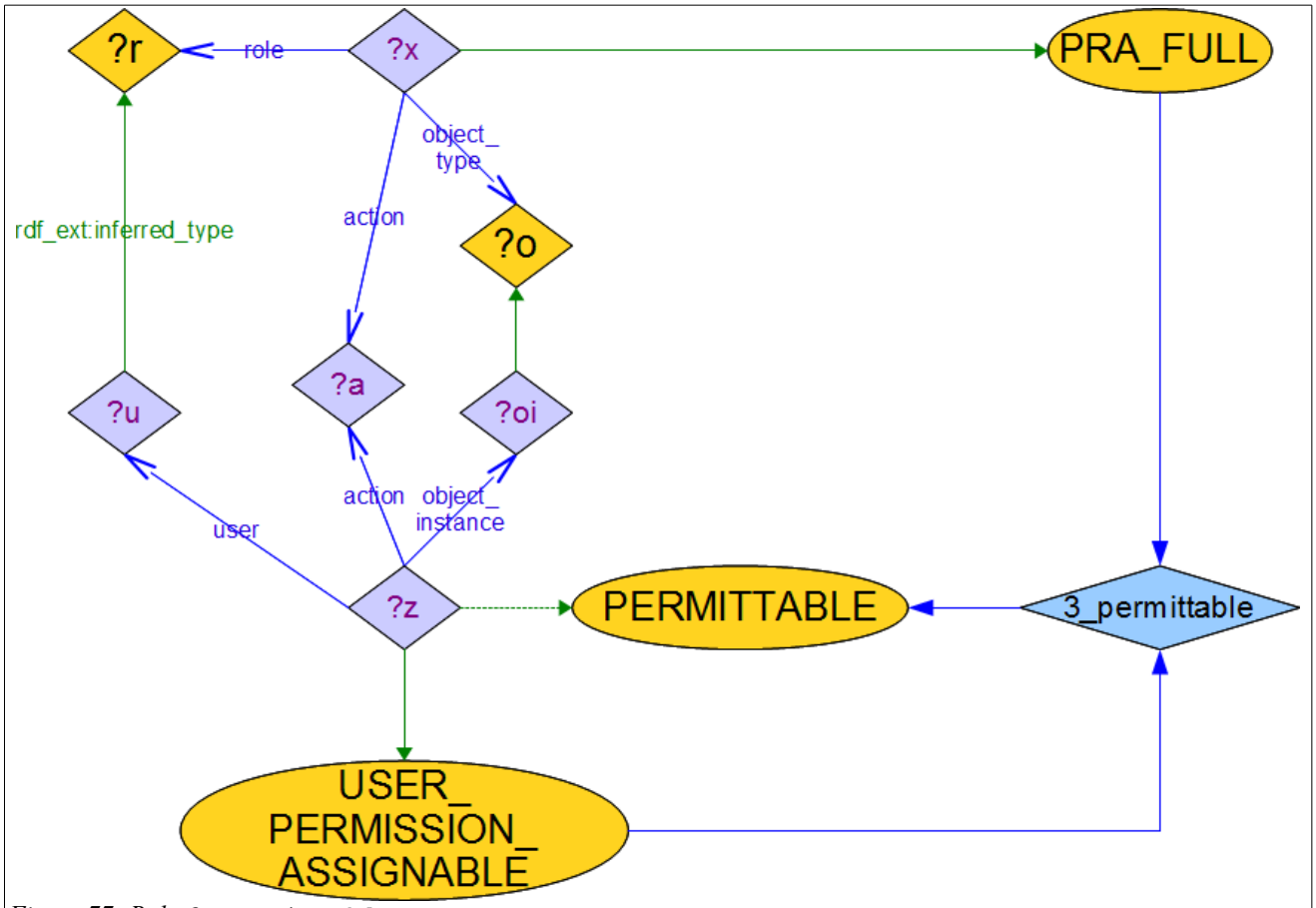

Figure 77: Rule `3_permittable`.

The first rule in Step 3, given in Figure 77, is called `3_permittable`. This rule moves an individual from USER_PERMISSION_ASSIGNABLE to PERMITTABLE if that individual is found to represent an actual user-permission assignment in the RBAC model. That is, if an individual in USER_PERMISSION_ASSIGNABLE has the same action as an individual in PRA_FULL; has object instance that is linked to an object type in this USER_PERMISSION_ASSIGNABLE individual, and has a user that is assigned to a role in this USER_PERMISSION_ASSIGNABLE individual, or a role that is included in this role, then it is moved to PERMITTABLE.

A formal description of the matching rule in `3_permittable` is below. USER_PERMISSION_ASSIGNABLE instance *?z* represents a potential user-permission assignment. It has the following properties:

- *rbac:action* linked to *?a*, representing an action performed by a user;
- *rbac:user ?u*, and
- *rbac:object_instance ?oi*, representing a specific data object that may be accessed by user *?u*.

?x is an instance in PRA_FULL with the following properties:

- *rbac:action* linked to *?a*;
- *rbac:role ?r1*, and
- *rbac:object_type ?o*, representing a type of object that may be accessed by users in role *?r1*.

?z is moved to PERMITTABLE if it is found to be an actual user-permission assignment in the RBAC model, according to the following rules:

i) *?z* has user *?u*;

ii) *?u* belongs to role class *?r2*;

iii) *?r2* is a sub-class of *?r1* (*?r2* is linked to *?r1* via property *rdf_ext:inferredSubclassOf*);

iv) PRA_FULL instance *?x* has role *?r1*;

v) Both *?z* and *?x* have *rbac:action ?a*;

vi) *?z* has *rbac:object_instance ?oi*;

vii) *?oi* is an individual representing a data object, belonging to class *?o* representing an object type, and

viii) *?x* has *rbac:object_type ?o*.

Figure 77 is converted into Jena syntax in Text 39 above.

```
[3_denied: (?z rdf:type rbac:DENIED )
           <-
           (?z rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
           (?x rdf:type rbac:DRA_FULL )
           (?x rbac:role ?r )
           (?x rbac:action ?a )
           (?x rbac:object_type ?o )
           (?z rbac:user ?u )
           (?z rbac:role ?r )
           (?z rbac:action ?a )
           (?z rbac:object_instance ?oi )
           (?oi rdf:type ?o )
           (?u rdf_ext:inferred_type ?r )
]
```

Text 40: Jena for rule `3_denied`.

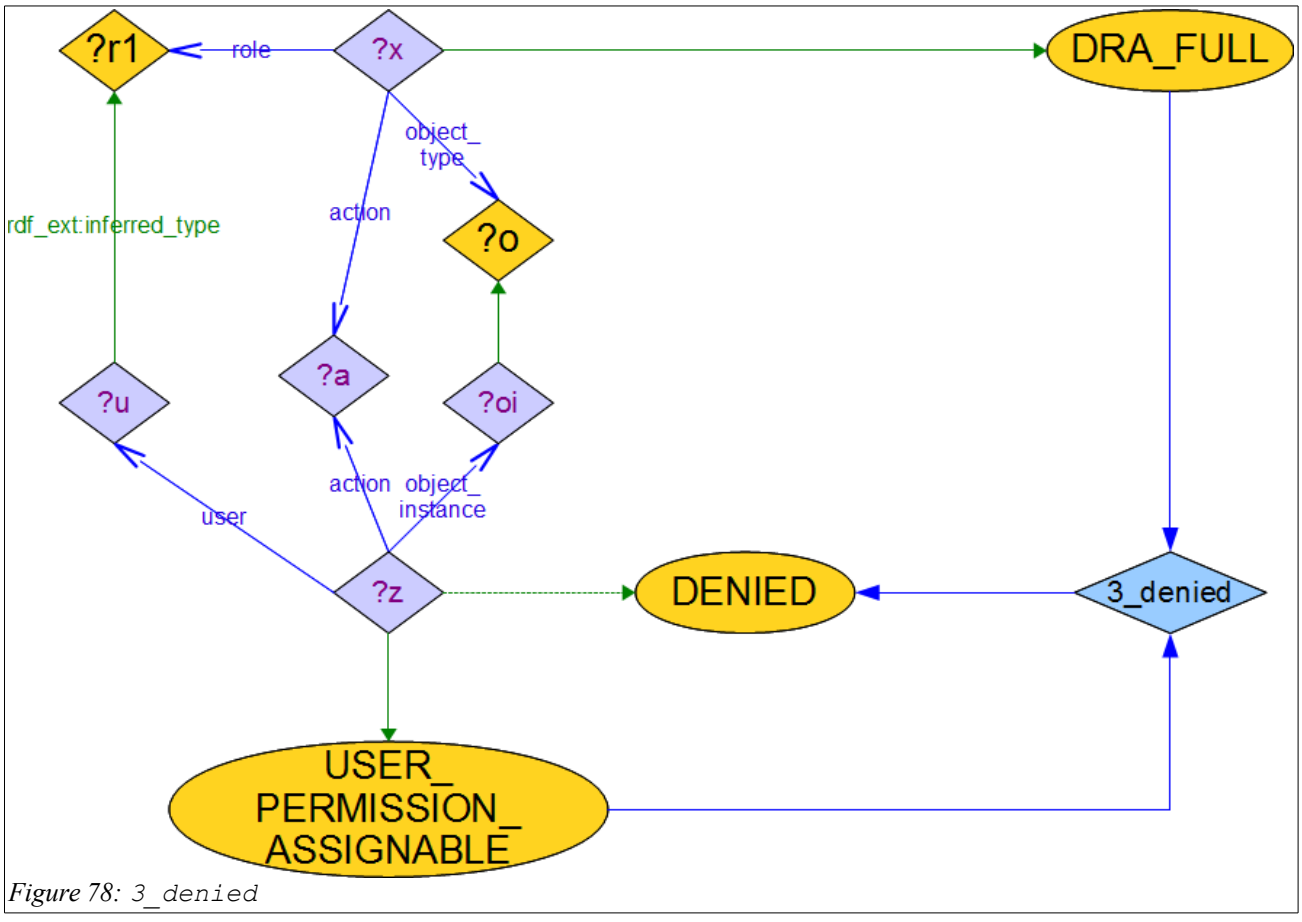

Figure 78: `3_denied`

The second rule in Step 3, given in Figure 78, is called `3_denied`. This rule moves an individual from USER_PERMISSION_ASSIGNABLE is moved to DENIED if it is found to represent an actual user-denial assignment in the RBAC model. That is, if an individual in USER_PERMISSION_ASSIGNABLE has the same action as an individual in DRA_FULL; has object instance that is linked to an object type in this USER_PERMISSION_ASSIGNABLE individual, and has a user that is assigned to a role in this USER_PERMISSION_ASSIGNABLE individual, or a role that is included in this role, then it is moved to DENIED. A formal description of the matching in rule in `3_denied` is below.

USER_PERMISSION_ASSIGNABLE instance *?z* is moved to DENIED if:

i) *?z* has *rbac:user ?u*;

ii) *?u* is an inferred member of role class *?r* (*?u* is linked to *?r* via property *rdf_ext:inferred_type*);

iii)DRA_FULL instance *?x* has *rbac:role ?r1*;

iv)both *?z* and *?x* have *rbac:action ?a*;

v) *?z* has *rbac:object_instance ?oi*;

vi)*?oi* is a data object of type *?o* (*?oi* is linked to*?oi* is an individual representing a data object, belonging to class *?o* representing an object type, and

vii)*?x* has *rbac:object_type ?o*.

Figure 78 is converted into Jena syntax in Text 40 above.

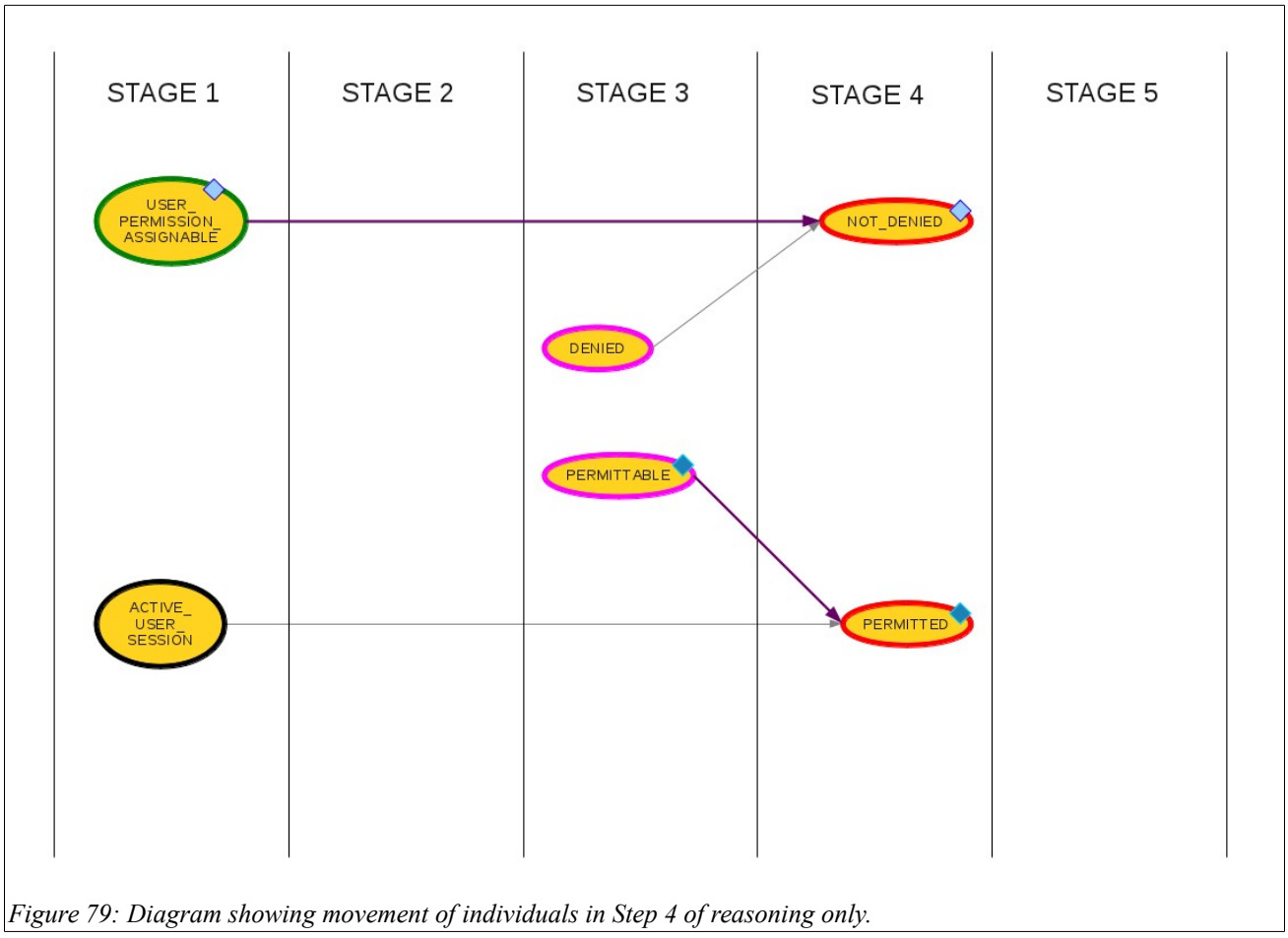

Figure 79: Diagram showing movement of individuals in Step 4 of reasoning only.

Figure 79 shows the movement of individuals in Step 4, in which PERMITTED is populated from PERMITTABLE and ACTIVE_USER_SESSION, and NOT_DENIED from USER_PERMISSION_ ASSIGNABLE and DENIED.

```
[4_not_denied: (?x rdf:type rbac:NOT_DENIED )
    <-
    (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
    noValue(?x rdf:type rbac:DENIED )
]
```

Text 41: Jena for rule `4_not_denied`.

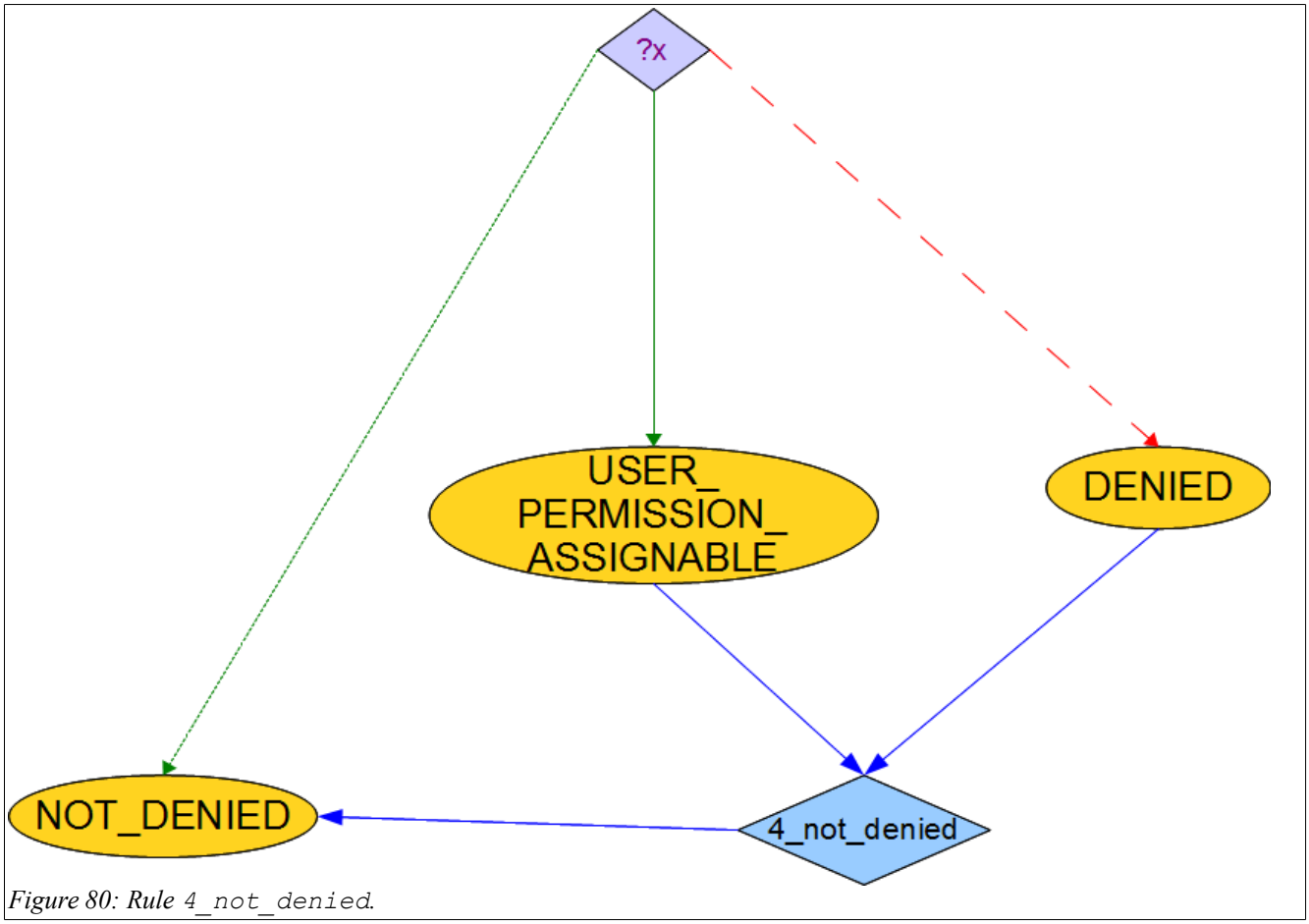

Figure 80: Rule `4_not_denied`.

The first rule in Step 4, given in Figure 80, is called `4_not_denied`. This rule populates NOT_DENIED as all individuals in USER_PERMISSION_ASSIGNABLE that are not in DENIED. Mathematically, NOT_DENIED is defined as the set difference of USER_PERMISSION_ASSIGNABLE and DENIED:

$$\text{NOT\_DENIED} = \text{USER\_PERMISSION\_ASSIGNABLE} - \text{DENIED}$$

A formal description of the matching in rule in `4_not_denied` is below. USER_PERMISSION_ASSIGNABLE instance *?x* is moved to NOT_DENIED if *?x* is not in DENIED. This is determined as follows:

i) *?d* is a set of all instances *?y* in DENIED, and

ii) *?x* is not in *?d*.

$$\forall y, \forall x, y \in \text{DENIED}, x \in \text{USER\_PERMISSION\_ASSIGNABLE}, x \neq y \Rightarrow x \in \text{NOT\_DENIED}$$

Formula 8: Matching NOT_DENIED.

Mathematically, this can be represented as in Formula 8.

$$\forall x, x \notin \text{DENIED}, x \in \text{USER\_PERMISSION\_ASSIGNABLE} \Rightarrow x \in \text{NOT\_DENIED}$$

Formula 9: Simplified matching NOT_DENIED.

Or, more simply, as in Formula 9.

The implementation of this negation formula is much simpler in Jena than in SWRL. Instead of using SQWRL, Jena implements classical negation using the function `noValue`, which returns true if an RDF triple is *not* valid for any individuals in a set. Unlike in SO-RBAC using SWRL, the negation test does not need DENIED to have a dummy individual.

Figure 80 is converted into Jena syntax in Text 41 above.

```
[4_permitted: (?z rdf:type
rbac:PERMITTED )
    <-
    (?z rdf:type rbac:PERMITTABLE )
    (?z rbac:user ?u )
    (?z rbac:role ?r )
    (?s rbac:user ?u )
    (?s rbac:role ?r )
    (?s rdf:type
rbac:ACTIVE_USER_SESSION )
]
```

Text 42: Jena for rule `4_permitted`.

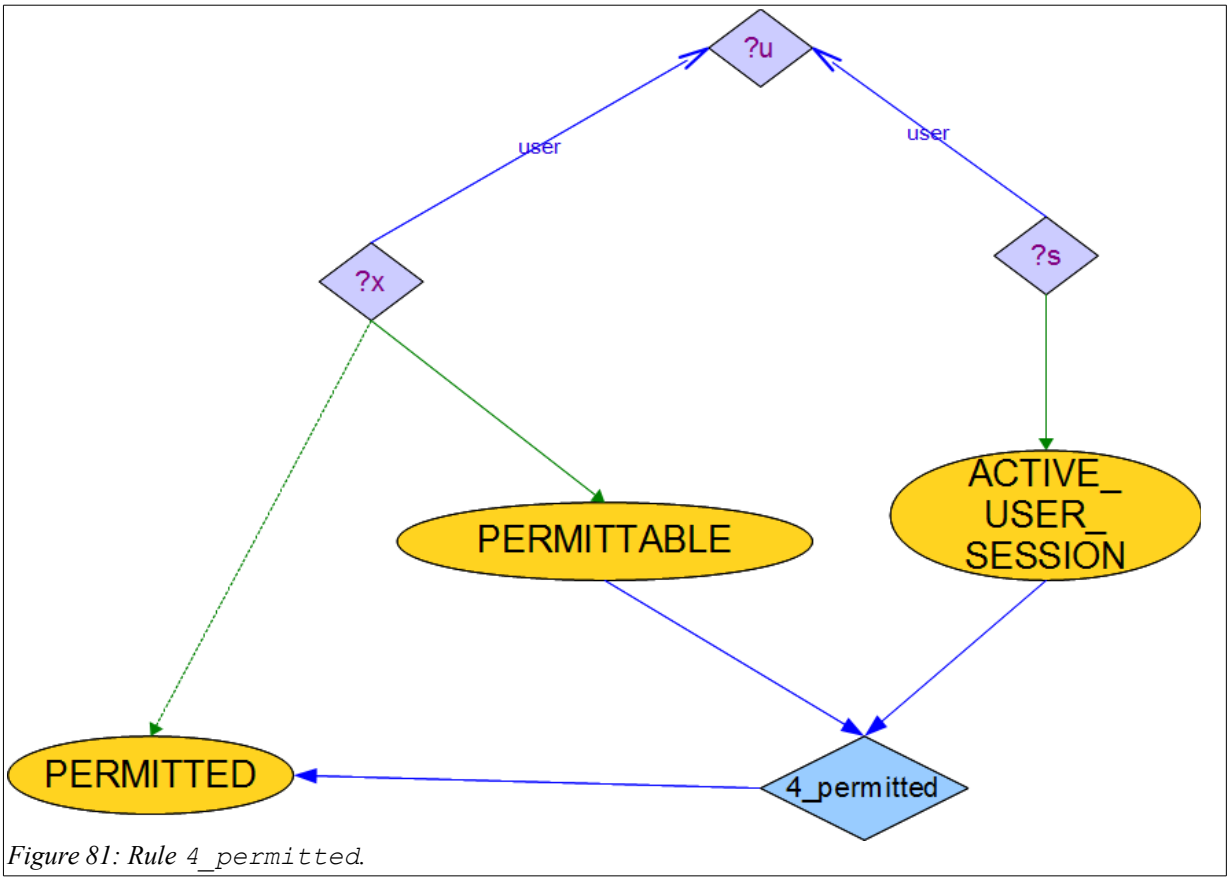

Figure 81: Rule `4_permitted`.

The second rule in Step 4, given in Figure 81, is called `4_permitted`. This rule moves an individual in PERMITTABLE to PERMITTED if the individual has a user that is also a user in an active user session, as given by an individual in the class ACTIVE_USER_SESSION. The difference between PERMITTABLE and PERMITTED is that PERMITTABLE represents potential permissions, while PERMITTED represents actual permissions, as determined by active user sessions.

A formal description of the matching in rule in `4_permitted` is below. PERMITTABLE instance *?x* is moved to PERMITTED if:

i) *?x* has *rbac:user ?u*, and

ii) ACTIVE_USER_SESSION user *?s* has *rbac:user ?u*.

Figure 81 is converted into Jena syntax in Text 42 above.

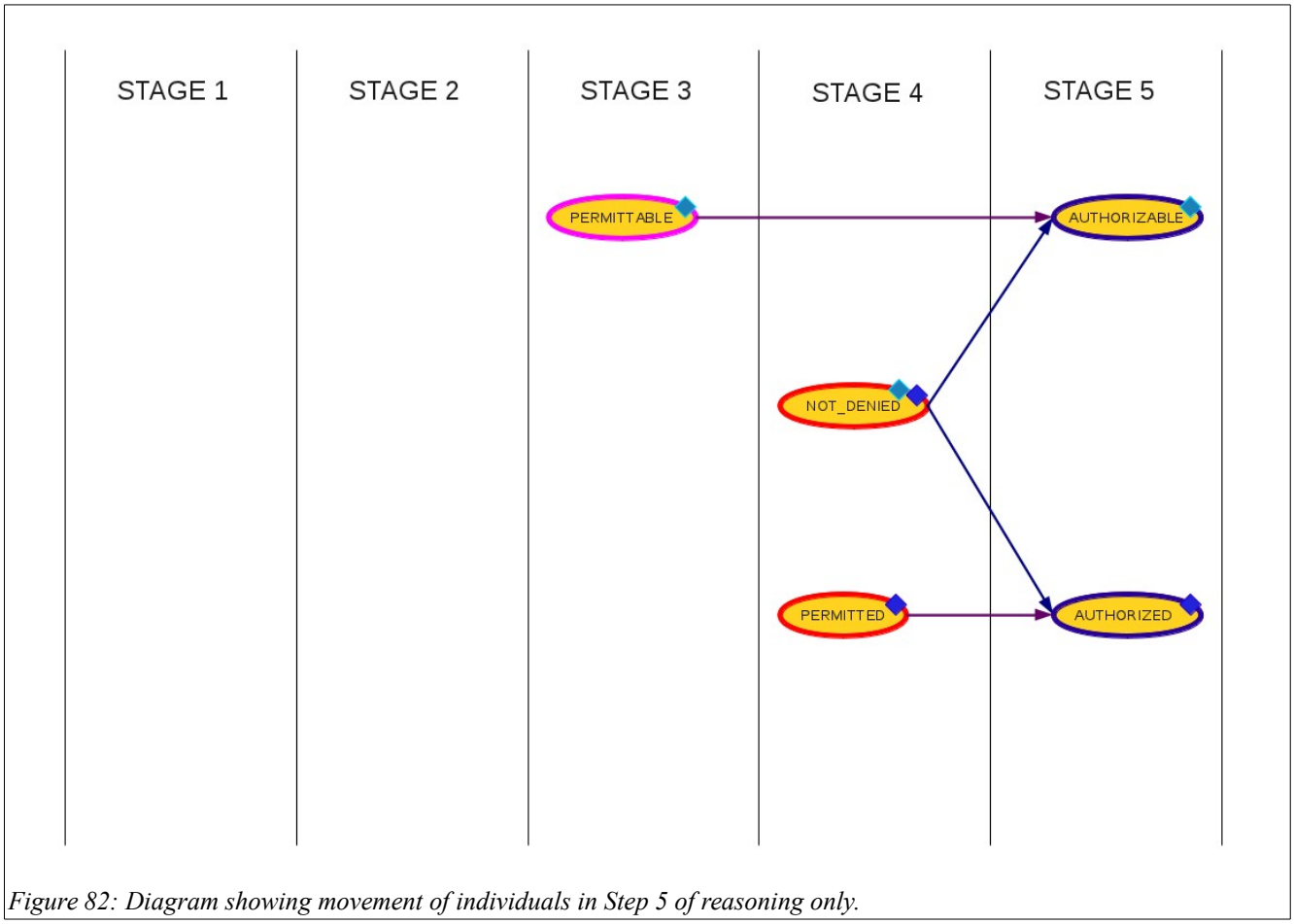

Figure 82: Diagram showing movement of individuals in Step 5 of reasoning only.

Figure 82 shows the movement of individuals in Step 5, in which AUTHORIZED is populated from PERMITTED and NOT_DENIED. AUTHORIZABLE is populated from PERMITTABLE and NOT_DENIED.

```
[5_authorizable: (?x rdf:type rbac:AUTHORIZABLE )
    <-
    (?x rdf:type rbac:PERMITTABLE )
    (?x rdf:type rbac:NOT_DENIED )
]
```

Text 43: Jena for rule `5_authorizable`.

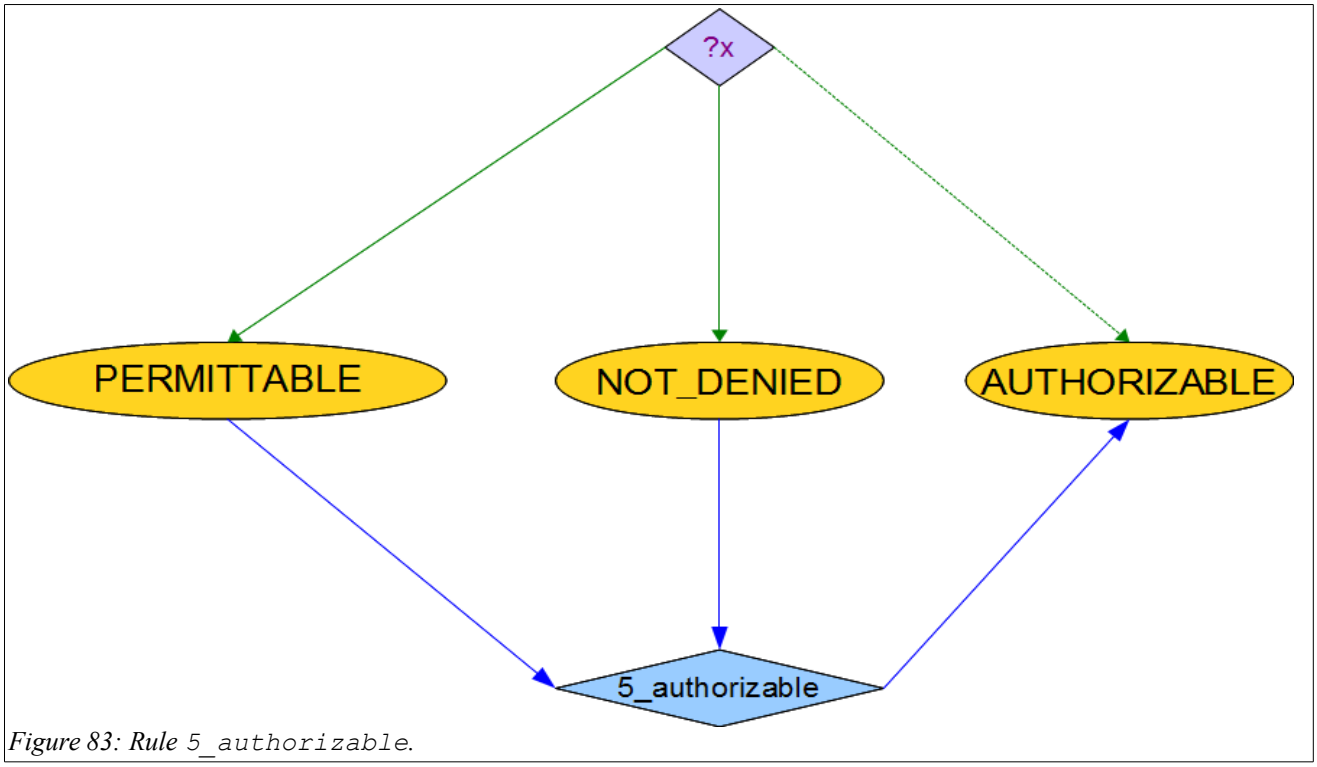

Figure 83: Rule `5_authorizable`.

The first rule in Step 5, given in Figure 83, is called `5_authorizable`. AUTHORIZABLE is defined as the intersection of PERMITTABLE and NOT_DENIED: an individual is in AUTHORIZABLE if it is in both PERMITTABLE and NOT_DENIED (Formula 10).

PERMITTABLE instance *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED. Note that this means that the same actual instance *?x* has to be in both PERMITTABLE and NOT_DENIED (not different instances with the same object properties).

$$AUTHORIZABLE = PERMITTABLE \cap NOT\_DENIED$$

Formula 10: Definition of AUTHORIZABLE.

Figure 83 is converted into Jena syntax in Text 43 above.

```
[5_authorized: (?x rdf:type rbac:AUTHORIZED )
    <-
    (?x rdf:type rbac:PERMITTED )
    (?x rdf:type rbac:NOT_DENIED )
]
```

Text 44: Jena for rule `5_authorized`.

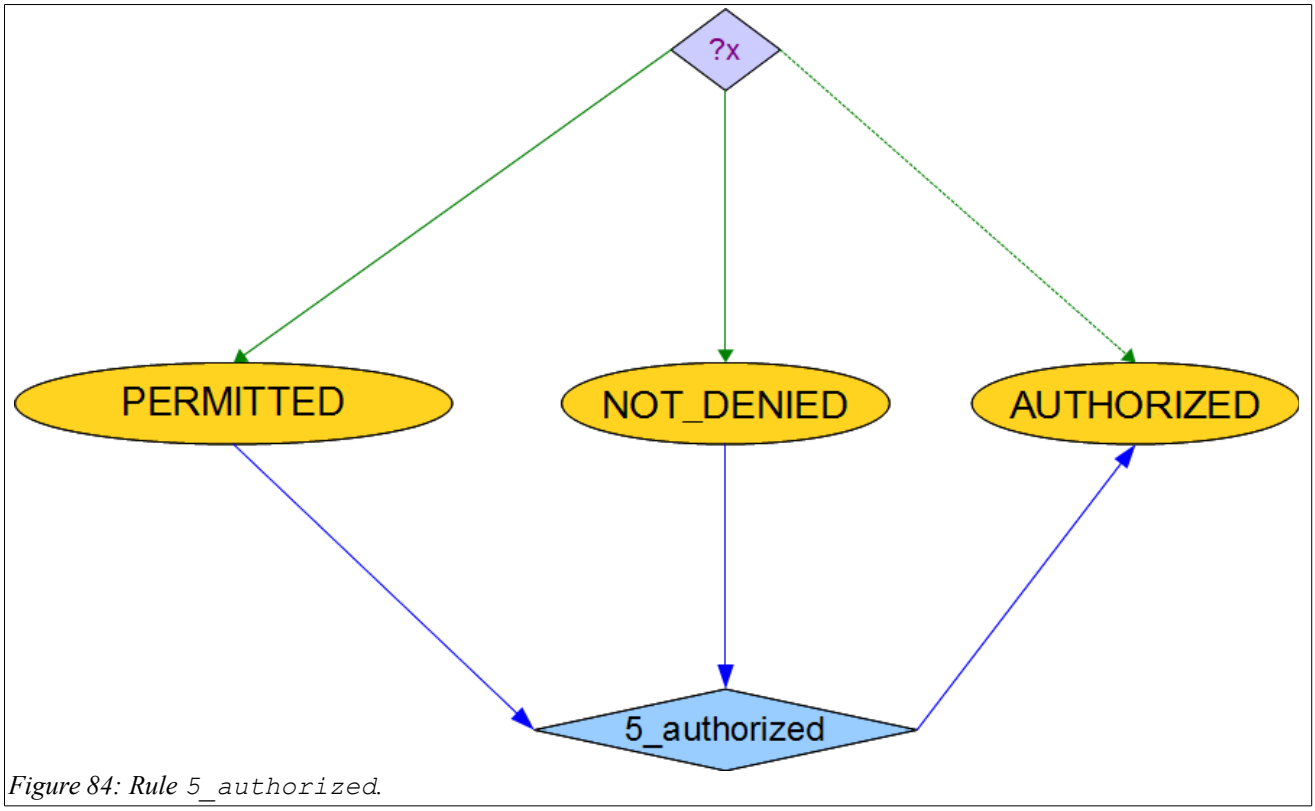

Figure 84: Rule `5_authorized`.

The second rule in Step 5, given in Figure 84, is called `5_authorized`. This defines AUTHORIZED as the intersection of PERMITTED and NOT_DENIED: an individual is in AUTHORIZED if it is in both PERMITTED and NOT_DENIED.

PERMITTED instance *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED.

$$AUTHORIZED = PERMITTED \cap NOT\_DENIED$$

Formula 11: Definition of AUTHORIZED.

Figure 84 is converted into Jena syntax in Text 44 above.

6.3 ESO-RBAC Process

Figure 85 (on page 150) shows a flowchart of the process in which ESO-RBAC is run. This is very similar to the process for SO-RBAC.

Each potential *role* permission or denial to perform an action on an object is represented by an individual in the ESO-RBAC class ROLE_PERMISSION_ASSIGNABLE. The process moves individuals representing role permissions and denials to the class PRA_FULL (indicating a permission) or DRA_FULL (indicating a denial).

Each potential *user* permission or denial to perform an action on an object is represented by an individual in the SO-RBAC class USER_PERMISSION_ASSIGNABLE. The process moves individuals representing role permissions and denials to the classes PERMITTABLE or PERMITTED (indicating a permission) or DENIED (indicating a denial). Finally, an individual representing a user permission that is not also a denial is moved to AUTHORIZABLE or AUTHORIZED. All this is done according to rules in the ESO-RBAC process.

The first step, step A, is to set up the ontology class hierarchy. This includes setting up the data classes under OBJECT_INSTANCE, to which the data that the ESO-RBAC model governs access, and the classes relevant to the ESO-RBAC model itself, under RBAC class. OBJECT_INSTANCE hierarchies are always domains specific, but RBAC sub-hierarchies are likely to remain the same across domains.

Step B populates the ontology with the base information. It has two parts, B1 and B2, which can be run in parallel. This is because they are independent of each other.

```
domain := "rbac";
domain_uri := "http://www.cgce.net/Ontology/RBAC";

class: = "ROLE_PERMISSION_ASSIGNABLE";
for each role
  for each action
    for each object_type
      id := "role_action_object_type"
      print "  <domain:class rdf:ID=\"id\">";
      print "    <domain:action rdf:resource=\"#action\"/>";
      print "    <domain:role rdf:resource=\"#role\"/>";
      print "    <domain:object_type rdf:resource=\"#$object_type\"/>";
      print "  </$domain:$class>";
    next
  next
next

class := "USER_PERMISSION_ASSIGNABLE";
for each action
  for each object_instance
    for each user
      id = user_action_object_instance";
      print "  <domain:class rdf:ID=\"id\">";
      print "    <domain:action rdf:resource=\"#action\"/>";
      print "    <domain:object_instance rdf:resource=\"#object_instance\"/>";
      print "    <domain:user rdf:resource=\"#user\"/>";
      print "  </domain:class>";
    next
  next
next
```

Text 45: Pseudocode for step C

```
[inferred_subClassOf_1: (?c1 rdfs:inferred_subClassOf ?c2)
           <-
           (?c1 rdfs:subClassOf ?c2)
]
[inferred_subClassOf_2: (?c1 rdfs:inferred_subClassOf ?c3)
           <-
           (?c1 rdfs:subClassOf ?c2)
           (?c2 rdf_ext:inferred_subClassOf ?c3)
]
[inferred_type_1: (?i rdf_ext:inferred_type ?c)
           <-
           (?i rdf:type ?c)
]
[inferred_type_2: (?i rdf_ext:inferred_type ?c)
           <-
           (?c1 rdf_ext:inferred_subClassOf ?c)
           (?i rdf:type ?c1)
]
[1_senior_to_1: (?r rbac:senior_to ?r)
    <-
   (?r rdf:type rbac:ROLE_SET)
   (?r rbac:directly_senior_to ?r1)
]
[1_senior_to_2: (?r rbac:senior_to ?r)
    <-
   (?r rdf:type rbac:ROLE_SET)
   (?r1 rbac:directly_senior_to ?r)
]
[1_senior_to_4: (?r1 rbac:senior_to ?r3)
    <-
   (?r1 rdf:type rbac:ROLE_SET)
   (?r1 rbac:directly_senior_to ?r2)
   (?r2 rbac:senior_to ?r3)
]
[1_junior_to: (?r1 rbac:junior_to ?r2)
           <-
          (?r rdf:type rbac:ROLE_SET)
          (?r2 rbac:senior_to ?r1)
]
[1_inherits_pra_1: (?r rbac:inherits_pra ?r)
    <-
    (?r rdf:type rbac:ROLE_SET)
    notEqual(?r, rbac:ROLE)
]
[1_inherits_pra_3: (?r2 rbac:inherits_pra ?r3)
    <-
     (?r1 rbac:senior_to ?r2)
     (?r2 rbac:senior_to ?r3)
     (?r3 rbac:senior_to ?r4)
     (?r1 rbac:inherits_pra_path ?r4)
]
```

Code 59: Jena Rules for Step E.

In step B1, the classes OBJECT_TYPE and ROLE_SET (and thus creating the class hierarchy under ROLE), representing the RBAC object types and roles, are populated. Note that unlike in SO-RBAC, users are not set up until step H, due to the different way in which the relationship between users and roles is represented. In B2, the data classes (sub-classes of OBJECT_INSTANCE) are populated. In step C, the classes ROLE_PERMISSION_ASSIGNABLE

and USER_PERMISSION_ASSIGNABLE are populated, with all possible combinations of hypothetical role and user permission assignment. Due to the exponentially increasing number of combinations, this is most likely to be done using a program or script, according to the pseudocode in Text 45.

In step D, the asserted relationships between members of ROLE_SET (*directly_senior_to*, *is_a*, *inherits_pra_path*) are set up.

In step E, the Jena rules are run to infer the object properties that depend on the properties asserted in step D, namely *senior_to*, *included_in* and *inherits_pra*, which are respectively dependent on *directly_senior_to*, *is_a* and *inherits_pra_path*. The nine Jena rules are described in Section 6.2.3.1, and are summarised in Code 59.

In step F, the PRA and DRA classes are populated to set up role permissions and denials, because the individuals in these classes are base information for reasoning in the RBAC model. This can be done after step E because the information about role permissions and denials is not needed for inferring relationships between roles.

```
[2_dra_full: (?z rdf:type rbac:DRA_FULL )
    <-
    (?z rdf:type rbac:ROLE_PERMISSION_ASSIGNABLE )
    (?z rbac:role ?r2 )
    (?z rbac:action ?a )
    (?z rbac:object_type ?o )
    (?r1 rbac:senior_to ?r2 )
    (?x rdf:type rbac:DRA )
    (?x rbac:role ?r1 )
    (?x rbac:action ?a )
    (?x rbac:object_type ?o )
]
[2_pra_full: (?z rdf:type rbac:PRA_FULL )
    <-
    (?z rdf:type rbac:ROLE_PERMISSION_ASSIGNABLE )
    (?z rbac:role ?r2)
    (?z rbac:action ?a )
    (?z rbac:object_type ?o )
    (?r2 rbac:senior_to ?r1 )
    (?r2 rbac:inherits_pra ?r1)
    (?x rdf:type rbac:PRA )
    (?x rbac:role ?r1 )
    (?x rbac:action ?a )
    (?x rbac:object_type ?o )
]
```

Code 60: Jena Rules for Step G.

In Step G, we populate the PRA_FULL and DRA_FULL classes with individuals through inference by running the following two Jena rules in Code 60 (*cf.* Section 6.2.3.2).

In step H, the user-role relationships are set up, i.e. ROLE sub-classes are populated with individuals representing users. Again, this is essential information needed for reasoning in the RBAC model, but it is not needed for inferring either relationships between roles or assignment of permissions or denials to roles.

Finally, in step J, the remaining reasoning steps (3–5) are performed. We run 6 Jena rules (Code 61, page 149) which ultimately populated DENIED or AUTHORIZED classes (*cf.* Section 6.2.3.2).

At each stage in Figure 85, the ESO-RBAC ontology is in a state where the process can be run from the following step onwards. In other words, it is not necessary to always re-run the ESO-RBAC process from the beginning.

```
[3_permittable: (?z rdf:type rbac:PERMITTABLE )
           <-
           (?z rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
           (?x rdf:type rbac:PRA_FULL )
           (?x rbac:role ?r )
           (?x rbac:action ?a )
           (?x rbac:object_type ?o )
           (?z rbac:user ?u )
           (?z rbac:role ?r )
           (?z rbac:action ?a )
           (?z rbac:object_instance ?oi )
           (?oi rdf:type ?o )
           (?u rdf_ext:inferred_type ?r )
]
[3_denied: (?z rdf:type rbac:DENIED )
           <-
           (?z rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
           (?x rdf:type rbac:DRA_FULL )
           (?x rbac:role ?r )
           (?x rbac:action ?a )
           (?x rbac:object_type ?o )
           (?z rbac:user ?u )
           (?z rbac:role ?r )
           (?z rbac:action ?a )
           (?z rbac:object_instance ?oi )
           (?oi rdf:type ?o )
           (?u rdf_ext:inferred_type ?r )
]
[4_not_denied: (?x rdf:type rbac:NOT_DENIED )
    <-
    (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
    noValue(?x rdf:type rbac:DENIED )
]
[4_permitted: (?z rdf:type rbac:PERMITTED )
    <-
    (?z rdf:type rbac:PERMITTABLE )
    (?z rbac:user ?u )
    (?z rbac:role ?r )
    (?s rbac:user ?u )
    (?s rbac:role ?r )
    (?s rdf:type rbac:ACTIVE_USER_SESSION )
]
[5_authorizable: (?x rdf:type rbac:AUTHORIZABLE )
    <-
    (?x rdf:type rbac:PERMITTABLE )
    (?x rdf:type rbac:NOT_DENIED )
]
[5_authorized: (?x rdf:type rbac:AUTHORIZED )
    <-
    (?x rdf:type rbac:PERMITTED )
    (?x rdf:type rbac:NOT_DENIED )
]
```

Code 61: Jena Rules for Step J.

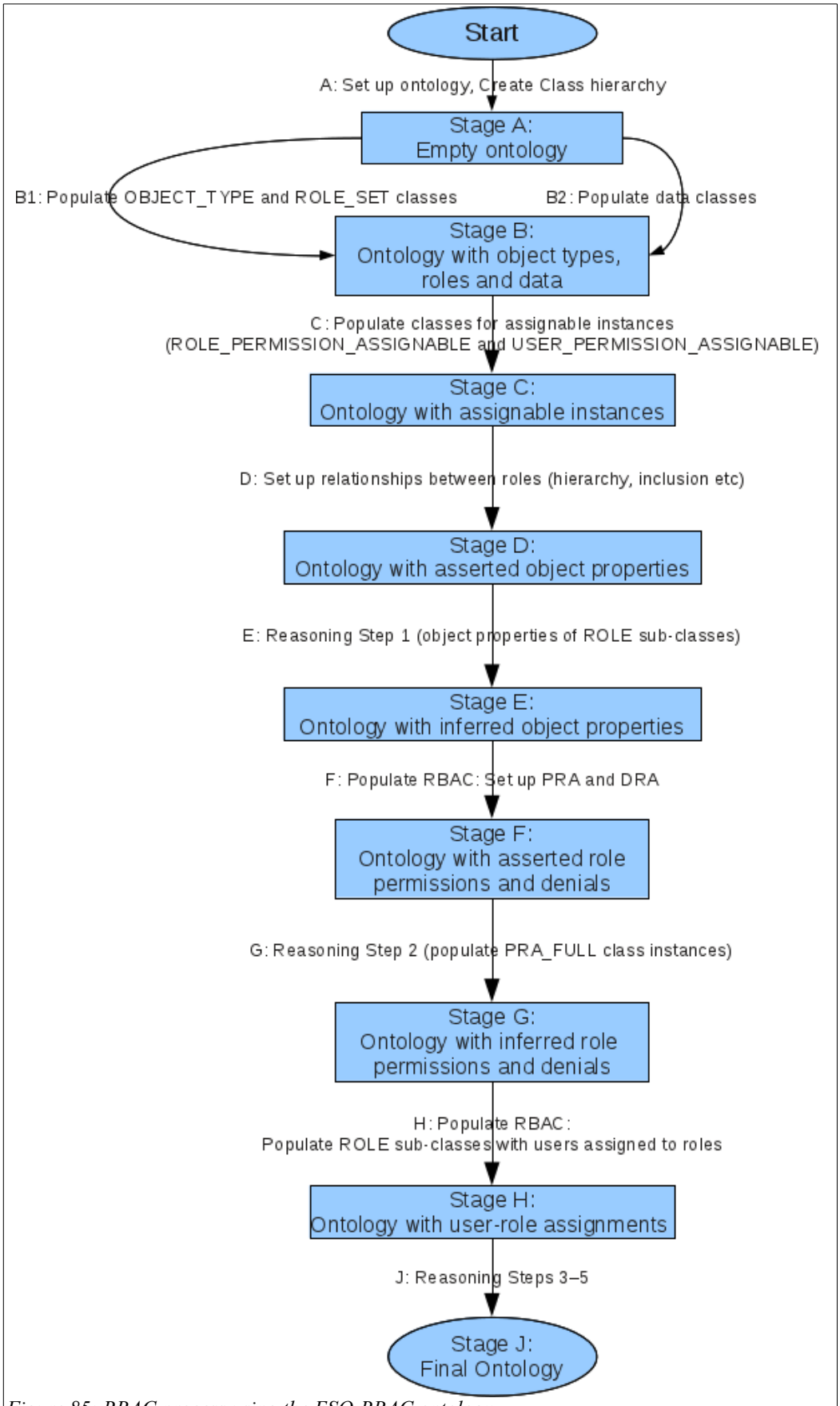

Figure 85: RBAC process using the ESO-RBAC ontology.

6.4 Modelling Dynamic RBAC in ESO-RBAC

Dynamic RBAC was also modelled according to the Strembeck & Neumann [21] model, and an example context constraint was created and tested.

Table 18: Fact definition used in dynamic RBAC design in Prolog.

Fact Formula	*Description*
`associated_cc(Role, Permission, Object, ContextConstraint).`	The context condition `ContextConstraint` applies when a user with role Role accesses object Object using Permission.

Table 19: Rules in Prolog dynamic RBAC design.

Rule Name	*Description*
`applied_cc`	Whether a context constraint applies to a user performing an action.
`fail_context_constraint`	Whether an action fails a context constraint, considering its applicability.
`violated`	Whether an action would fail a context constraint, irrespective of its applicability.
`context_condition`	Defines the circumstances in which a user can perform an action on an object.

To recap, dynamic RBAC in predicate logic is based on the Prolog facts in Table 18, and the rules in Table 19. Note that Separation of Duties was not implemented here.

Dynamic RBAC in ESO-RBAC uses Jena rules `context_constraint_applied`, `context_condition_pass_1` and `context_condition_pass_2`. Dynamic RBAC requires the following new classes in the ESO-RBAC ontology:

- CONTEXT_CONSTRAINT directly under RBAC.
- CONTEXT_CONDITION_PASS, CONTEXT_CONDITION_POTENTIAL and CONTEXT_CONDITION under USER_PERMISSION_ASSIGNABLE. CONTEXT_CONDITION is a sub-class of CONTEXT_CONDITION_POTENTIAL. These have properties *user*, *action*, *object* and *context_constraint*; *context_constraint* has CONTEXT_CONSTRAINT as its range.
- CONTEXT_CONSTRAINT_APPLICABLE, CONTEXT_CONSTRAINT_ASSOCIATED and CONTEXT_CONSTRAINT_APPLIED, under ROLE_PERMISSION_ASSIGNABLE. CONTEXT_CONSTRAINT_ASSOCIATED and CONTEXT_CONSTRAINT_APPLIED are sub-classes of CONTEXT_CONSTRAINT_APPLICABLE.

It should be noted that context *conditions* apply to combinations of <user, action, object>, while context *constraints* apply to combinations of <role, action, object>.

A context constraint is represented as follows:

- An individual in the class CONTEXT_CONSTRAINT, given the canonical name of the context constraint, but with no other information about it.
- Individuals in the class CONTEXT_CONSTRAINT_APPLICABLE representing all possible combinations of context constraint, role, action and object. This may be populated using a script. Some individuals in

CONTEXT_CONSTRAINT_APPLICABLE are also members of CONTEXT_CONSTRAINT_ASSOCIATED, which is equivalent to the `associated_cc` facts in the Prolog implementation.

- Individuals in the class CONTEXT_CONDITION_POTENTIAL representing all possible combinations of context constraint, user, (role), action and object. This may also be populated using a script.
- One or more Jena rules defining the applicability of the context constraint.

```
[context_constraint_applied: (?x rdf:type
rbac:CONTEXT_CONSTRAINT_APPLIED)
            <-
              (?cc rdf:type rbac:CONTEXT_CONSTRAINT)
              (?y rdf:type rbac:CONTEXT_CONSTRAINT_ASSOCIATED)
              (?y rbac:context_constraint ?cc)
              (?y rbac:role ?r3)
              (?y rbac:action ?a)
              (?y rbac:object ?o)
              (?r3 rbac:senior_to ?r2)
              (?r1 rdf_ext:inferred_subClassOf ?r2)
              (?x rdf:type rbac:CONTEXT_CONSTRAINT_APPLICABLE)
              (?x rbac:role ?r1)
              (?x rbac:action ?a)
              (?x rbac:object ?o)
              (?x rbac:context_constraint ?cc)
]
```

Text 46: Jena rule for `context_constraint_applied`.

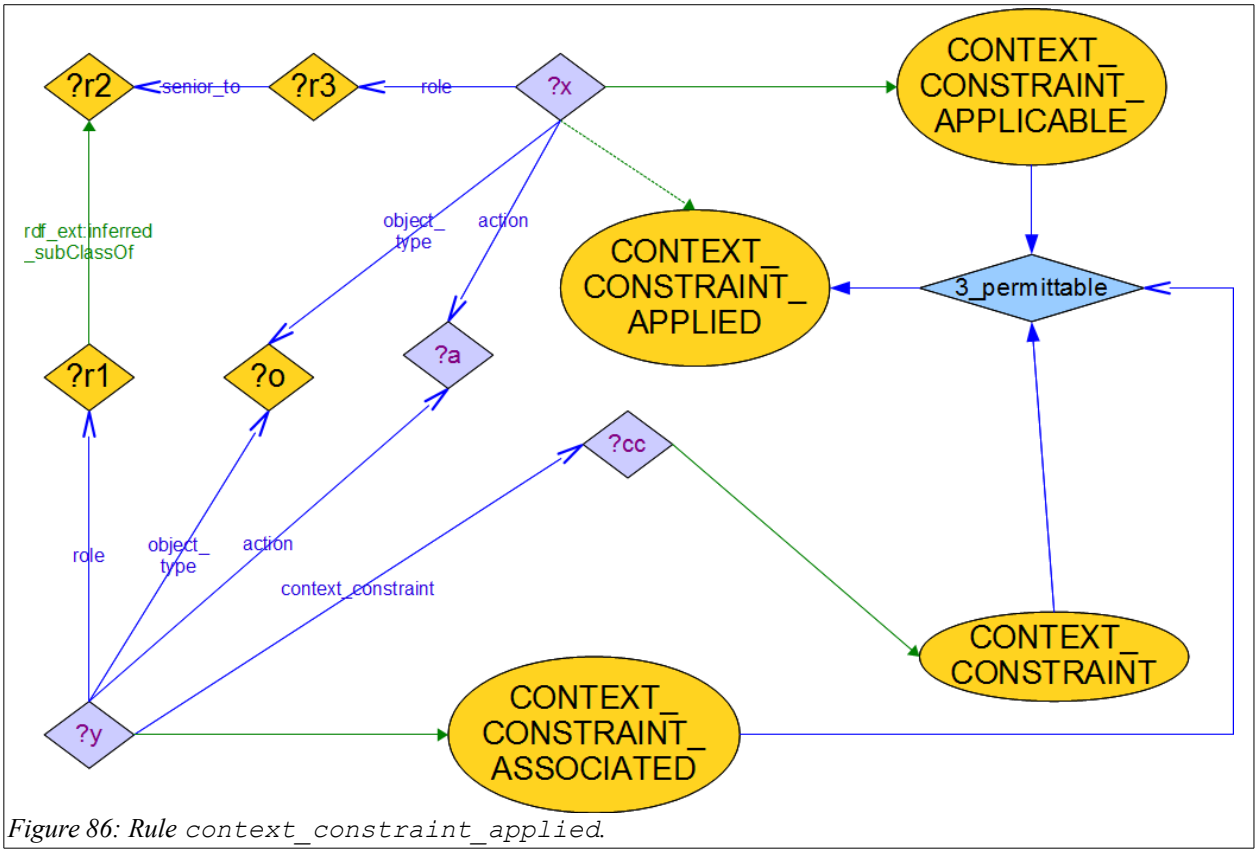

Figure 86: Rule `context_constraint_applied`.

The Jena rule `context_constraint_applied`, given in Figure 86, is analogous to the Prolog rule `applied_cc`. It determines whether a context constraint is applicable to a particular combination of role, action and object, depending on membership of CONTEXT_CONSTRAINT_ASSOCIATED and seniority and inclusion relationships among roles. As in the Prolog implementation, context constraints filter down the seniority hierarchy. Thus if a <role, action, object> combination is explicitly associated with a context constraint, via an individual in CONTEXT_CONSTRAINT_ASSOCIATED, then any <role, action, object> combinations for this role and any roles junior to and/or inside it have the context constraint applied to it. all individuals linking these <role, action, object> with the context constraint are moved to CONTEXT_CONSTRAINT_APPLIED.

CONTEXT_CONSTRAINT_APPLICABLE individual *?x* is moved to CONTEXT_CONSTRAINT_APPLIED if:

1. *?x* has role *?r1*;
2. *?r1* is an inferred sub-class of *?r2*, and *?r3* is senior to *?r2*;
3. CONTEXT_CONSTRAINT_ASSOCIATED individual *?y* has role *?r3*, and
4. *?x* and *?y* both have action *?a*, object *?o* and CONTEXT_CONSTRAINT individual ?cc.

Figure 86 is converted into Jena syntax in Text 46 above.

Jena rules `context_condition_pass_1` and `context_condition_pass_2` determine whether a <user, action, object> combination passes a particular context constraint, either because it passes any context condition (`context_condition_pass_1`) or because no context condition applies to it (`context_condition_pass_2`).

```
[context_condition_pass_1: (?x rdf:type rbac:CONTEXT_CONDITION_PASS)
           <-
            (?y rdf:type rbac:CONTEXT_CONDITION)
            (?y rbac:user ?u)
            (?y rbac:action ?a)
            (?y cgce:object ?o)
            (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE)
            (?x rbac:user ?u)
            (?x rbac:action ?a)
            (?x cgce:object ?o)
]
```

Text 47: Jena for rule `context_condition_pass_1`.

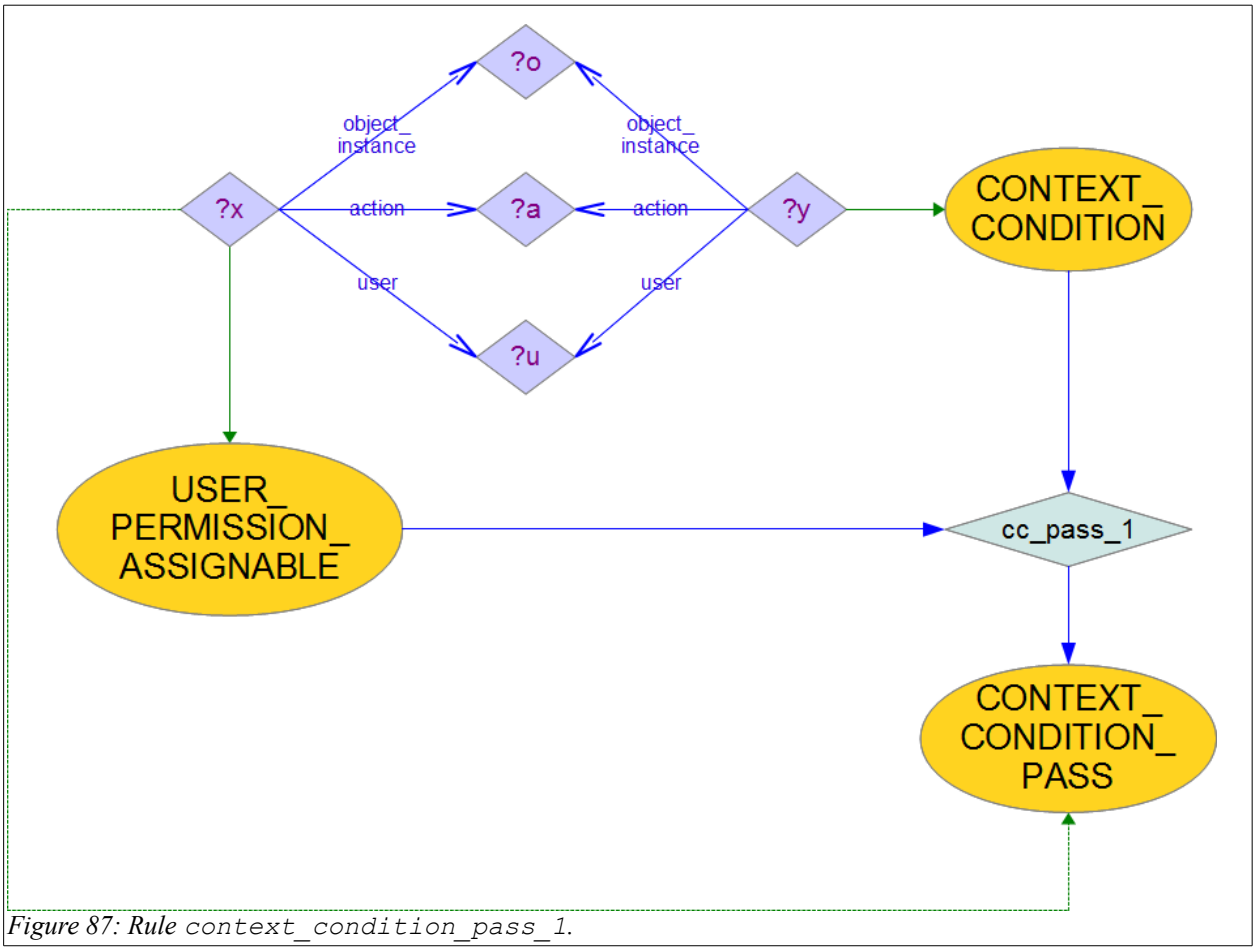

Figure 87: Rule `context_condition_pass_1`.

`context_condition_pass_1`, given in Figure 87, moves USER_PERMISSION_ASSIGNABLE individual *?x* to CONTEXT_CONDITION_PASS if *?x* has the same values for *rbac:user*, *rbac:action* and *object* as a CONTEXT_CONDITION individual *?y*. Note that *?y* also has a *context_constraint* property, but *context_condition_pass_1* is not concerned about the value of this: it only needs to know that *?y* exists, not what context constraints it is linked to.

Figure 87 is converted into Jena syntax in Text 47 above.

```
[context_condition_pass_2: (?x rdf:type rbac:CONTEXT_CONDITION_PASS)
           <-
            (?y rdf:type rbac:CONTEXT_CONSTRAINT_APPLICABLE)
            (?y rbac:action ?a)
            (?y rbac:object ?o)
            (?y rbac:role ?r)
           noValue(?y rdf:type rbac:CONTEXT_CONSTRAINT_APPLIED )
            (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE)
            (?x rbac:user ?u)
            (?x rbac:action ?a)
            (?x cgce:object ?oi)
            (?u rdf:type ?r)
            (?oi rdf:type ?o )
]
```

Text 48: Jena for rule `context_condition_pass_2`.

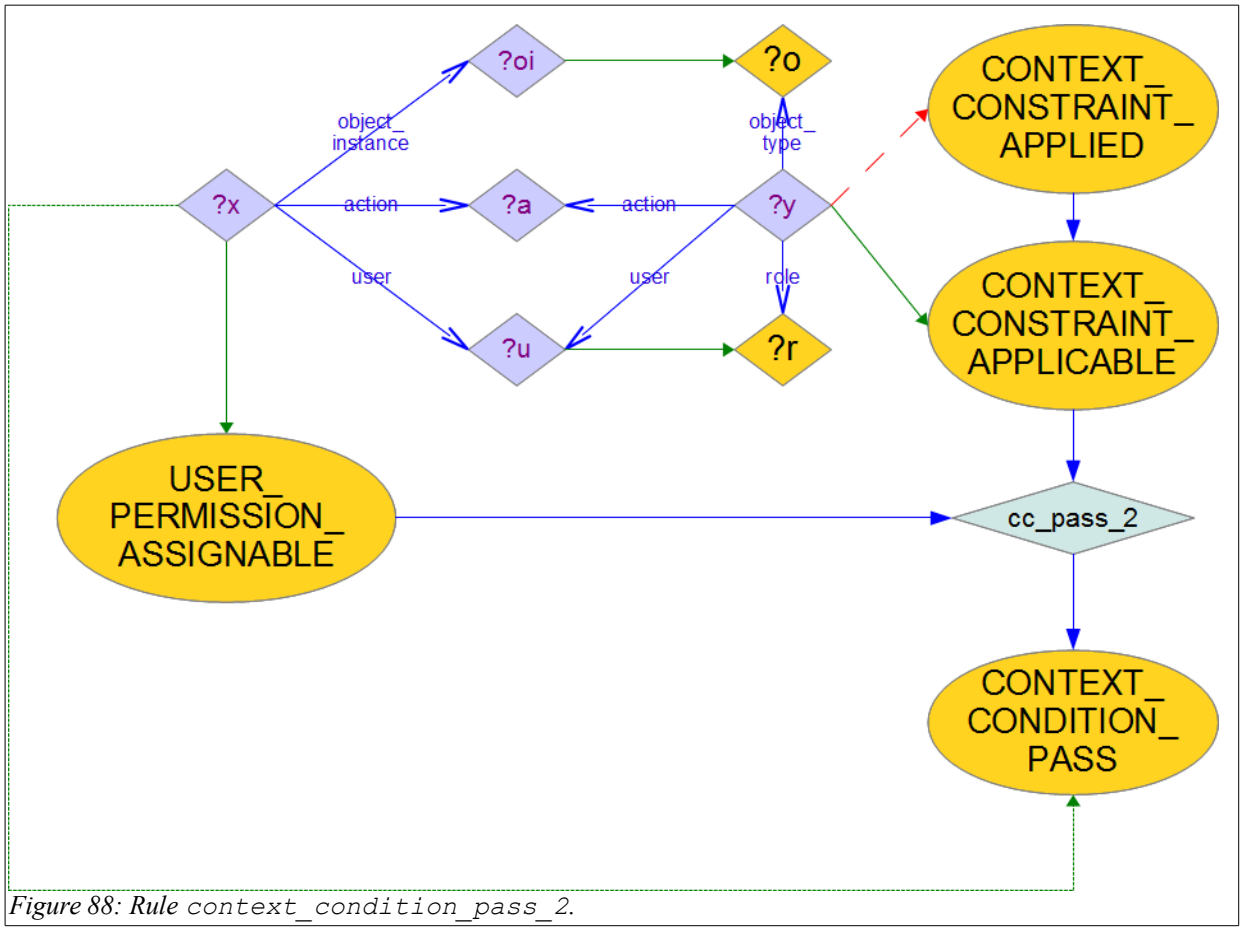

Figure 88: Rule `context_condition_pass_2`.

`context_condition_pass_2`, given in Figure 88, moves USER_PERMISSION_ASSIGNABLE individual *?x* to CONTEXT_CONDITION_PASS if there is no individual in CONTEXT_CONSTRAINT_APPLIED with the same *user*, *action* and *object* properties as *?x*. The rule uses the `noValue` function to check all individuals in CONTEXT_CONSTRAINT_APPLICABLE and determine that none of them are in CONTEXT_CONSTRAINT_APPLIED. An individual would have been moved into CONTEXT_CONSTRAINT_APPLIED by the rule `context_constraint_applied`.

Figure 88 is converted into Jena syntax in Text 48 above.

Individuals are moved into CONTEXT_CONDITION by the Jena rules relating to the specific context constraints.

The following is an example of a context constraint, *nurse_in_same_ward_as_patient*, which tests whether a nurse is attached to the same ward that a particular patient is in. This context constraint is associated with the role SENIOR_NURSE, for actions read and write on individuals in the class PATIENT. That is, the <role, action, object> combinations associated with *nurse_in_same_ward_as_patient* are <SENIOR_NURSE, read, PATIENT> and <SENIOR_NURSE, write, PATIENT>. In other words, a nurse in role SENIOR_NURSE or junior to this can only read and write information about patient in a ward to which he or she is attached. This context constraint is defined as follows:

- Individuals in CONTEXT_CONSTRAINT_ APPLICABLE are defined to represent all possible <role, action, object> combinations in the ontology for the context constraint *nurse_in_same_ward_as_patient*. The two individuals representing the combinations <SENIOR_NURSE, read, PATIENT> and <SENIOR_NURSE, write, PATIENT> are moved to CONTEXT_CONSTRAINT_ASSOCIATED.
- Individuals in CONTEXT_CONDITION_POTENTIAL are defined representing all possible <user, action, object> combinations for the context constraint *nurse_in_same_ward_as_patient*.

```
[nurse_in_same_ward_as_patient: (?x rdf:type rbac:CONTEXT_CONDITION)
            <-
             (?x rdf:type rbac:CONTEXT_CONDITION_POTENTIAL)
             (?x rbac:context_constraint cgce:nurse_in_same_ward_as_patient)
             (?x rbac:user ?nurse)
             (?x cgce:object ?patient)
             (?patient rdf:type cgce:PATIENT)
             (?nurse cgce:nurse_ward ?ward)
             (?patient cgce:patient_ward ?ward)
]
```

Text 49: Jena for rule `nurse_in_same_ward_as_patient`.

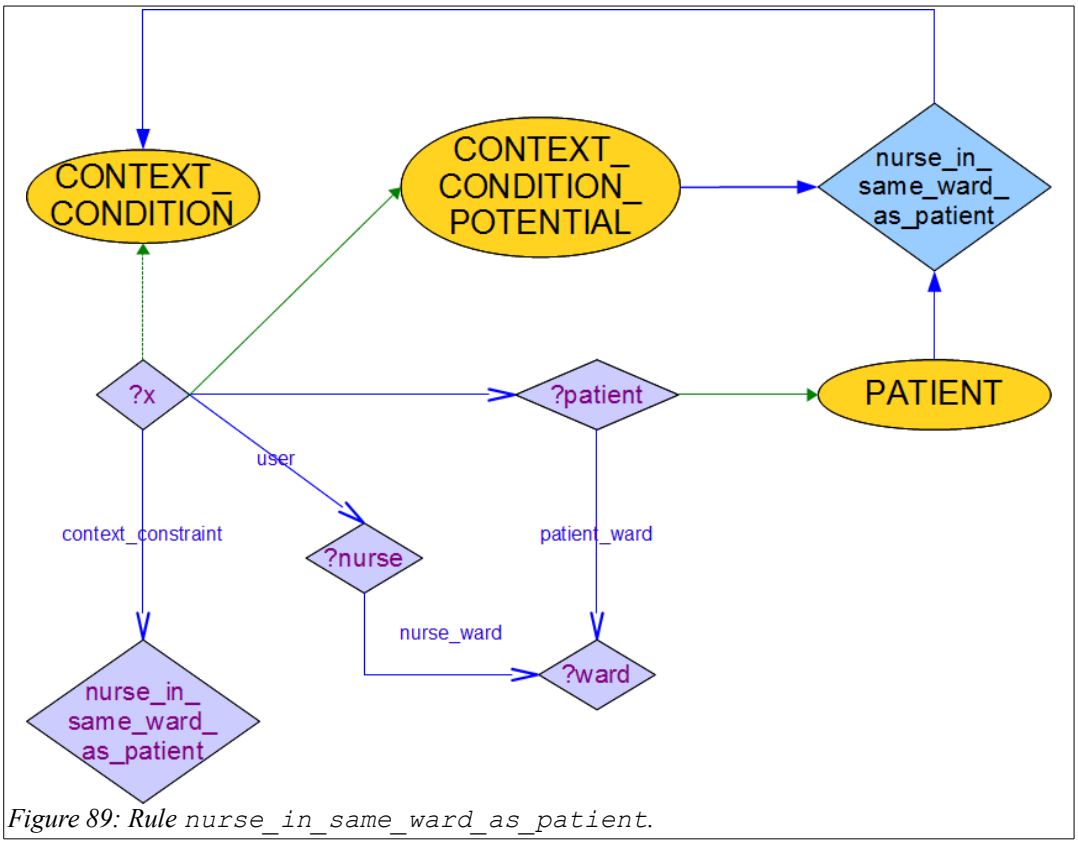

Figure 89: Rule `nurse_in_same_ward_as_patient`.

The Jena rule `nurse_in_same_ward_as_patient` defines the context condition test for the context constraint *nurse_in_same_ward_as_patient*.

The rule `nurse_in_same_ward_as_patient` moves CONTEXT_CONDITION_POTENTIAL individual *?x* to CONTEXT_CONDITION if:

- *?x* has is linked to the *nurse_in_same_ward_as_patient* context constraint, i.e., *?x* has *context_constraint* property *nurse_in_same_ward_as_patient*;
- *?x* has user *?nurse* and object *?patient*;
- *?nurse* is linked to individual *?ward* via property *nurse_ward*, and
- *?patient* is linked to the same individual *?ward* via property *patient_ward*.

Figure 89 is converted into Jena syntax in Text 49 above.

When the context constraint *nurse_in_same_ward_as_patient* is run, the following needs to happen:

1. `context_constraint_applied` is run, moving all CONTEXT_CONSTRAINT_APPLICABLE individuals for the context constraint *nurse_in_same_ward_as_patient*, representing roles SENIOR_NURSE and junior roles to read and write to objects in the PATIENT class, to CONTEXT_CONSTRAINT_APPLIED.
2. The rule `nurse_in_same_ward_as_patient` is run, moving all individuals in CONTEXT_CONDITION_POTENTIAL for the context constraint *nurse_in_same_ward_as_patient*, individuals representing users in roles SENIOR_NURSE and junior roles to read and write to objects in the PATIENT class to CONTEXT_CONDITION.
3. The rule `context_condition_pass_1` is run, moving all CONTEXT_CONDITION individuals for the context constraint *nurse_in_same_ward_as_patient* to CONTEXT_CONDITION_PASS.
4. The rule `context_condition_pass_2` is run, moving all CONTEXT_CONDITION individuals for which the context constraint *nurse_in_same_ward_as_patient* does *not* apply to CONTEXT_CONDITION_PASS.

In ESO-RBAC, a context condition rule always applies to an individual *?x* in class CONTEXT_CONDITION_POTENTIAL, and runs a test to determine whether to move *?x* to the class CONTEXT_CONDITION, based on *?x* having as its *context_constraint* property the individual in the class CONTEXT_CONSTRAINT that specifies this context constraint.

Finally, the Jena rules `authorizable` and `authorized` are modified so that an individual must be in CONTEXT_CONDITION_PASS to be in the AUTHORIZABLE and AUTHORIZED classes. The condition `(?x rdf:type rbac:CONTEXT_CONDITION_PASS)` is added to both rules.

The new `5_authorizable`, given in Figure 90, defines AUTHORIZABLE as the intersection of PERMITTABLE, NOT_DENIED and CONTEXT_CONDITION_PASS: an individual is moved to AUTHORIZABLE if it is in all three of PERMITTABLE, NOT_DENIED and CONTEXT_CONDITION_PASS (Formula 12).

PERMITTABLE individual *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED and in CONTEXT_CONDITION_PASS. Note that this means that the same actual individual *?x* has to be in all three of PERMITTABLE, NOT_DENIED and CONTEXT_CONDITION_PASS (not different individual with the same object properties).

$$\text{AUTHORIZABLE} = \text{PERMITTABLE} \cap \text{NOT\_DENIED} \cap \text{CONTEXT\_CONDITION\_PASS}$$

Formula 12: Definition of AUTHORIZABLE.

Figure 90 is converted into Jena syntax in Text 50 below.

```
[5_authorizable: (?x rdf:type rbac:AUTHORIZABLE )
            <-
            (?x rdf:type rbac:PERMITTABLE )
            (?x rdf:type rbac:NOT_DENIED )
            (?x rdf:type rbac:CONTEXT_CONDITION_PASS )
]
```

Text 50: Jena for new rule `5_authorizable`.

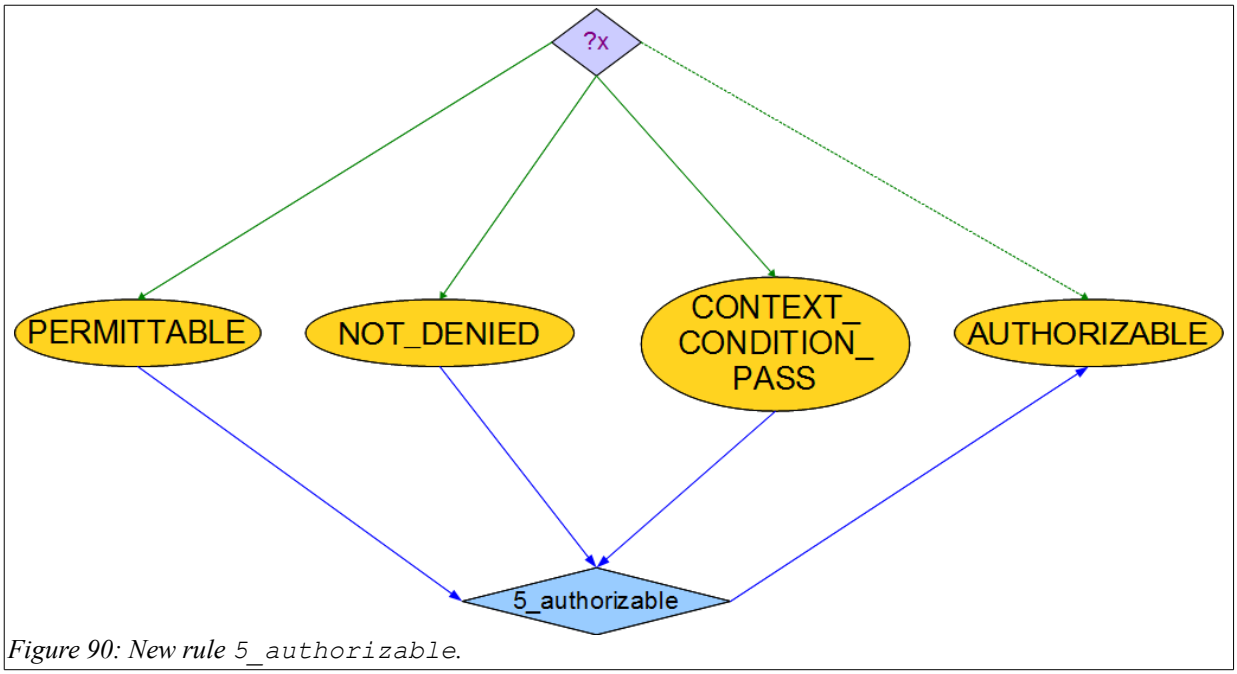

Figure 90: New rule `5_authorizable`.

The new `5_authorized`, given in Figure 91, defines AUTHORIZED as the intersection of PERMITTED, NOT_DENIED and CONTEXT_CONDITION_PASS: an individual is moved to AUTHORIZED if it is in all three of PERMITTED, NOT_DENIED and CONTEXT_CONDITION_PASS (Formula 13).

PERMITTED instance *?x* is moved to AUTHORIZABLE if *?x* is also in NOT_DENIED.

$$AUTHORIZED = PERMITTED \cap NOT\_DENIED \cap CONTEXT\_CONDITION\_PASS$$

Formula 13: Definition of AUTHORIZED.

Figure 91 is converted into Jena syntax in Text 51 below.

```
[5_authorized: (?x rdf:type rbac:AUTHORIZED )
            <-
            (?x rdf:type rbac:PERMITTED )
            (?x rdf:type rbac:NOT_DENIED )
            (?x rdf:type rbac:CONTEXT_CONDITION_PASS )
]
```

Text 51: Jena for new rule `5_authorized`.

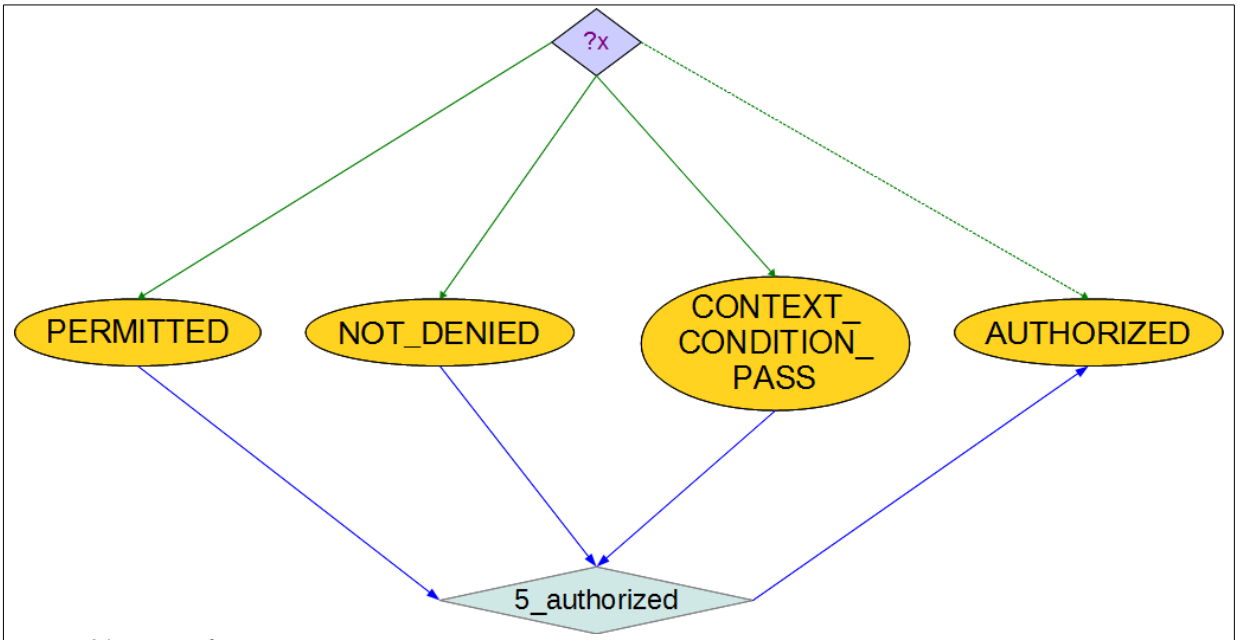

Figure 91: New rule `5_authorized`.

Because the context condition rules do not depend on any of the classes populated in steps 2–4, they can be run at any point after step 1 and before step 5.

6.5 Contrasting ESO-RBAC with SO-RBAC and with Prolog

Due to differences between Jena and SWRL, some rules in Steps 1 and 4 were implemented differently in ESO-RBAC from in SO-RBAC.

The main advantage of Jena over SWRL in implementing ESO-RBAC is its ability to treat classes as individuals. However, it has certain flaws. Unlike SWRL, it cannot work on inferred axioms. This means that it cannot identify an individual as belonging to a sub-class of a class, and nor can it see relationships defined for sub-properties. Therefore, certain properties have to be defined explicitly in ESO-RBAC, so that it can be run in Jena, when it is unnecessary to do so for running in SWRL.

Thus, ESO-RBAC does not have the *is_a* property for roles, as this is represented by sub-classing roles. Additionally, although Jena cannot natively infer recursive sub-class or super-class relationships, additional rules have been defined in ESO-RBAC to handle this. Therefore, *included_in* is also not used.

Most Jena rules in ESO-RBAC are direct transformations of the SWRL rules in SO-RBAC. The major difference is in the properties that link the individuals, and that *?r* is a class, queried as an individual. The antecedent of `1_included_in_1`, instead of requiring *?r* to a member of class ROLE, requires it to be a member of ROLE_SET. Note that the RDF property *rdf:type* defines an individual as a member of a class. The syntax `?r rdf:type rbac:ROLE_SET` in Jena is equivalent to rbac:ROLE_SET(?r) in SWRL. `1_included_in_1` has an additional condition, `notEqual(?r, rbac:ROLE)`. ROLE is the top level of the hierarchy of classes representing roles. It is placed in the meta-class ROLE_SET so that any classes added immediately below ROLE are also added to ROLE_SET. However, this means that ROLE itself would be treated as a role by Jena. To prevent this, the class ROLE must be explicitly excluded from the reasoning process.

In the recursive rule `1_included_in_3`, the *is_a* condition is replaced by determining whether *?r1* is a sub-class of *?r2*. This is done using the RDF property *rdfs:subClassOf*.

Jena also does not populate inferences based on inverse relationships. Therefore, an additional rule `1_junior_to` is defined in Step 1 to assert *junior_to* axioms as inverses of corresponding *senior_to* axiom.

The other difference is in Step 4, with the rule to populate NOT_DENIED. Jena does not use SQWRL properties, but has different syntax for achieving classical negation, namely the built-in function `noValue`. The rule is thus as given in Text 52.

```
[4_not_denied: (?x rdf:type rbac:NOT_DENIED )
    <-
    (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
    noValue(?x rdf:type rbac:DENIED )
]
```

Text 52: Jena rule for populating NOT_DENIED.

Therefore, Jena rule `4_not_denied` is, in terms of syntax, similar to an equivalent rule in Prolog, rather than to the equivalent SWRL rule. Unlike in SO-RBAC using SWRL, the class DENIED does not need a dummy individual.

Table 20 shows the correspondences between Prolog functions and ESO-RBAC classes and properties.

Table 20: Correspondences between Prolog functions and ESO-RBAC classes and properties.

Prolog	***ESO-RBAC***	***Comments***
`login_session(SessionID,User,IP,Start_ Date,End_Date,Authentication_Strength, LocationType,Computer,IP,OSLogin).`	USER_SESSION	
`user(Username,LastName,FirstName,Address, DOB).`	PERSON, USER	
`ura(User,Role).`	(assignment of instance of USER to ROLE)	
`d_s(Senior_role,Junior_role).`	*senior_to*	*junior_to* is inverse property. Transitive.
`is_a(Inner_Role,Outer_Role).`	(assignment of Inner_Role as a subclass of Outer_Role)	
`pra(Role,Action,Object).`	PRA	
`dra(Role,Action,Object).`	DRA	
`associated_cc(Role,Permission,Object, ContextConstraint).`	ASSOCIATED_CC	

6.6 Implementing ESO-RBAC based on a hospital environment

The ESO-RBAC implementation is illustrated through a scenario with roles, permissions, denials, seniority relationships, inclusion relationships and inheritance paths.

- *ADMIN*, *CLERK*, *MANAGER*
- *DOCTOR*, *SPECIALIST_DOCTOR*, *CONSULTANT*, *JUNIOR_STAFF_DOCTOR*, *JUNIOR_STAFF_DOCTOR_DAY*, *JUNIOR_STAFF_DOCTOR_NIGHT*, *SENIOR_STAFF_DOCTOR*, *SENIOR_STAFF_DOCTOR_DAY*, *SENIOR_STAFF_DOCTOR_NIGHT*
- *TECHNICIAN*, *JUNIOR_TECHNICIAN*, *SENIOR_TECHNICIAN*
- *NURSE*, *SENIOR_NURSE*, *SPECIALIST_NURSE*, *STAFF_NURSE*, *STAFF_NURSE_DAY*, *STAFF_NURSE_NIGHT*, *STUDENT_NURSE*, *STUDENT_NURSE_DAY*, *STUDENT_NURSE_NIGHT*

Text 53: Sub-classes of ROLE *defined as individuals in class* ROLE_SET *in the ESO-RBAC model.*

Text 53 lists the classes (sub-classes of ROLE) were defined as individuals in class ROLE_SET, reflecting a simplified hospital scenario.

- PRA: *junior_staff_doctor_read_patient*, *junior_staff_doctor_read_room*, *junior_staff_doctor_read_vital_sign*, *junior_staff_doctor_read_ward*, *senior_staff_doctor_write_patient*, *senior_staff_doctor_write_room*, *senior_staff_doctor_write_vital_sign*, *consultant_write_vital_sign*, *consultant_read_computer*, *specialist_doctor_write_computer*, *student_nurse_read_patient*, *staff_nurse_read_room*, *staff_nurse_read_ward*, *staff_nurse_write_patient*, *senior_nurse_read_vital_sign*, *senior_nurse_write_ward*, *specialist_nurse_read_computer*, *specialist_nurse_write_room*, *specialist_nurse_write_vital_sign*, *specialist_nurse_write_computer*
- DRA: *consultant_read_room*, *consultant_write_ward*, *senior_nurse_read_ward*, *senior_staff_doctor_read_computer*, *staff_nurse_write_patient*

Text 54 Individuals representing permission and denial assertions in the ESO-RBAC model.

Text 54 lists the individuals representing permission and denial assertions in the ESO-RBAC model.

1. *DOCTOR*: *JUNIOR_STAFF_DOCTOR* → *SENIOR_STAFF_DOCTOR* → *CONSULTANT* → *SPECIALIST_DOCTOR*
2. *NURSE*: *STUDENT_NURSE* → *STAFF_NURSE* → *SENIOR_NURSE* → *SPECIALIST_NURSE*
3. *TECHNICIAN*: *JUNIOR_TECHNICIAN* → *SENIOR_TECHNICIAN*
4. *ADMIN*: *CLERK* → *MANAGER*
5. *JUNIOR_STAFF_DOCTOR: JUNIOR_STAFF_DOCTOR_DAY, JUNIOR_STAFF_DOCTOR_NIGHT*
6. *SENIOR_STAFF_DOCTOR: SENIOR_STAFF_DOCTOR_DAY, SENIOR_STAFF_DOCTOR_NIGHT*
7. *STUDENT_NURSE: STUDENT_NURSE_DAY, STUDENT_NURSE_NIGHT*
8. *STAFF_NURSE: STAFF_NURSE_DAY, STAFF_NURSE_NIGHT*

Text 55: Seniority relationships in the ESO-RBAC model.

Text 55 shows the role hierarchies indicated by the seniority relationships defined using *directly_senior_to* axioms.

1. inherits_pra_path(SPECIALIST_DOCTOR, JUNIOR_STAFF_DOCTOR)
2. inherits_pra_path(SPECIALIST_NURSE, STUDENT_NURSE)

Text 56: Path inheritance axioms in the ESO-RBAC model.

Text 56 shows the path inheritance axioms defined in the ESO-RBAC model.

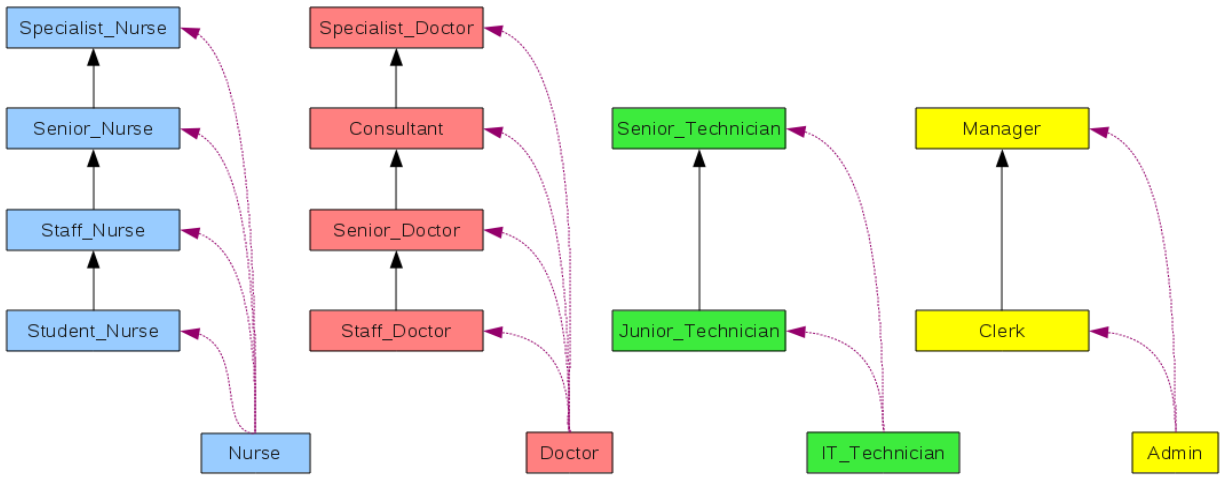

Figure 92: RBAC Model used to demonstrate SO-RBAC, excluding night and day duties. Solid (black) lines represent seniority (`d_s`) relationships. Dashed (purple) lines represent `is_a` relationships. Arrows show direction of inheritance of positive authorizations (permissions).

Figure 92 shows the full RBAC hierarchy.

Table 21: Numbers of users in each role defined in the ESO-RBAC ontologies.

Role	***Small***	***Large***
CLERK	1	2
MANAGER	1	1
JUNIOR_STAFF_DOCTOR	3	4
SENIOR_STAFF_DOCTOR	3	4
CONSULTANT	1	2
SPECIALIST_DOCTOR	1	1
STUDENT_NURSE	3	4
STAFF_NURSE	3	4
SENIOR_NURSE	1	2
SPECIALIST_NURSE	1	1
JUNIOR_TECHNICIAN	1	2
SENIOR_TECHNICIAN	1	1

One or more USER individuals for each ROLE was created (Table 21), except for the roles *ADMIN*, *DOCTOR*, *TECHNICIAN* and *NURSE*, as these are intended as abstract super-class roles to allow permissions to be defined for a particular type of user generically.

User individuals are named simply as *<role>_<n>*, where *<n>* is a number. The roles with day and night sub-roles defined each had 3 or 4 users defined, named for the main role. For example, *junior_staff_doctor_1* was assigned directly to *JUNIOR_STAFF_DOCTOR*; *junior_staff_doctor_2* to *JUNIOR_STAFF_DOCTOR_DAY*, and *junior_staff_doctor_3* to *JUNIOR_STAFF_DOCTOR_NIGHT.* No personalized data were defined for any of these users, because they are not relevant in this static RBAC model.

The users were linked to roles by assignment of the user individuals as members of the relevant ROLE sub-classes.

Instances were created for object types (classes) *COMPUTER*, *PATIENT*, *ROOM*, *VITAL_SIGN* and WARD. One instance of each type was created for the small scenario, and three of each type for the large scenario. The instances were named *<object_name>_n*, e.g. *patient_1*.

6.7 Results of Implementation

The ontological model was implemented using the Protégé Ontology Editor, using the Protégé-OWL plugin. The Pellet Reasoner Inspector was used to test the consistency of the ontology's classes, properties and instances.

The classes ROLE_PERMISSION_ASSIGNABLE and USER_PERMISSION_ASSIGNABLE were populated with individuals representing all possible permutations of roles, users and permissions using a Perl script, which also added to PRA and DRA the individuals listed above.

There is no plug-in for Jena in Protégé. Therefore, Jena rules were defined in plain text files, to be run on the command line using a Jena engine in Java, and running it. The rules for each step were defined in a separate Jena file. A Unix shell script was written to execute all 5 steps in turn. The resulting OWL files (one for each stage) were then examined in Protégé, and OWL *n*-triple files were created from them, to check that they ran correctly. The results of the examination of the *n*-triple files are presented. The same models were run in ESO-RBAC as in SO-RBAC.

The numbers of triples of affected classes and properties at each stage were determined by exporting the OWL files as *n*-triple files, and analyzing these using the Unix shell tool `grep`. Jena produces no reports of numbers of classes, individuals and axioms, so these are not provided.

Table 22: Numbers of rules run and triples obtained by Jena for each ontology.

	Step 1	***Step 2***	***Step 3***	***Step 4***	***Step 5***
Jena rules exported to Jess	8	2	2	2	2
Unique triples created (small ontology)	126	62	131	166	98
Unique triples created (large ontology)	126	62	423	477	306

Table 23: Numbers of triples at stage 1.

Property	***Small***	***Large***
senior_to	26	26
junior_to	26	26
included_in	62	62
inherits_pra	49	49
Total	***163***	***163***

Table 22 shows the numbers of rules run and triples created by Jena in each step for each ontology. Note that ESO-RBAC has one more rule than SO-RBAC in Step 1 (`1_junior_to`, described in Section 6.2.3.1, page 127).

The same numbers of triples were found for both ontologies, because Step 1 only operates on roles, and both have the same roles (Table 23). Unlike in SO-RBAC, a rule for inferring *junior_to* axioms (as the reverse of *senior_to*

axioms) in rule was defined (`1_junior_to`, described in Section 6.2.3.1, page 127), because Jena does not automatically create corresponding axioms for inverse properties. The numbers of *included_in* and *inherits_pra* individuals were slightly larger than in SO-RBAC due to differences in the model used (the model used for ESO-RBAC implemented the DAY_DUTY and NIGHT_DUTY roles, whereas the SO-RBAC model did not).

Table 24: Numbers of triples at stage 2.

Class	***Small***	***Large***
PRA_FULL	49	49
DRA_FULL	13	13
Total	***62***	***62***

Table 25: Numbers of triples at stage 3.

Class	***Small***	***Large***
PERMITTABLE	95	300
DENIED	36	123
Total	***131***	***423***

At Stage 2, the same triples were found in ESO-RBAC as in SO-RBAC (Table 24).

At Stage 3, the numbers of individuals in DENIED are one less than in SO-RBAC, due to the lack of the dummy individual used in SO-RBAC (Table 25).

Table 26: Numbers of triples in stage 4.

Class	***Small***	***Large***
NOT_DENIED	144	477
PERMITTED	22	66
Total	***166***	***543***

Table 27: Numbers of triples in stage 5.

Class	***Small***	***Large***
AUTHORIZABLE	79	249
AUTHORIZED	19	57
Total	***98***	***306***

At Stage 4, the same triples were found in ESO-RBAC as in SO-RBAC (Table 26).

At Stage 5, the AUTHORIZABLE and AUTHORIZED classes were populated (Table 27). The same triples were found in ESO-RBAC as in SO-RBAC.

6.8 Results of ESO-RBAC Process in Protégé

This section displays screen shots captured using the Protégé OWLViz tab [117] at various stages of reasoning (Figs. 94–114). All screen shots are taken from the small ontology.

6.8.1 Classes and Individuals

6.8.1.1 General

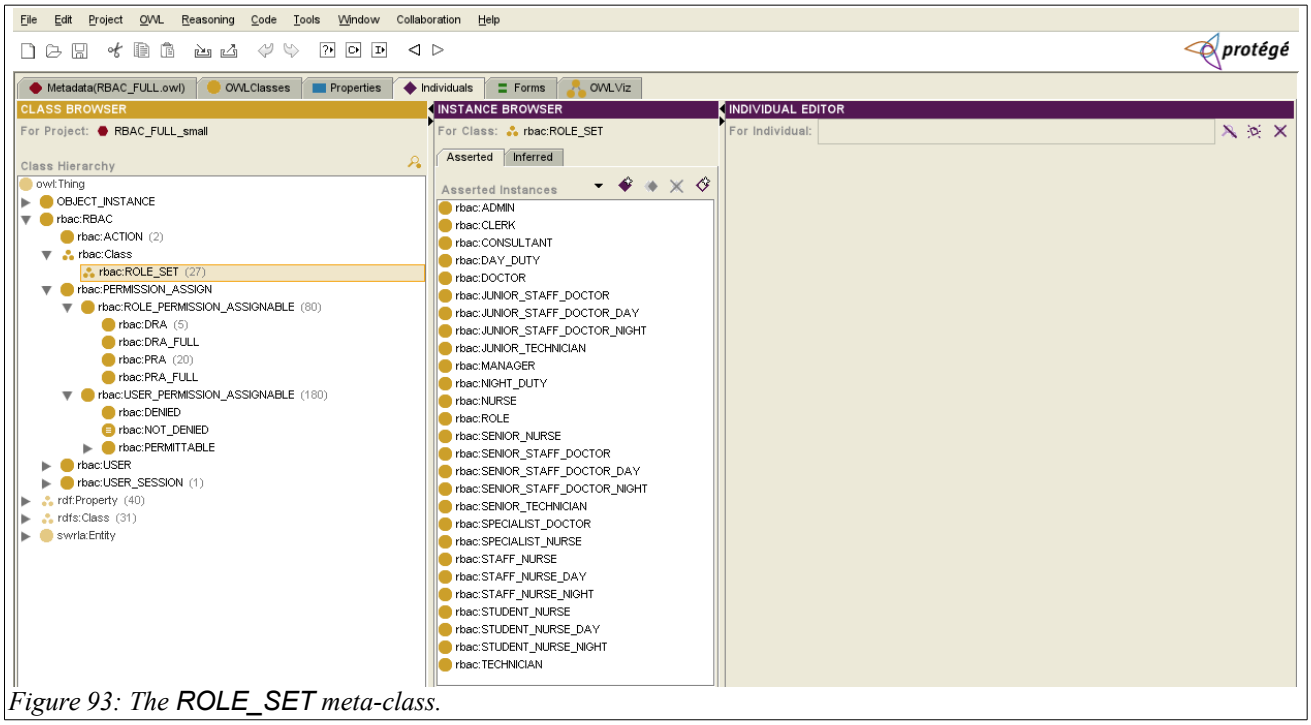

Figure 93: The ROLE_SET meta-class.

Figure 93 shows the ROLE_SET meta-class. Meta-classes are represented differently in Protégé from normal classes (using the set of three small dots ( ) rather than the large dot ( )). The asserted instances of ROLE_SET are the classes under ROLE, these show up as classes rather than as individuals in the Asserted tab. It can be seen from the figure that ROLE_SET is a member of rbac:Class, which is a member directly of owl:Class as well as of RBAC.

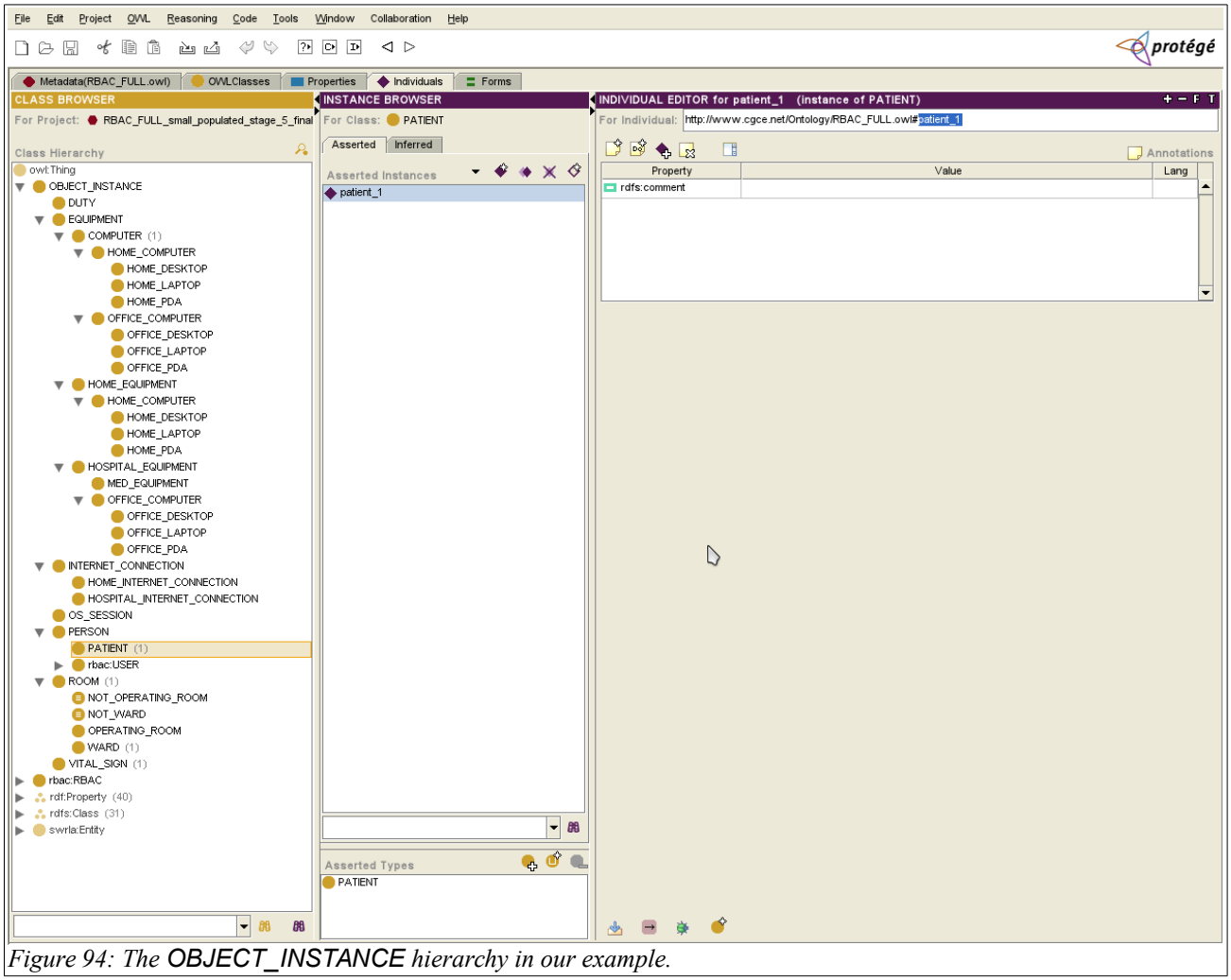

Figure 94: The OBJECT_INSTANCE hierarchy in our example.

Figure 94 shows the OBJECT_INSTANCE hierarchy in ESO-RBAC. The main difference between this and the equivalent in SO-RBAC is that here there is no *object_instance* property, because this is represented by class membership. (Likewise, there is no OBJECT_TYPE class).

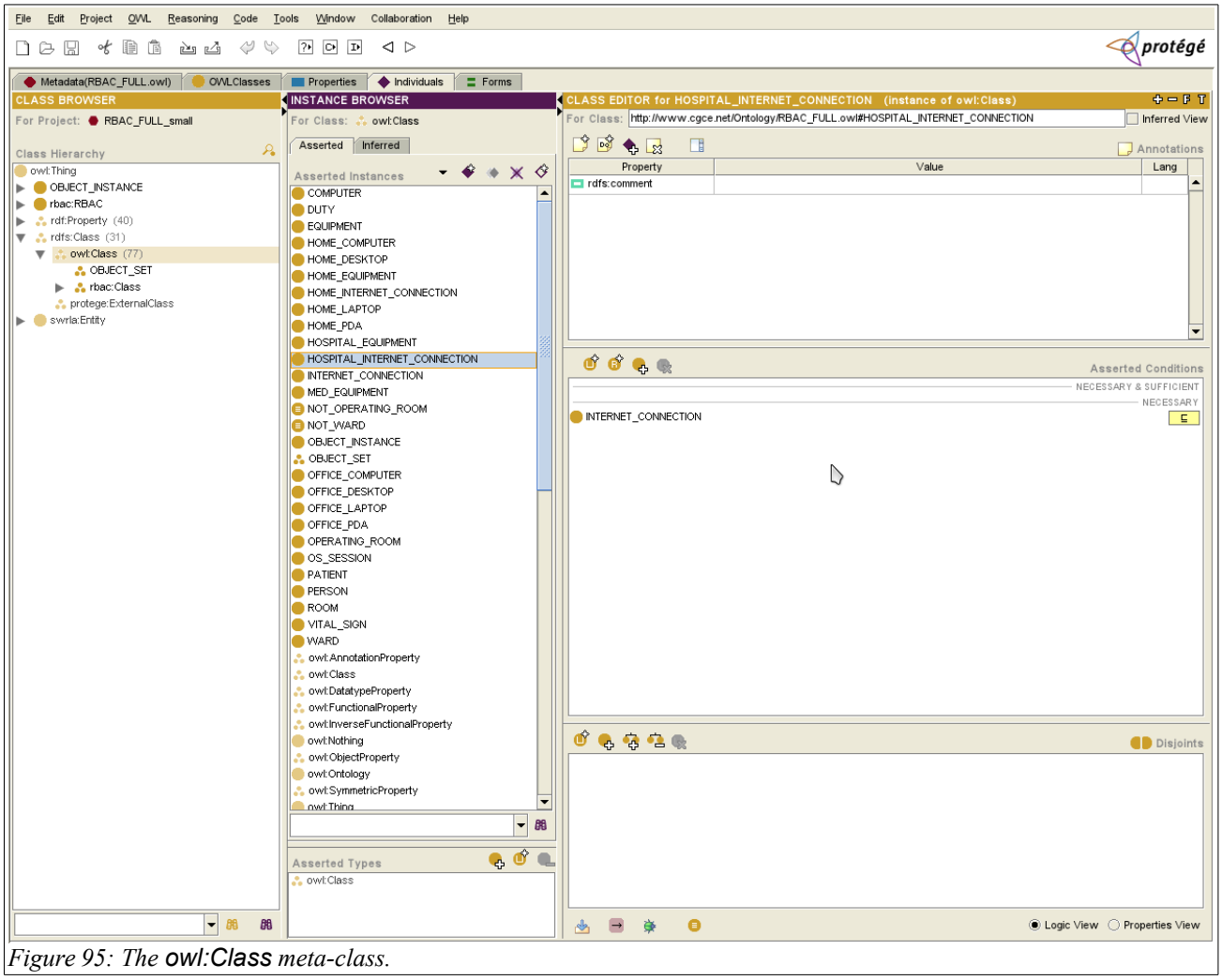

Figure 95: The owl:Class *meta-class.*

Figure 95 shows the owl:Class meta-class, which contains all classes other than those in ROLE_SET.

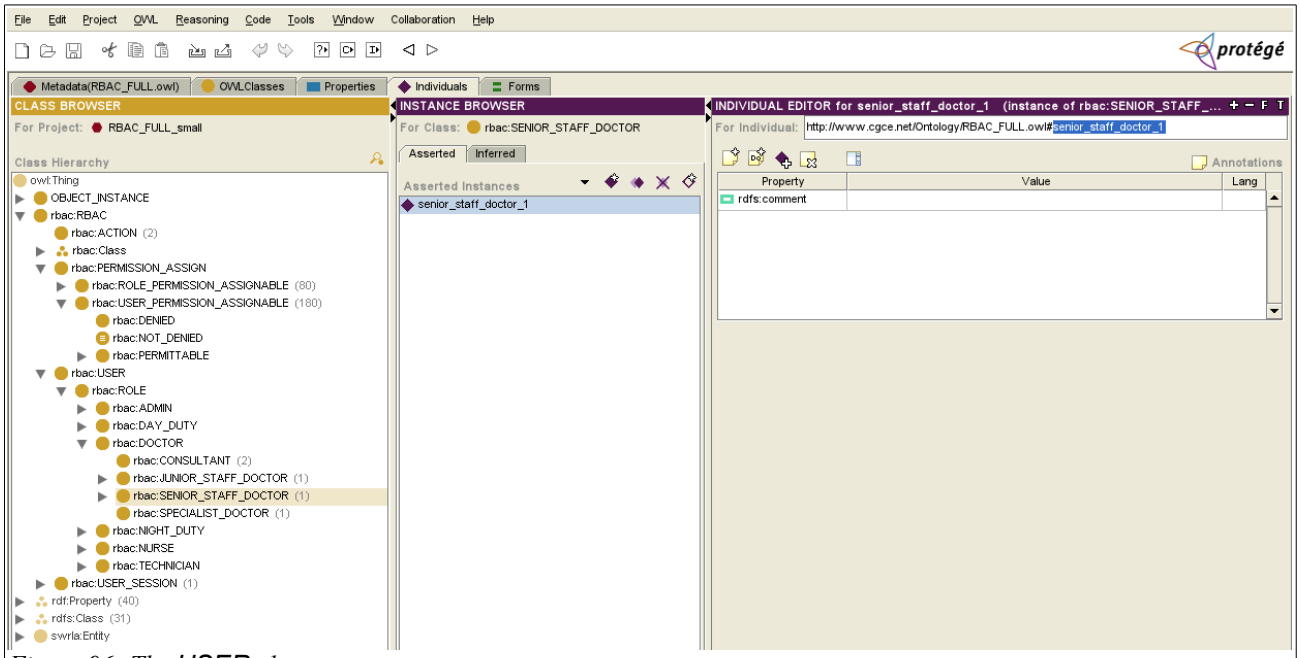

Figure 96: The USER class.

USER, as shown in Figure 96, is now the super-class of ROLE, and USER instances are defined as members of the ROLE classes.

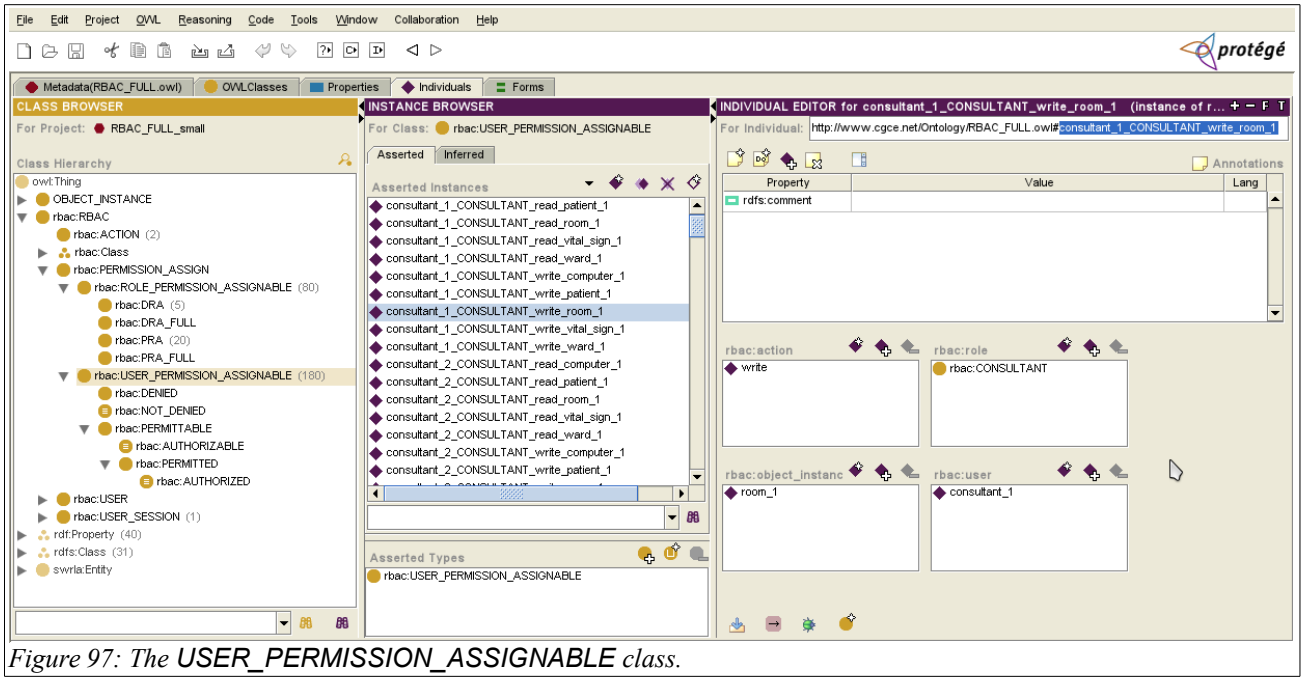

Figure 97: The USER_PERMISSION_ASSIGNABLE class.

USER_PERMISSION_ASSIGNABLE is similar to that in SO-RBAC. However, as shown in Figure 97, the range of rbac:ROLE is now a class (an instance of ROLE_SET) rather than an individual.

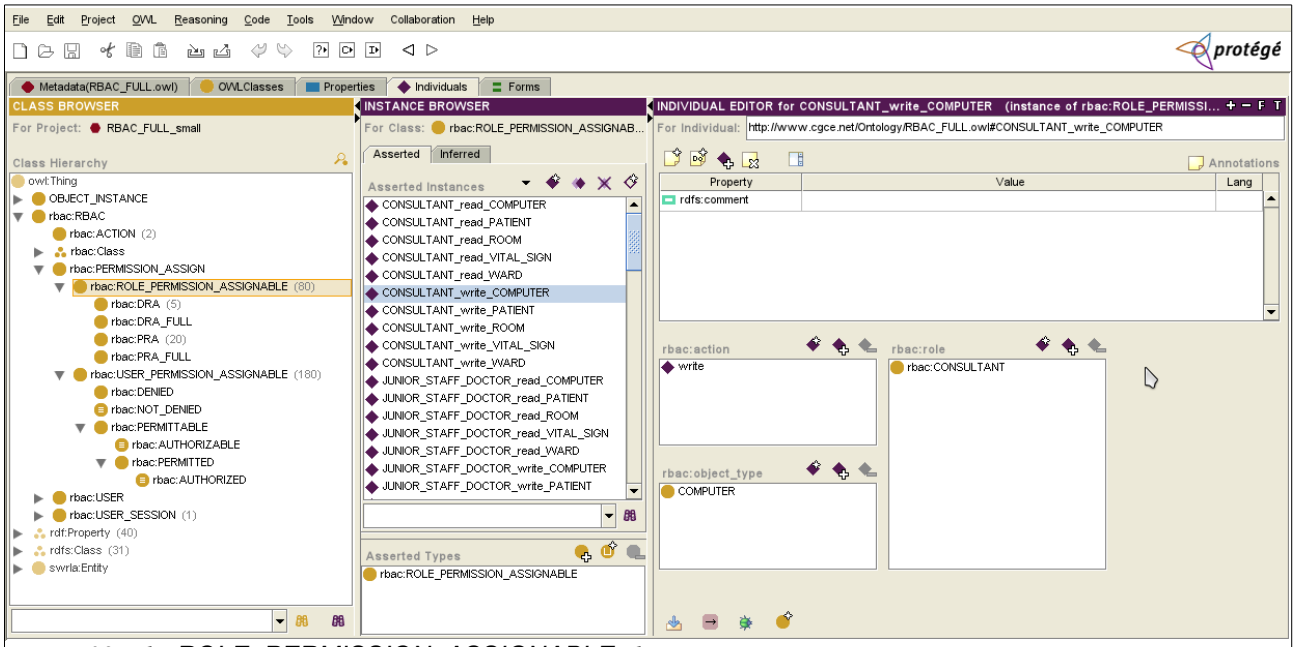

Figure 98: The ROLE_PERMISSION_ASSIGNABLE class.

ROLE_PERMISSION_ASSIGNABLE is, again, similar to that in SO-RBAC (Figure 98). The ***object_type*** and ***role*** properties have classes as their ranges. The range of ***object_type*** is owl:Class; note that as ROLE_SET is a sub-class of this, role classes can also appear as the range of ***object_type***.

6.8.1.2 Initialization

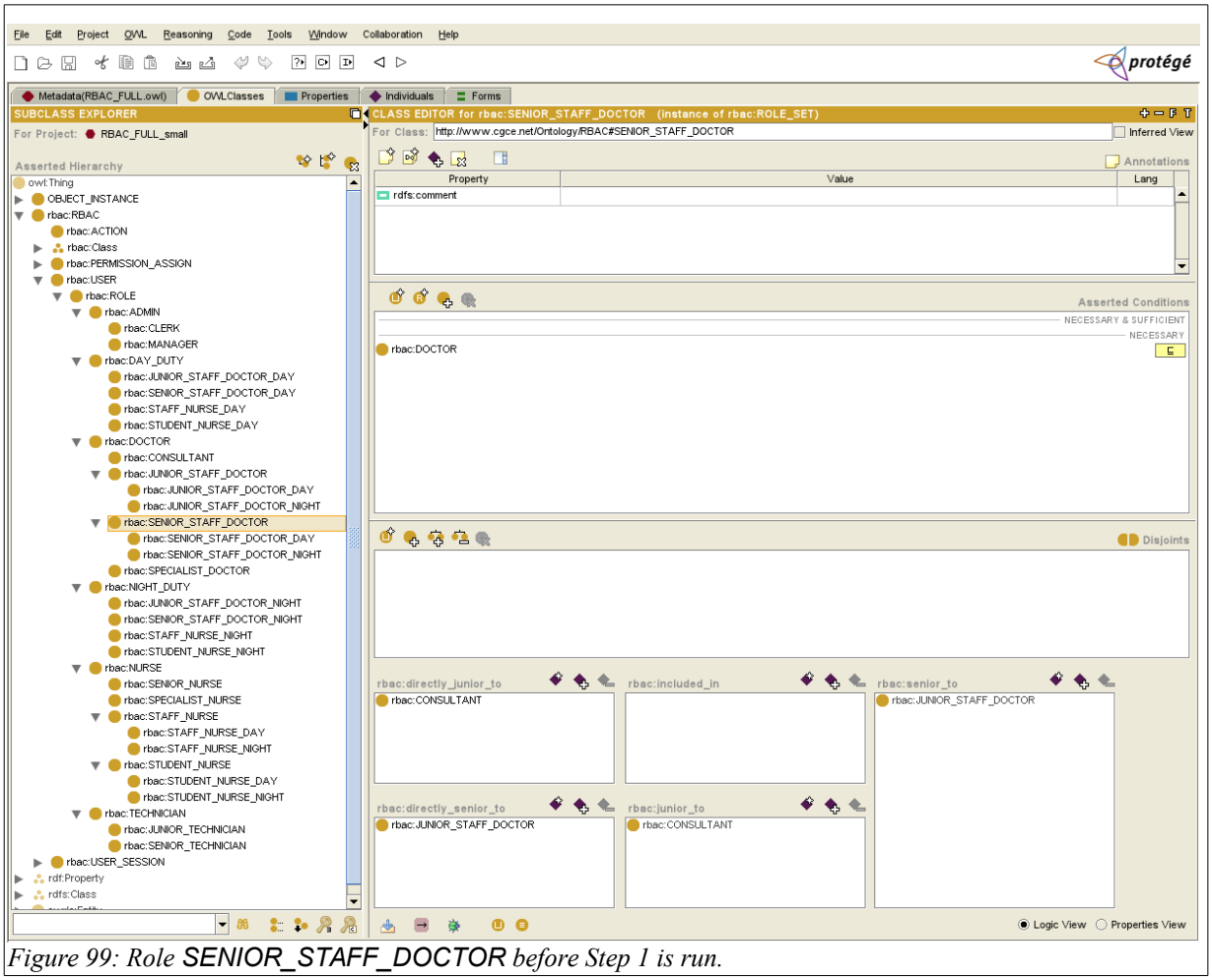

Figure 99: Role SENIOR_STAFF_DOCTOR before Step 1 is run.

Figure 99 shows a ROLE class definition in ESO-RBAC. The screenshot is of the role class definition, rather than that of the canonical individual (which does not exist in ESO-RBAC). (It is also possible to look at a ROLE class as an individual in the meta-class ROLE_SET). All the properties used in roles in SO-RBAC are here, apart from *is_a*, which is represented by super-classing.

6.8.2 Reasoning

6.8.2.1 Stage 1

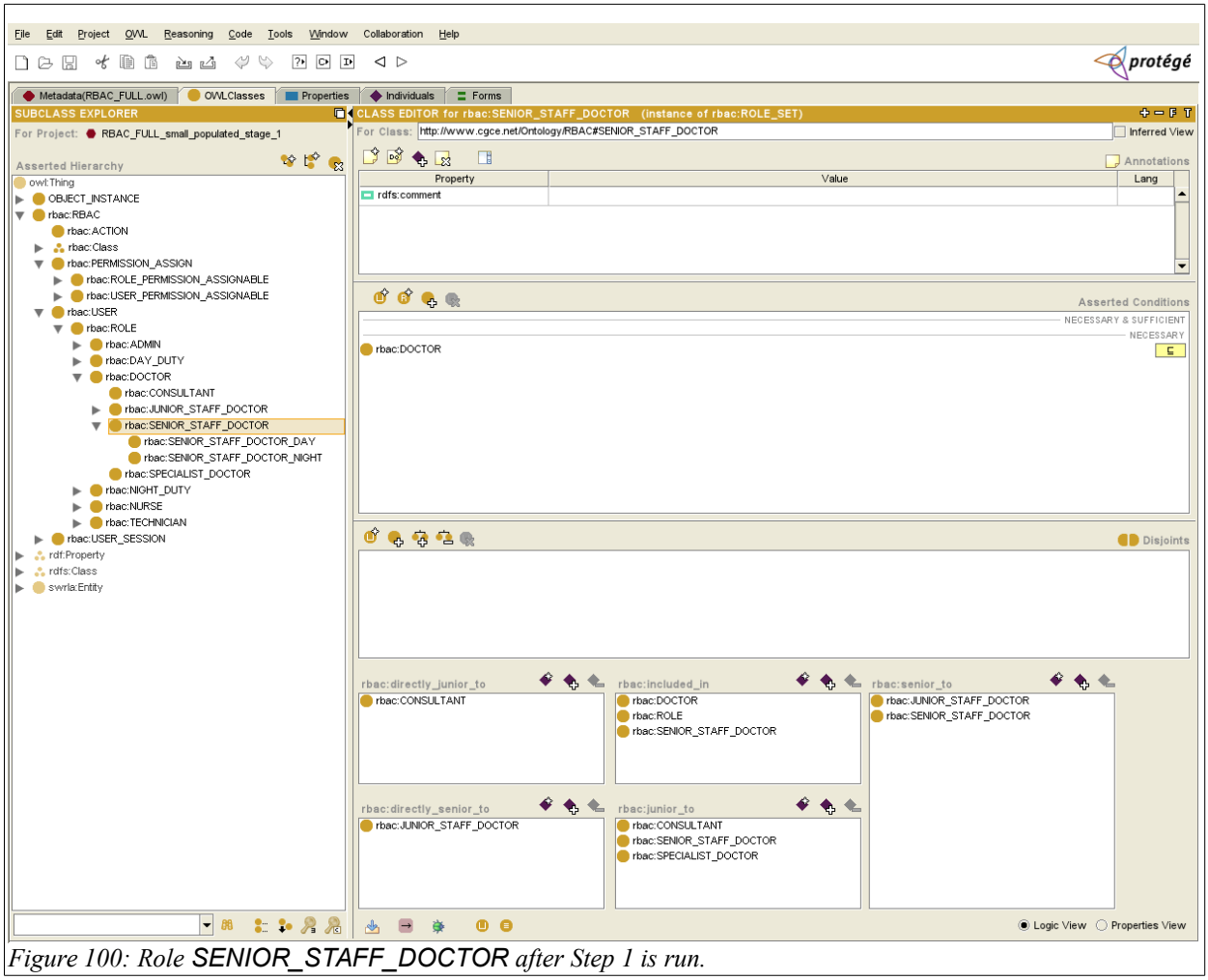

Figure 100: Role SENIOR_STAFF_DOCTOR after Step 1 is run.

Figure 100 shows the role SENIOR_STAFF_DOCTOR after Step 1 is run. *included_in* is now fully populated, by SENIOR_STAFF_DOCTOR being a sub-class of DOCTOR.

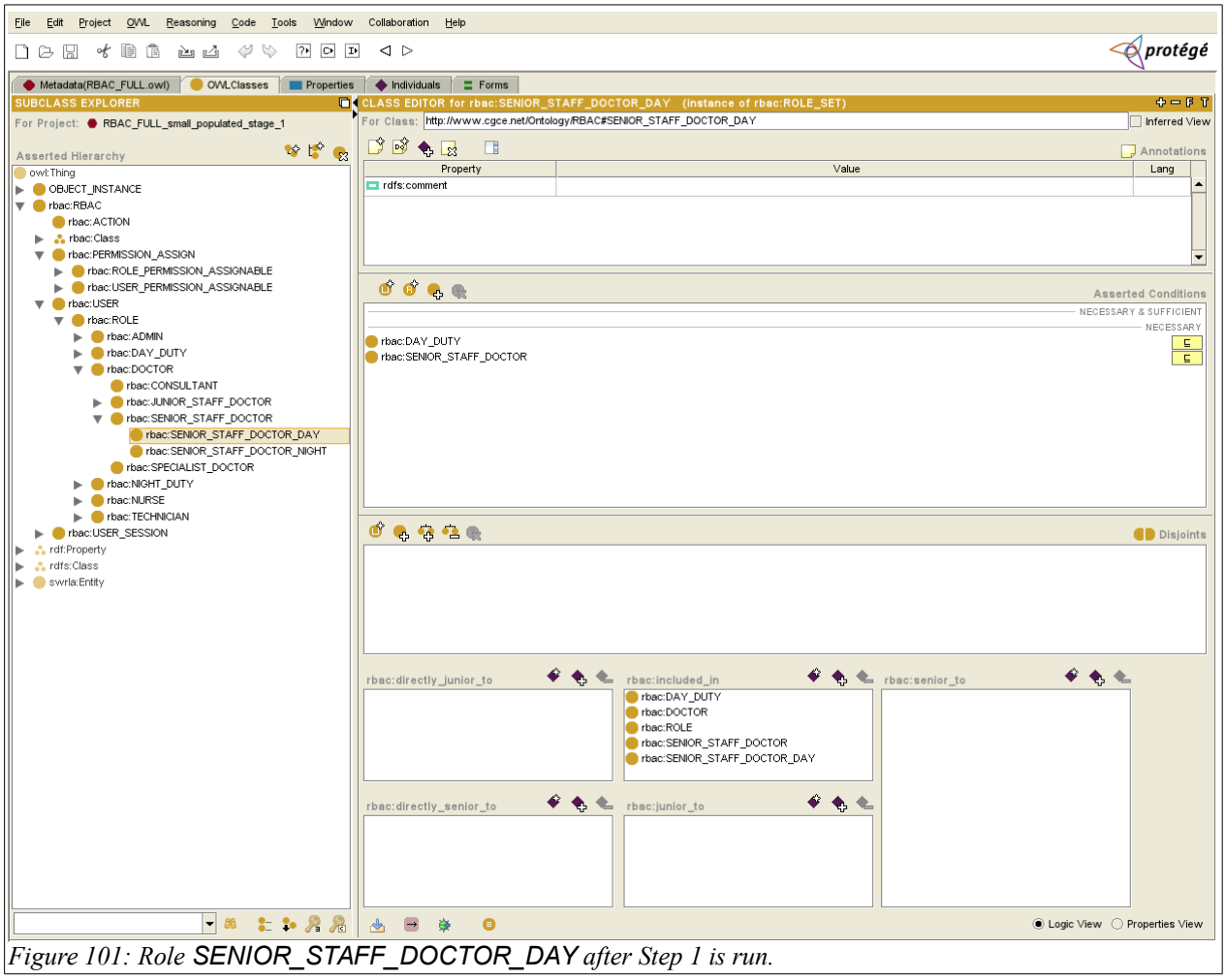

Figure 101: Role SENIOR_STAFF_DOCTOR_DAY after Step 1 is run.

Figure 101 shows the role SENIOR_STAFF_DOCTOR_DAY after Step 1 is run.

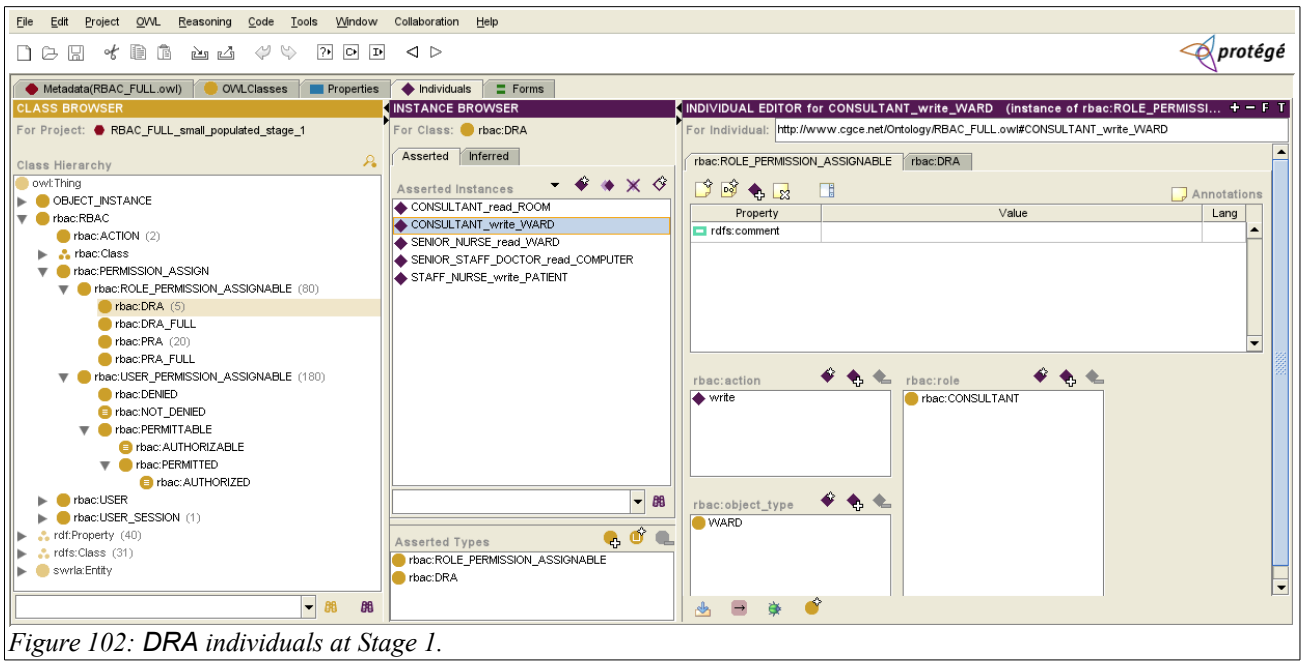

Figure 102: DRA individuals at Stage 1.

Figure 102 shows DRA at Stage 1, containing the individuals with which it is initialized. This is similar to the DRA of SO-RBAC (Figure 52, page 111). However, the figure shows that for the highlighted individual, *CONSULTANT_write_WARD* (as for all individuals in the class DRA), the individuals linked to it via properties *rbac:object_type* and *rbac:role* are classes, not plain individuals.

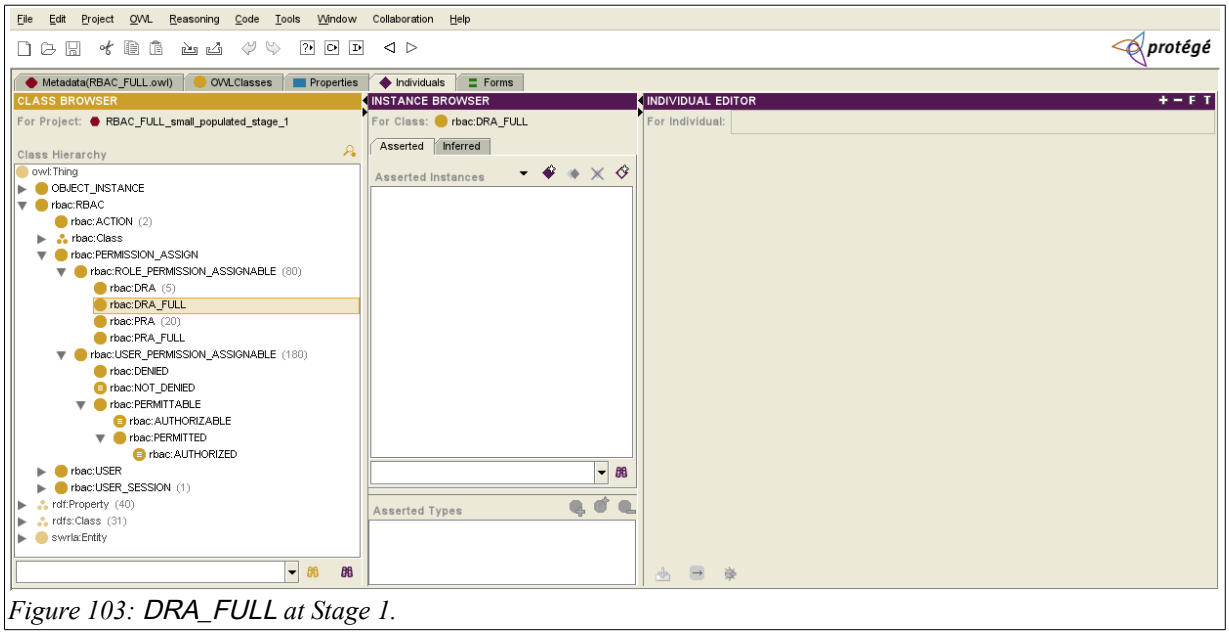

Figure 103: DRA_FULL at Stage 1.

Figure 103 shows DRA_FULL at Stage 1. This class is empty because it has not yet been populated in Step 2.

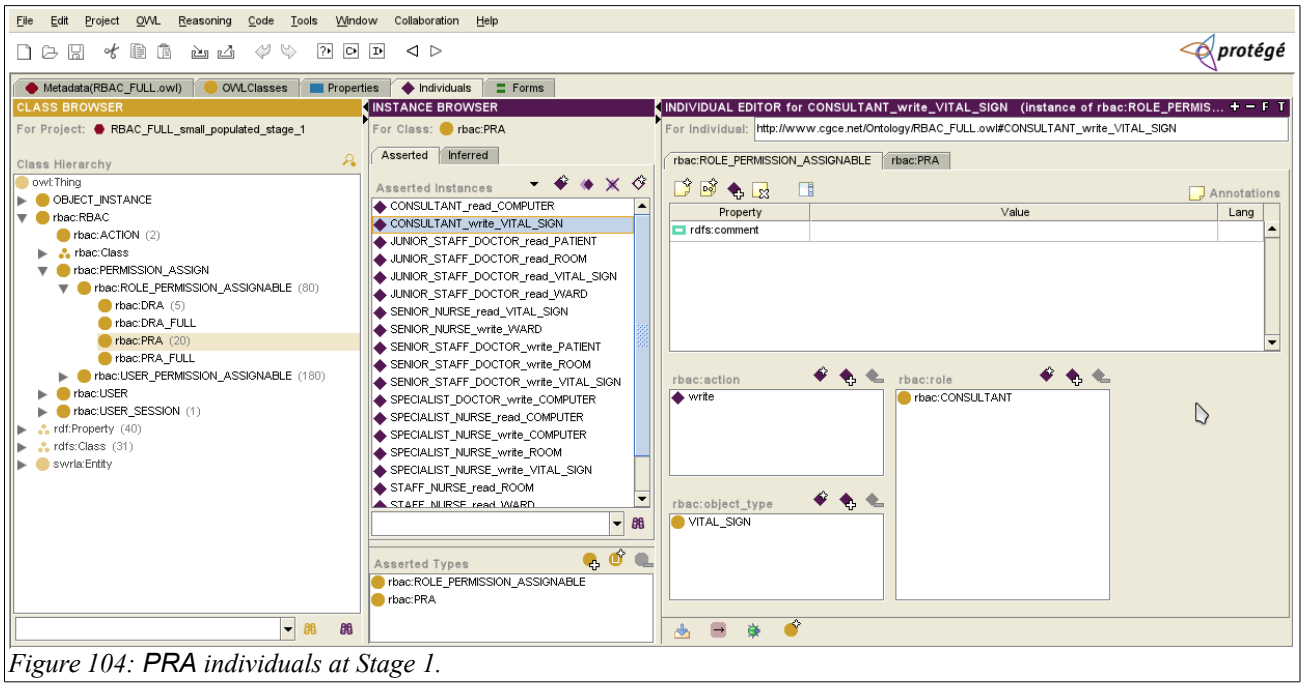

Figure 104: PRA individuals at Stage 1.

Figure 104 shows PRA at Stage 1, containing the individuals with which it is initialized.

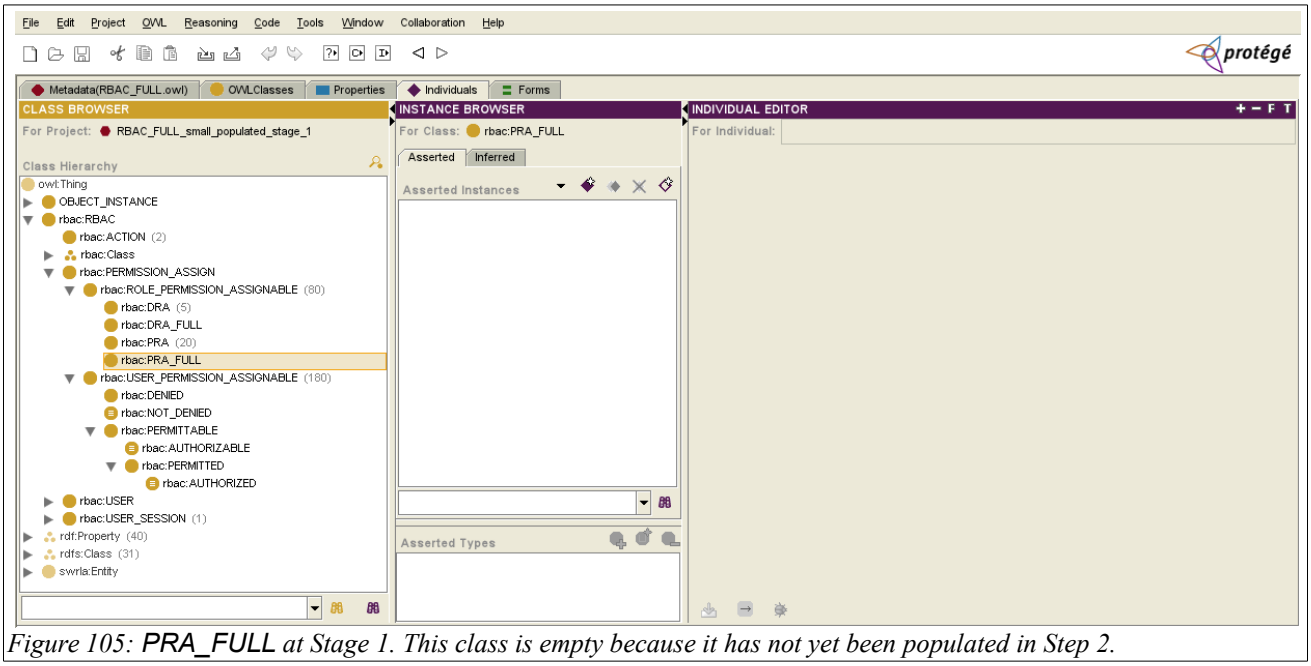

Figure 105: PRA_FULL at Stage 1. This class is empty because it has not yet been populated in Step 2.

Figure 105 shows DRA_FULL at Stage 1. This class is empty because it has not yet been populated in Step 2.

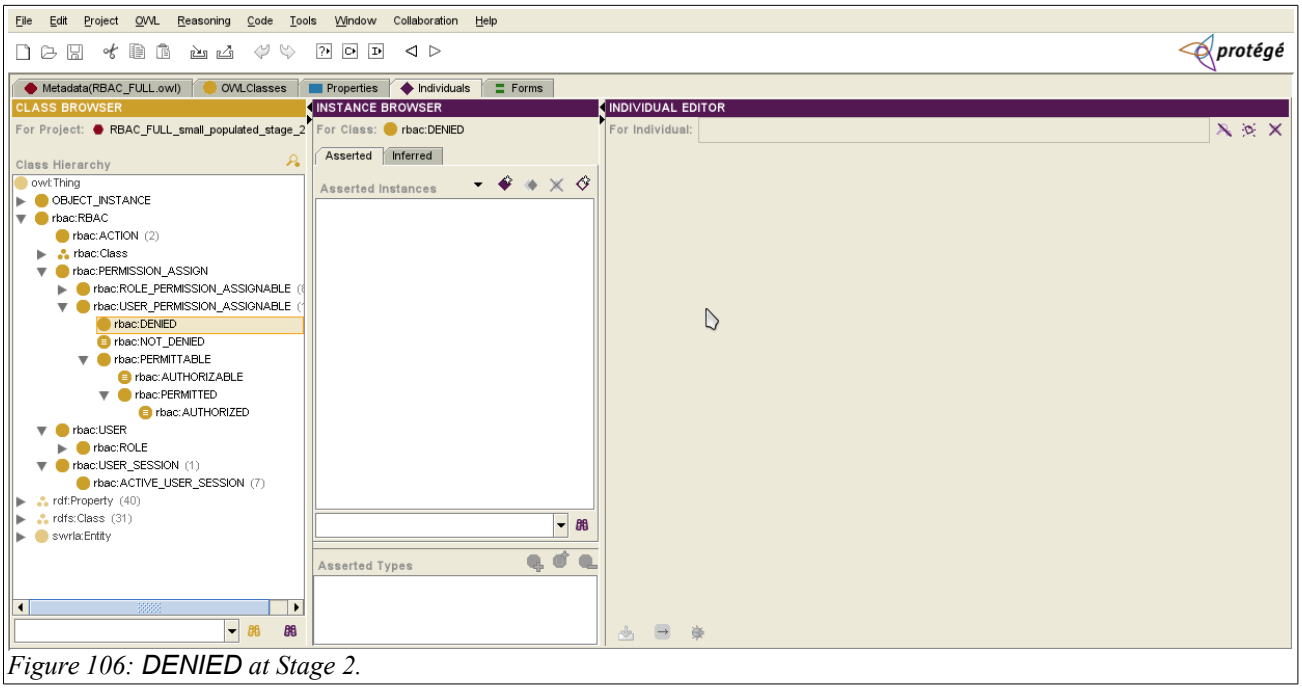

Figure 106: DENIED at Stage 2.

Figure 106 shows DENIED at Stage 2. This does not have the dummy individual that is needed in SO-RBAC (Figure 45, page 107).

6.8.2.2 Stage 2

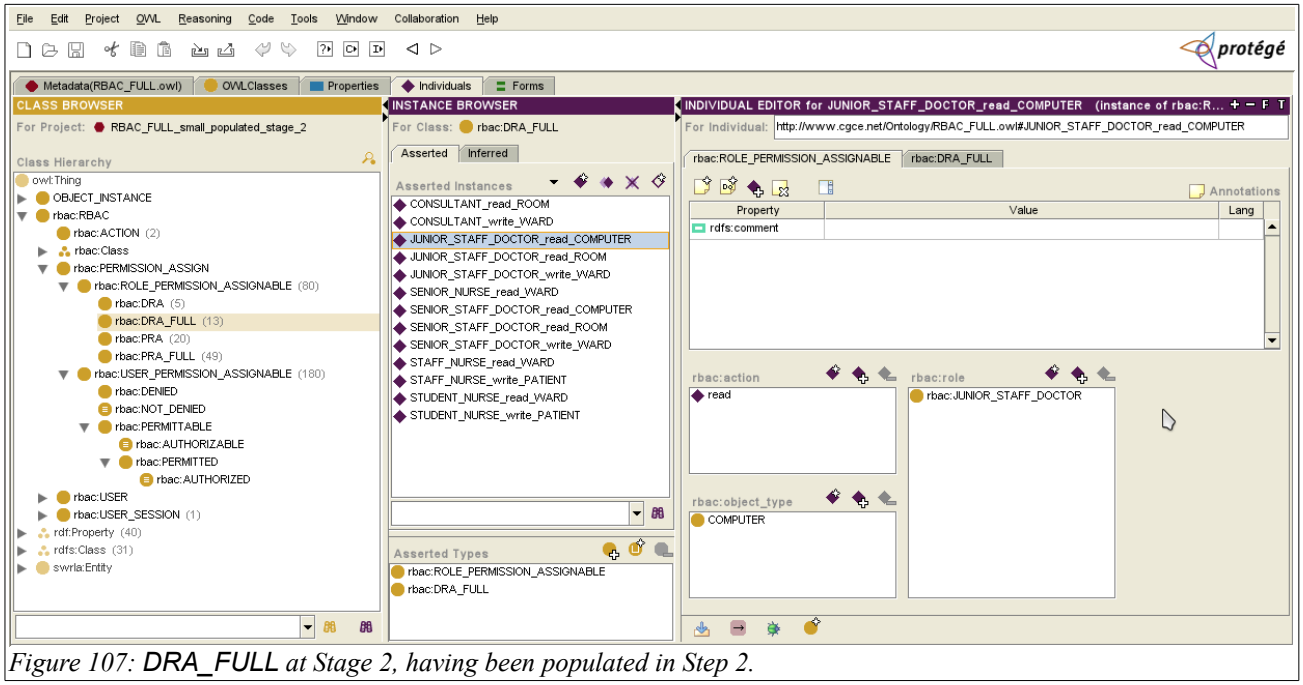

Figure 107: DRA_FULL at Stage 2, having been populated in Step 2.

Figure 107 shows DRA_FULL after it has been populated in Step 2. The individual *JUNIOR_STAFF_DOCTOR_read_COMPUTER* is highlighted. This individual is in DRA_FULL, but not in DRA, because it represents an inferred role-denial assignment.

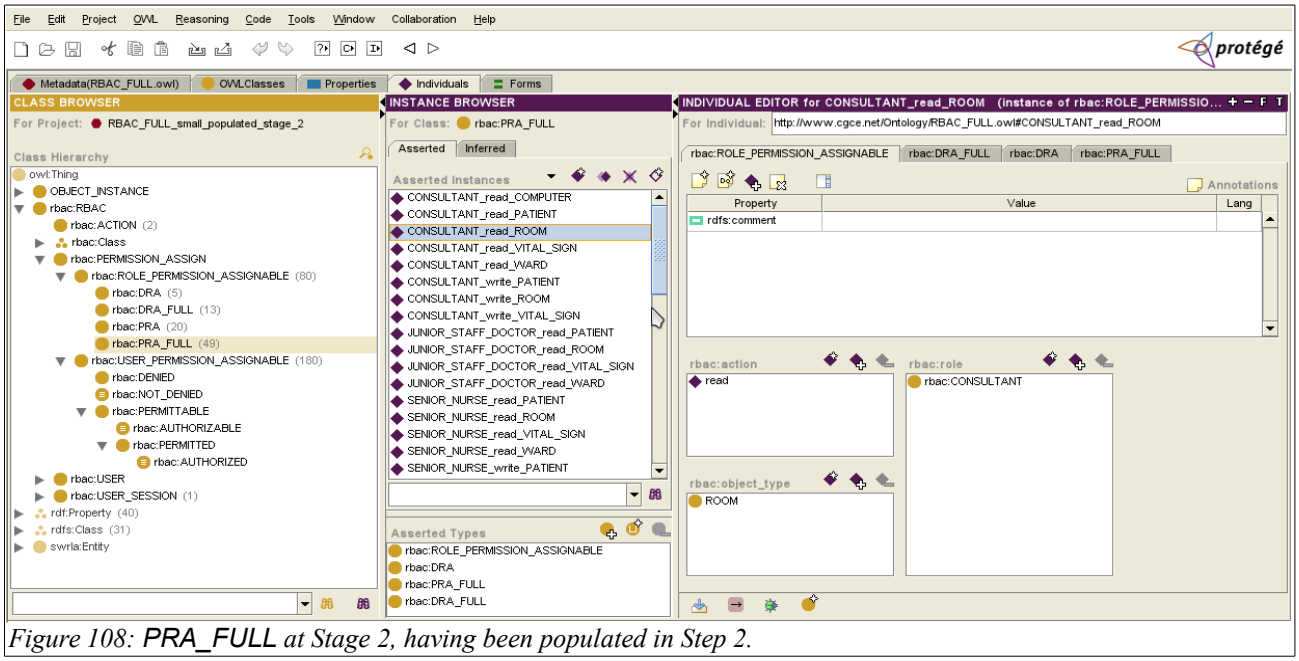

Figure 108: PRA_FULL at Stage 2, having been populated in Step 2.

Figure 108 shows PRA_FULL after Step 2 has run. This class is analogous to DRA_FULL.

6.8.2.3 Stage 3

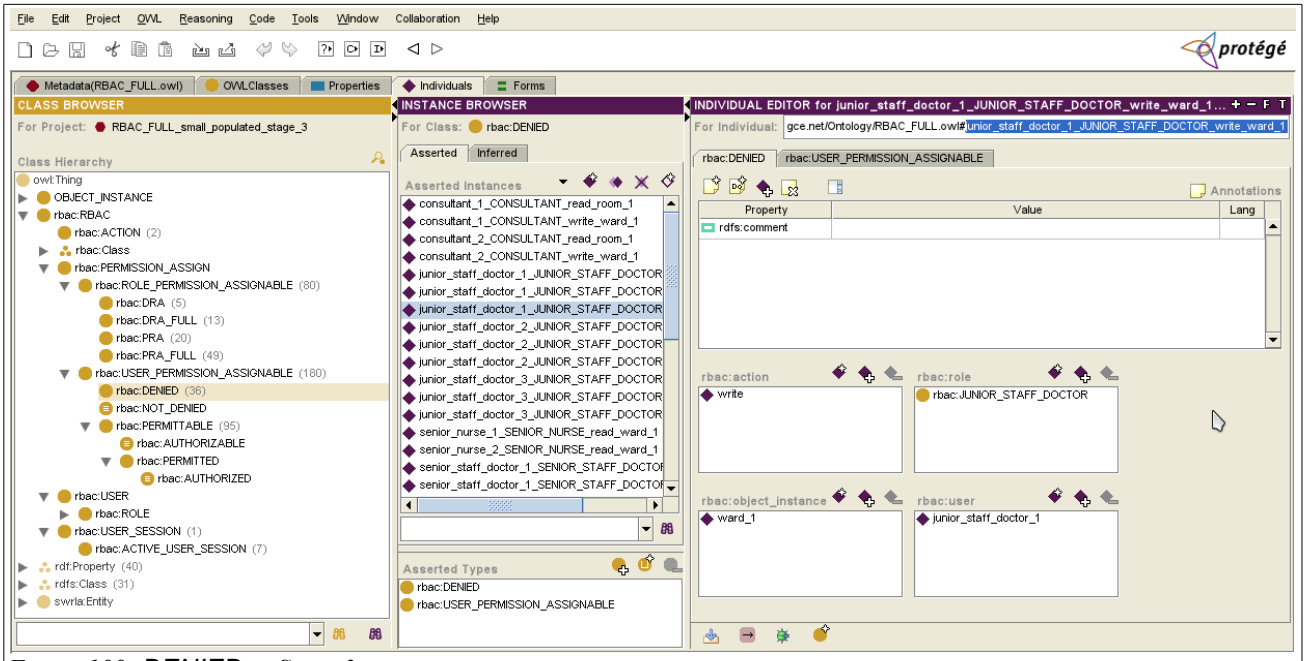

Figure 109: DENIED at Stage 3.

Figure 109 shows DENIED after Step 3 has populated it from USER_PERMISSION_ASSIGNABLE and DRA_FULL. Thus the individual highlighted also belongs to USER_PERMISSION_ASSIGNABLE.

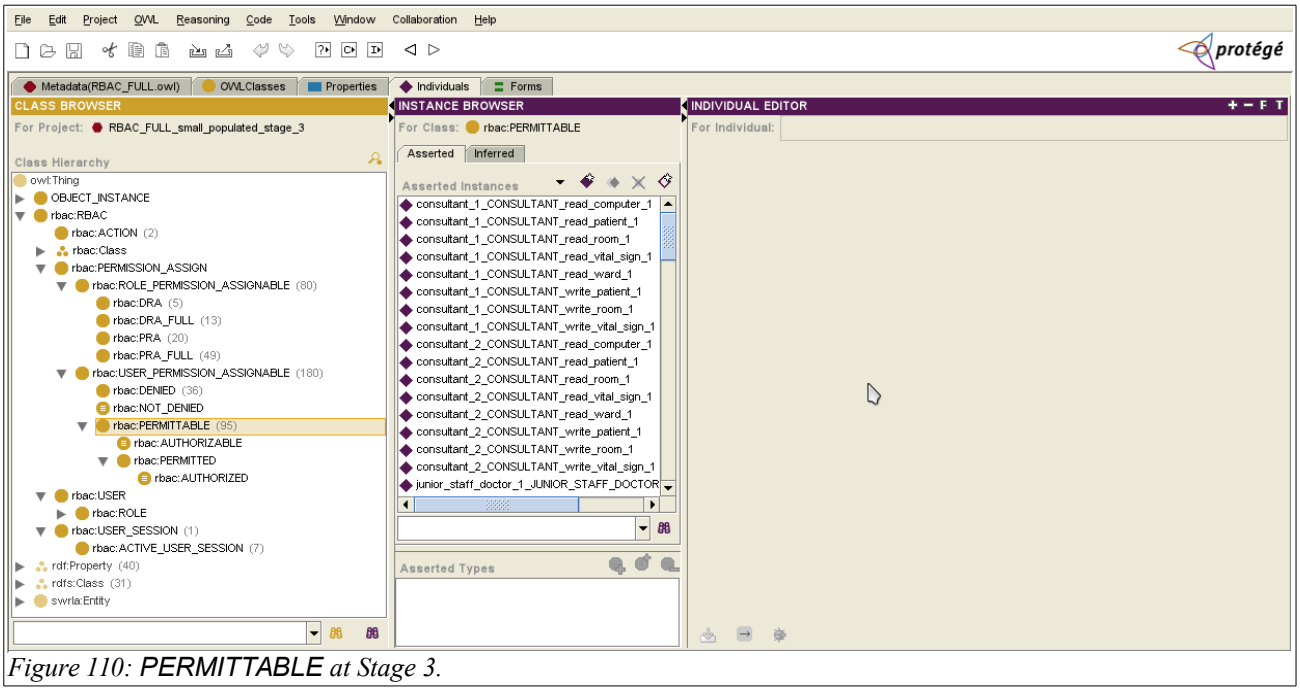

Figure 110: PERMITTABLE at Stage 3.

Figure 110 shows PERMITTABLE after Step 3 has run. Step 3 has populated it from USER_PERMISSION_ASSIGNABLE and PRA _FULL. Note that no individual is highlighted in this screenshot.

6.8.2.4 Stage 4

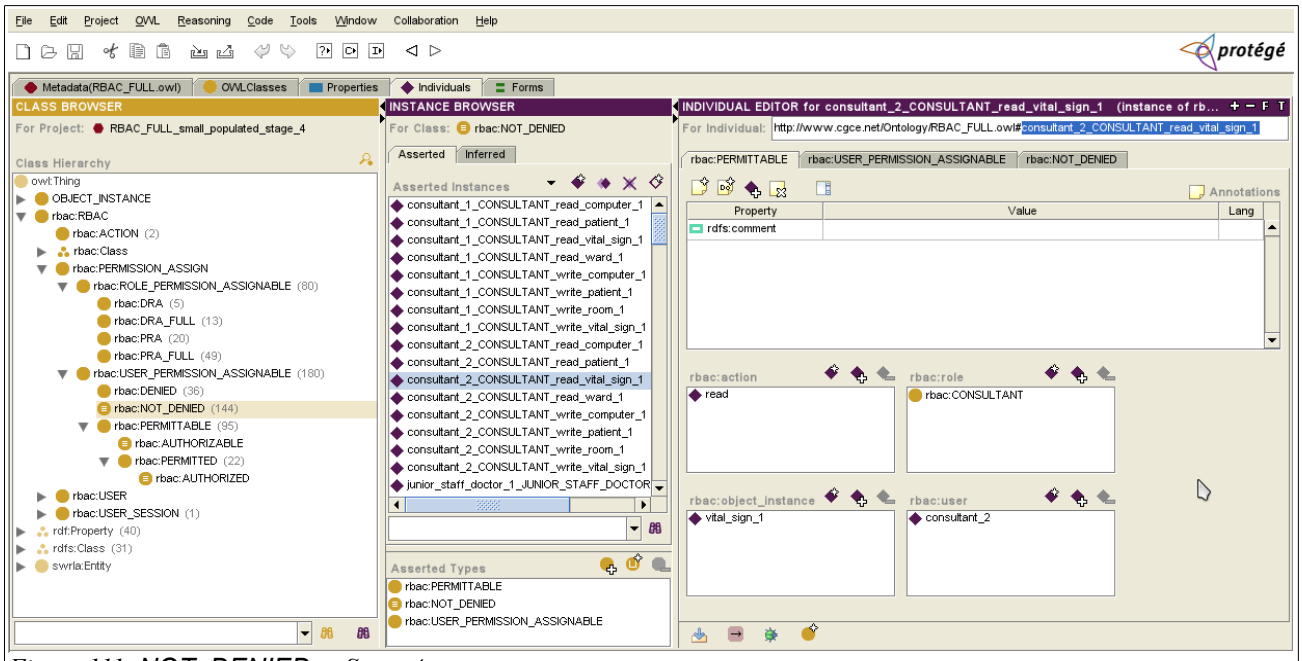

Figure 111: NOT_DENIED at Stage 4.

Figure 111 shows the results of populating NOT_DENIED in Step 4. Although each individual's membership of this class is defined many times due to the way the populating rule runs (as discussed earlier) each individual still appears only once in the Protégé window.

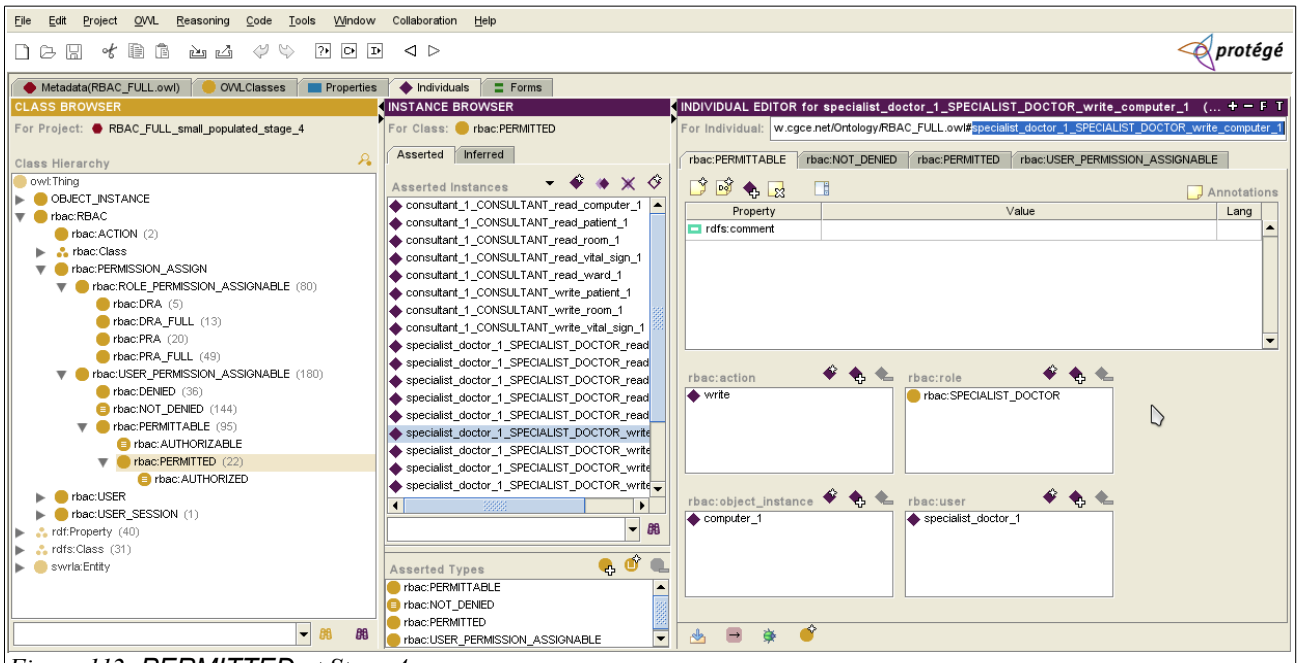

Figure 112: PERMITTED at Stage 4.

Figure 112 shows the results of populating PERMITTED in Step 4. As well as being a member of PERMITTABLE and USER_PERMISSION_ASSIGNABLE (as is necessary for membership of PERMITTED), the highlighted individual also belongs to NOT_DENIED. At Stage 4, every individual in USER_PERMISSION_ASSIGNABLE, PERMITTABLE and PERMITTED will belong to either DENIED or NOT_DENIED.

6.8.2.5 Stage 5

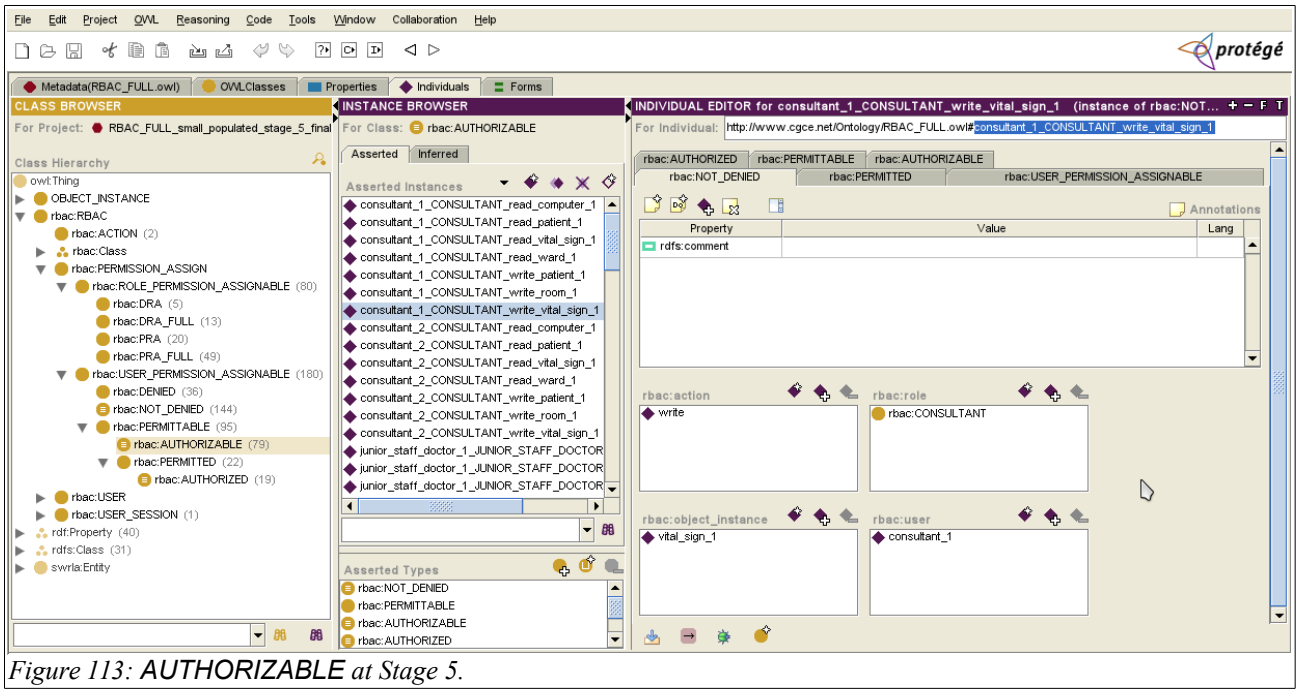

Figure 113: AUTHORIZABLE at Stage 5.

Figure 113 shows AUTHORIZABLE after Step 5. All individuals belonging to AUTHORIZABLE must by definition belong to the other three types listed for this individual (PERMITTABLE, USER_PERMISSION_ASSIGNABLE and NOT_DENIED).

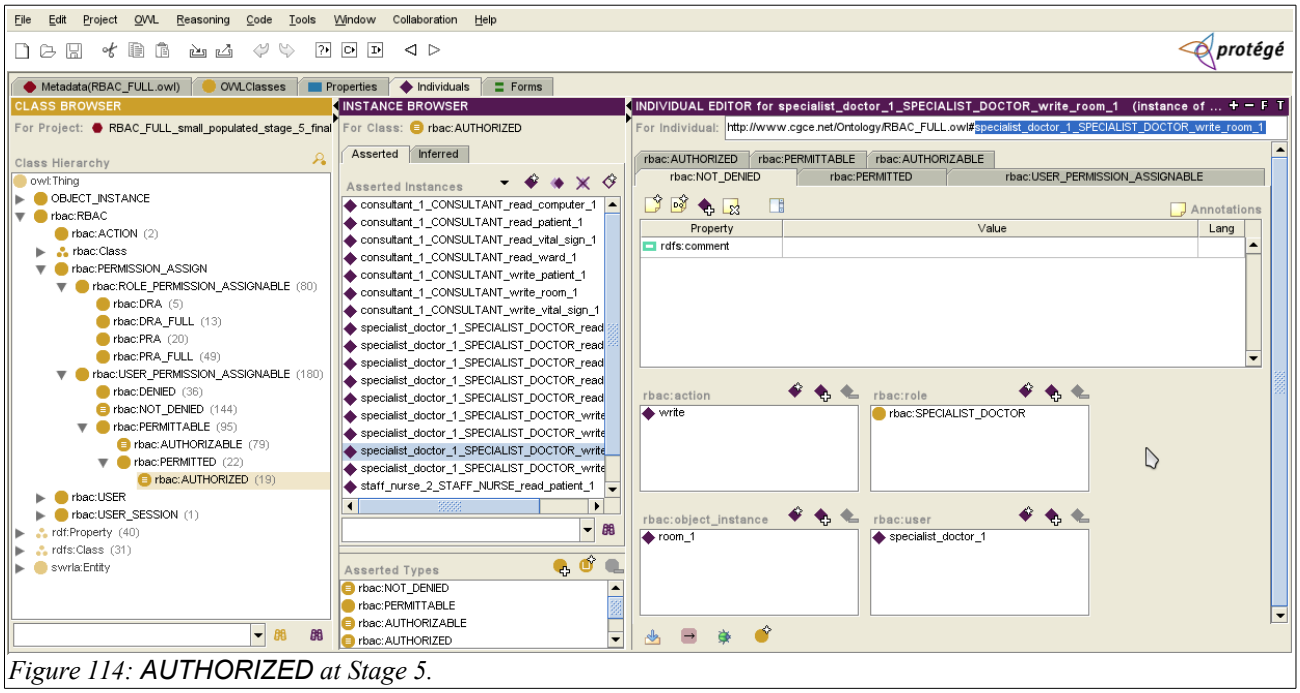

Figure 114: AUTHORIZED at Stage 5.

Figure 114 shows AUTHORIZED after Step 5. Again, any individual in AUTHORIZED must also be a member of AUTHORIZABLE, PERMITTABLE, USER_PERMISSION_ASSIGNABLE and NOT_DENIED.

6.9 Conclusion

In this chapter we have created and tested ESO-RBAC, which builds on SO-RBAC to create a purely ontological context-aware RBAC model written in OWL-Full. The reasoning is performed using Jena, since SWRL and Protégé cannot reason on OWL-Full ontologies.

We have proved that it is feasible to move towards semantic modelling of access control in terms of using rules which control permissions dynamically and take into account context-awareness of software applications which reside in pervasive computational spaces. However, ESO-RBAC has still not managed to fully address access control that assigns permissions according to situations created by pervasiveness of environments and computational spaces where applications and their data reside.

As ESO-RBAC is written in OWL, it has the same issues for negation and axiom reasoning as SO-RBAC, as described in Section 5.8. That is, negation has to be simulated, and each potential axiom needs to be explicitly defined as an individual. This means that reasoning is, as with SO-RBAC, a time-consuming process in ESO-RBAC.

However, because ESO-RBAC uses OWL-Full, and so is able to use the OWL class hierarchy in defining hierarchies of roles in an object-oriented fashion, by exploiting the class-individual duality of OWL-Full. That is, roles are defined as OWL classes. A role is defined as a 'type of' another role by sub-classing. A user is defined as being in a role by the USER individual being a member of the ROLE class. This approach contrasts ESO-RBAC with both predicate logic and SO-RBAC, and provides the major benefit of implementing RBAC in the Semantic Web, which is that hierarchies are defined natively in OWL.

However, Jena cannot work on inferred axioms, so it cannot identify an individual as belonging to a sub-class of a class, and nor can it see relationships defined for sub-properties. This means that in ESO-RBAC certain additional properties had to be defined to create instances that represent properties that are supposed to be inferred, such as recursive sub-classing. This is a flaw in the reasoner, rather than in OWL-Full itself, and a properly constituted OWL-Full reasoner would not have this problem.

Test results indicate that ESO-RBAC was successful in building a purely ontological dynamic RBAC model both the static and dynamic components of the model produced results that were consistent with the model based on predicate logic and with SO-RBAC (for the static component, as no dynamic rules were implemented in SO-RBAC). It is hoped that reasoning tools will be developed for OWL-Full that allow ESO-RBAC to be run without the workarounds that were found to be necessary in this testing. Beyond this, further work will be to develop the dynamic RBAC features in the ESO-RBAC model, for instance to introduce a context constraint hierarchy (using the OWL class hierarchy, as with roles), and to use heuristic rules to generate new context constraints dynamically. These are further discussed in Section 7.2.3.

7 Conclusion

7.1 Summary of Research

Predicate logic is useful for modelling access control to data, using logical rules based on a set of facts, and has been used to describe various access control models including Role-Based Access Control (RBAC). In this research, predicate logic is used throughout this research as the basis for implementing an RBAC model, based on the work of Barker & Stuckey [18] and Strembeck & Neumann [20][21], in Prolog, in a relational database management system (RDBMS) and in ontologies.

The RBAC model has the following RBAC features discussed by Barker & Stuckey [18]. Static RBAC governs access to data based only on the type of data (e.g. all data about patients, or about rooms in a hospital).

- User-Role Assignment, Role-Permission Assignment: This is the basic concept of RBAC, in which users are assigned to roles, and roles are assigned to permissions. In this way, the access that a user has to data is determined by the roles to which the user is assigned. Users are not assigned permissions directly.
- Role-Denial Assignment: This is the opposite of Role-Permission Assignment: roles are specifically denied access to data. Denials override permissions, so if a user is both permitted and denied access to data through different role assignments, then the user cannot access the object.
- Seniority: Roles are related to each other through a seniority hierarchy, and inherit permissions and denials depending on their position in the hierarchy. Permissions are inherited up the seniority hierarchy, while denials are inherited down it.
- Role inclusion: This is also a hierarchy of roles, but is separate from the seniority hierarchy. It defines a role as being a type of another role; for example, defining a 'junior doctor' as a type of 'doctor'. Unlike in the seniority hierarchy, permissions and denials are both inherited in the same direction in the inclusion hierarchy, towards included roles.
- Path inheritance: This is used for limiting the inheritance of permissions up the seniority hierarchy above certain levels.

It is often necessary to give users access to specific data in a data set, or only in specific circumstances; for example, a doctor only accessing data about patients he consults, or only having access at specific times of the day. This is known as dynamic RBAC. In this research, dynamic RBAC is implemented using a model devised by Strembeck & Neumann, [20][21] using *context constraints* which selectively prevent access to data according to rules. Context constraints, like denials, are inherited down the seniority hierarchy.

7.1.1 Modelling RBAC in Prolog

The model was first implemented in Prolog *facts* and *rules*. The data relating to *user-role assignments* and *role-permission assignments*, and dynamic context constraints, are codified in Prolog *facts*. Prolog *rules* are used to computationally determine whether *users* have access to data based on the facts. Each time we wish to determine whether a user should have access to data, a process is run on the base of Prolog facts to determine the permissions or denials. We used SWI-Prolog to implement and test the RBAC model.

7.1.2 Modelling RBAC in RDBMS

The Prolog model of RBAC was then applied to an RDBMS (Oracle 10g) using two different methods. The first method offers a much simpler way of translating the Prolog facts and rules into RDBMS concepts. The second approach provides greater security than the first by taking advantage of the inherent security features of the RDBMS. Each of these is explained in the two paragraphs below.

In the first method, the Prolog facts relating to user access are stored as records in database tables. The tables holding RBAC data are in the same database as the data over which we run RBAC, but probably in a different schema. The Prolog rules for RBAC are written either as database views using SQL upon the RBAC schema, or implemented as triggers on the tables from the RBAC schema, using the PL/SQL procedural database programming language. This is because some Prolog rules in the RBAC model use recursion, which current SQL does not handle. Using this method, all aspects of the RBAC models can be implemented, and the RBAC can be determined by issuing standard SQL queries on RBAC schema tables. This approach can be used to provide access control at the application level. It is important to note that at the database level, the application always accesses the data using one user ID, which is likely to be locked to accessing data from the application interfaces. The application would pass the user ID of the person who is logged into it as a parameter to the database when the user attempts to access data, and this would form part of the query to determine whether the application-level user gains the access. Furthermore, we can easily program both static and dynamic RBAC at the application level because the rules for both can easily be translated into either SQL views or PL/SQL (or equivalent) procedures.

The second method of implementing RBAC on a relational database provides access control at the *database* level by using the meta-data (or data dictionary) of the RDBMS. In this method, we have to distinguish between static and dynamic RBAC. The static RBAC was mostly implemented using standard SQL **CREATE ROLE**, **CREATE USER** and **GRANT** commands. However, while RBAC permissions can be implemented this way, denials cannot be so implemented because **GRANT** is only a positive granting of permission: there is no negative authorisation in SQL access control syntax. The dynamic RBAC was then implemented using Oracle's Virtual Private Databases (also called Row-Level Access Control) feature. [58] We found that most, but not all, of the features of the RBAC model could be implemented. We could not implement path inheritance restrictions. However, denials can be implemented using this feature, because a rule can be set up such that a role is denied access to data in a table even if given access to it via a GRANT command. The implementation of dynamic RBAC is product-specific, as it is not part of the SQL standard. Postgres has a feature called VEIL [66] that also implements dynamic RBAC, but its syntax is different from that of Oracle VPD. By contrast, the static RBAC implementation uses standard SQL commands, and is likely to be very similar across RDBMSs, although some, such as MySQL, do not support RBAC in their data dictionary.

It is not appropriate to compare these two methods and give recommendations that one should be used instead of another. The first method can implement all aspects of the RBAC model. The second method cannot implement all features of the model. In particular, it could not implement the path inheritance restrictions, as there is no provision for limiting preventing privileges from being inherited by roles in the Oracle 10g data dictionary. The implementation using the second method is specific to the RDBMS, while the first method can be implemented similarly across all DBMSs, since it uses standard SQL and straightforward trigger procedures.

7.1.3 Modelling RBAC in OWL

In OWL/SWRL enabled ontologies we have managed to translate Prolog facts and rules into OWL/SWRL concepts. It is important to note that there are three versions of OWL. OWL-Lite was considered to be too limited for use in defining an RBAC model. RBAC models were developed in OWL-DL and OWL-Full. OWL-Full is more expressive than OWL-DL in that it can fully express RDF syntax; it allows classes to be manipulated as individuals, which OWL-DL does not.

We started with OWL-DL for two reasons. OWL-DL is better supported by reasoners than OWL-Full, and it is sufficient to demonstrate the concept of building an RBAC model in OWL. However, if we really wanted to take advantage of the power of OWL ontologies, we had to move to OWL-Full, which unfortunately is not widely supported by reasoners. In this research, we have demonstrated how both OWL-DL and OWL-Full can be used for managing RBAC, and it remains to be seen if future development of reasoners will open more options in RBAC through OWL-Full.

7.1.3.1 SO-RBAC in OWL-DL

The OWL-DL ontology was programmed using Protégé [24]. Within Protégé, relationships from some logical characteristics of OWL properties (symmetry and inversity, but not transitivity) can be inferred.

Initially, the Prolog facts and rules for RBAC were translated into OWL individuals, classes and properties with little change in semantics of the original Prolog RBAC model. We still have facts and rules from Prolog in OWL/SWRL-enabled ontologies. The Prolog facts became either individuals bound to classes, or object properties, in the OWL ontology. The Prolog rules were translated into SWRL rules, which were run upon the OWL ontology. However, a few Prolog rules do not need to be represented as SWRL rules in SO-RBAC, because they can be represented through the property hierarchy, allowing some object property relationships to be inferred.

Some of the rules in Prolog are recursive. In theory, the need for recursive rules in ontologies, defining relationships between roles using object properties, should have been eliminated by defining object properties as transitive. However, Protégé does not infer properties based on transitivity. Therefore, these recursive SWRL rules still have to be defined.

It is important to note that we used SWRL for two separate purposes in SO-RBAC. In principle, every fact in Prolog can be represented as an individual of a class in OWL. However, facts representing binary relationships between individuals can be represented using object properties in OWL, which is a natural choice. We give two examples.

(a) The direct seniority fact `d_s(manager, worker)` is represented in OWL using a *directly_senior_to* property linking (OWL constraints) the USER individuals manager and worker.

(b) The binary relationship of user-role assignment (`ura` facts in Prolog, e.g. `ura(john, doctor)`) is, in contrast, represented using individuals in a class in OWL (URA) to maintain the analogy with permission-role assignment (`pra` facts in Prolog, e.g. `pra(doctor, write, patient)`, which is a ternary relationship and therefore has to be represented by a class (PRA) in OWL.

All facts represented in the SO-RBAC model by object properties (method (a) above) link one role to another in the RBAC model. We used SWRL in both cases, because they define constraints upon the OWL model. They are prerequisites for running reasoning rules that ultimately grant permissions and denials.

In this thesis, the implementation in OWL-DL is only shown for static RBAC. Dynamic RBAC can be implemented in SO-RBAC using OWL-DL, with additional OWL classes and SWRL rules to address the context constraints. The reasoning process for dynamic RBAC would, like the reasoning process granting permissions and denials in static RBAC, be a direct translation from the Prolog implementation in Section 3.3.

Many reasoners have been implemented for OWL-DL, making it easy to create and test an ontology in this version of OWL. However, the OWL-DL implementation of SO-RBAC largely follows the predicate logic implementation, which was predetermined by the Prolog RBAC model. Therefore, we were unable to take advantage of the native features of OWL, such as the direct modelling of a role hierarchy using the OWL class hierarchy. This has been addressed in OWL-Full, which is used for modelling ESO-RBAC.

7.1.3.2 ESO-RBAC in OWL-Full

We have implemented static and dynamic RBAC using OWL-Full. This RBAC model is named ESO-RBAC. OWL-Full makes it much easier to design an RBAC model natively in OWL. This is because it can take full advantage of the OWL class hierarchy, so that OWL sub-classes can be used to define RBAC role inclusion, rather than having to define separate object properties and use SWRL for it as in (b) from the previous section.

OWL-Full has class-individual duality, which means that it allows classes to be manipulated by reasoning rules as if they were individuals. This is clearly an advantage, because it allows inference at two different levels: at the level of individuals when copying them through reasoning rules across ontological classes, and at the class level where the initial hierarchies of OWL classes can be extended through reasoning. However, there was no need to exploit OWL at this level in ESO-RBAC, but we instead manipulated the object properties of classes as if these were individuals. Thus, in ESO-RBAC, the relationship between roles and sub-roles, which are OWL classes, can be implemented naturally in OWL-Full. Additionally, the relationship of a user to a role can be defined simply by defining the user as an individual that is a type of a particular ROLE class, rather than using a separate URA class. This is also a 'natural' OWL-Full implementation of the relationship between a role and a user, as defined in Prolog.

In ESO-RBAC, unlike in SO-RBAC, there is not a precise relationship between Prolog facts and OWL classes or properties, or between Prolog rules and reasoning rules. As noted above, some Prolog facts are implemented as class-individual memberships in OWL-Full, because of class-individual duality. Furthermore, in theory, some rules (for example, the recursive rules relating to the RBAC concept of Role Inclusion) can be eliminated due to their representation via the OWL-Full class hierarchy.

ESO-RBAC was tested by running reasoning rules in Jena (we did not use SWRL because it does not support OWL-Full and cannot deal with class-individual duality). Jena follows the same semantics in terms of creating reasoning rules as SWRL. It is important to note that we had to write additional reasoning rules when running Jena compared with running reasoning rules in SWRL. This is because there is no Jena plug-in for Protégé that works on OWL-Full.

Inverse relationships are used in the (E)SO-RBAC model to define inverse properties to *directly_senior_to* and *senior_to*, respectively called *directly_junior_to* and *junior_to*. The Protégé environment automatically fills in inverse object property relationships. Thus, when a *senior_to* relationship is defined between two individuals, the corresponding inverse *junior_to* relationship is also defined. If r1 senior_to r2 is defined, then because *junior_to* is defined as inverse of *senior_to*, the triple r2 junior_to r1 is also filled in.

The Jena rules were run outside of Protégé on the command line using a Java program that implements Jena. Therefore, the Protégé environment was not available to infer the inverse relationships, and so in this ESO-RBAC implementation using Jena, the properties *directly_junior_to* and *junior_to* have to be defined directly using Jena rules.

Furthermore, the SWRL plug-in for Protégé can identify indirect sub-classes (sub-classes of sub-classes), and identifies an individual as a membership of a class if it is membership of any sub-class of this class. However, Jena does not recognise either inferred sub-classing or inferred class membership, but only direct sub-classes and direct membership of a class. Therefore, it was found to be necessary when using Jena in ESO-RBAC to define additional rules to infer these relationships.

7.2 Evaluation

7.2.1 OWL in general

7.2.1.1 Concerns with OWL

Monotonicity in OWL

Unlike description logic, OWL is monotonic. This has two meanings in the context of an OWL ontology.

Persistence of Reasoning Results

First, reasoning places individuals into classes. In other words, individuals placed in a class in an OWL ontology cannot be retracted by the reasoning process. The results of reasoning in OWL are always persistent. Therefore, running the same reasoning process repeatedly upon the same instance of an ontology, when the data asserted upon initialisation have changed, does not erase individuals from classes when the reasoning process based on the new data would not move them there. From that perspective, we cannot expect that our reasoning process, which grants either permissions or denials, can be re-run without first erasing individuals which had been moved to various classes as a result of previous reasoning. In SO-RBAC and ESO-RBAC, every class contains either only asserted individuals or only inferred individuals, thus making it simple to erase individuals where appropriate before any reasoning is performed. Similarly, reasoning in OWL-Full can define a class (treated as an individual) as a sub-class of another class, but cannot break a link in the class hierarchy.

Negation in OWL

Second, OWL uses an open-world assumption, in contrast to the closed-world assumption of DL systems. This has implications for modelling negations in our RBAC models.

Negation is handled differently in predicate logic and OWL. Predicate logic uses closed-world reasoning, i.e. 'negation as failure', in which any query not proven to be true is taken to be false. Prolog has a function `not`, which negates any predicate that it governs. However, there is no explicit negation function in SWRL, to indicate that an object-property relationship does not occur between two individuals, or that an individual is not a member of a class. OWL uses open-world reasoning, where something has to be explicitly asserted as being not true, and reasoning languages do not have a negation function as such.

However, closed-world reasoning can be simulated in ontological reasoning languages. In SO-RBAC, this is achieved using SQWRL (Semantic Query-enhanced Web Rule Language) functions makeSet and notElement to test for the presence or absence of an individual in a set. The SQWRL function makeSet makes a set consisting of a list of

previously defined individuals. The SQWRL functions element and notElement respectively check whether a given individual is, or is not, a member of a set. These two functions, in combination enables negation-as-failure to be used with OWL and SWRL. This is explained as follows. We want the class NOT_DENIED to contain all elements in the class USER_PERMISSION_ASSIGNABLE that are not in DENIED. This is done by first using makeSet to create a set containing all individuals that are in the class DENIED, then using notElement to check that an individual is not a member of that set.

```
rbac:USER_PERMISSION_ASSIGNABLE(?x) ∧ rbac:DENIED(?y) ˚
sqwrl:makeSet(?d, ?y) ˚ sqwrl:notElement(?x, ?d) →
rbac:NOT_DENIED(?x)
```

The function notElement(?x, ?d) has to compare an individual *?x* with each member of the set *?d* to check that *?x* is not in set *?d*. This may be a time consuming process, if there is a large number of individuals in the set ?d. Moreover, in this rule, notElement(?x, ?d) has to be run many times, once for each element ?x in the class USER_PERMISSION_ASSIGNABLE. Therefore, this rule takes a long time to run.

In Jena, for ESO-RBAC, classical negation can achieved using the function `noValue`. The equivalent to the above SWRL rule in Jena is as follows.

```
[4_not_denied: (?x rdf:type rbac:NOT_DENIED )
   <-
  (?x rdf:type rbac:USER_PERMISSION_ASSIGNABLE )
  noValue(?x rdf:type rbac:DENIED )
]
```

This function checks whether individual *?x* is in USER_PERMISSION_ASSIGNABLE, and is not in DENIED, and if both conditions are satisfied, puts *?x* in the class NOT_DENIED. The syntax for the negation is simpler in Jena than in SWRL, because there is only one function (`noValue`) instead of two (makeSet and notElement). However, the function `noValue` still needs to compare every *?x* against every individual in the class DENIED, so the negation process is still slow.

In summary, the lack of explicit classical negation in OWL means that this has to be simulated in the reasoning languages, and this simulation process is slow.

We take the liberty to interpret the monotonicity of OWL as an advantage from a software engineering perspective, because our ontologies will never grow as a consequence of repeatedly executed reasoning processes. Repeated reasoning upon the same ontological model does not make our ontological solutions complex in terms of the class hierarchy or in terms of the number of individuals. However, re-running the reasoning process on a changed data set involves erasing the ontology and repopulating (and possibly rebuilding) it.

Populating OWL classes with individuals

In Prolog, a rule can be queried based on a dataset, and all axioms that apply to it are automatically returned. Consider the following rule

```
pra_full(R1,A,O) :-
  senior_to(R1,R2),
  pra(R2,A,O).
```

This rule `rpa_full`, when run, returns every combination of `(R1,P,O)` (axiom) identified by the antecedents `senior_to` and `rpa`. It should be noted that no new facts are created when a rule such as this is run in Prolog. Instead, the axioms that meet the conditions of the rule are computed every time it is run. The axioms that meet the

antecedent predicates `senior_to` and `rpa` may themselves be either computed through some other rule, or be stored as facts.

In contrast, the OWL reasoning process can only move individuals that already exist. While new object property relationships can be created, new individuals cannot. Each reasoning rule works on a base of individuals that have been placed in a class, and object property relationships that have been created, and stored in the ontology. For example, consider the equivalent SWRL rule to the above Prolog rule for `rpa_full`:

PRA(?x) ∧ role(?x, ?r1) ∧ action(?x, ?a) ∧ object_type(?x, ?o) ∧
senior_to(?r2, ?r1) ∧ ROLE_PERMISSION_ASSIGNABLE(?z) ∧
role(?z, ?r2) ∧ action(?z, ?a) ∧ object_type(?z, ?o) → PRA_FULL(?z)

Note that in SWRL, each of the properties of an individual need to be specified separately: there is no construct similar to `rpa_full(R1,A,O)` in OWL. In this SWRL rule, PRA, ROLE_PERMISSION_ASSIGNABLE and PRA_FULL are classes. An individual *?z* is added to the PRA_FULL class if it matches the rules in the antecedent. However, the individual *?z*, with the object properties specified in the antecedent, must already exist in the ontology if it is to be moved to PRA_FULL. It is not created if it does not exist.

Therefore, all individuals representing potential permission states relating to a role, user, class and action need to be created when setting up and populating the ontology. This can take a long time to do, due to the large number of individuals that need to be created. In a model with 250 users, 10 roles, 2 actions and 700 objects in 10 object classes, the numbers are as follows:

- 10 × 2 × 10 = 200 ROLE_PERMISSION_ASSIGNABLE individuals
- 250 × 2 × 700 = 350,000 USER_PERMISSION_ASSIGNABLE individuals

If we add 5 dynamic context conditions, then the numbers of CONTEXT_CONSTRAINT_APPLICABLE and CONTEXT_CONDITION_POTENTIAL individuals, in addition to those, are as follows:

- 10 × 2 × 10 × 5 = 1,000 CONTEXT_CONSTRAINT_APPLICABLE individuals
- 250 × 2 × 700 × 5 = 1,750,000 CONTEXT_CONDITION_POTENTIAL individuals

It can be seen that the number of individuals that need to be created in the ontology before reasoning grows quickly with increasing size of model in terms of roles, data and users. Because of this, the reasoning process takes a long time to run. Even with the small models used in testing, it was necessary to perform chain reasoning, as running all the reasoning steps at once was found to take far too long, and in some cases crashed. However, chain reasoning leads to the creation of very large output files that need to be stored temporarily. But in a situation-aware system, where the permissions depend on factors external to the data (such as time of day, or temperature), the reasoning process would have to be performed, from some stage, with every query.

OWL Speed and Efficiency

We have identified two ways in which reasoning was slow. The first is in the rules used to perform negation, as noted above. This problem cannot easily be solved. The second is that processing was found to be slow when running a rule that reasons on individuals that had been moved in a previous rule, in the same process. To resolve this problem, the reasoning was broken down into steps, with the ontology saved in a new file after each step, to be reasoned on in the next step.

7.2.1.2 Advantages of OWL

Faster reasoning on persistence

If the permissions and data have not changed since reasoning was performed, then querying permissions involves simply querying a static ontology, rather than running a computation, i.e. reasoning rules. Therefore, queries on individuals, once reasoning has been performed, is likely to be faster in OWL than using predicate logic, because predicate logic may query on views, while OWL always queries on stored data.

In SO-RBAC and ESO-RBAC, we propose performing the reasoning in stages. This is done for two reasons. It is faster than doing it all at once, as detailed above. But also, it means that changes to the data and permission assignments do not necessarily require a complete renewal of the ontology. Instead, the ontology can be reset to an earlier stage, and the reasoning re-run from there.

Use of natural class and property hierarchy in OWL

A very common feature of RBAC, and one used in the model discussed in this thesis, is the use of role hierarchies. There is no natural way of representing hierarchies in predicate logic. By contrast, the class hierarchy in OWL means that OWL is naturally suited to representing hierarchies, without the need to define predicates that explicitly do this.

Similarly, an inherent feature of RBAC is defining a user as a 'member' of a role. In predicate logic, this has to be explicitly expressed using a predicate such as `ura(user,role)`. Due to the limitations of OWL-DL, we also found this to be necessary for the SO-RBAC model. However, in OWL-Full, it is possible to define users as individuals, and roles as classes, and assign a user to a role by making the user individual a member of the role class. This means that the relationship between the user and the role is represented in a way that is natural for OWL. The ability to use classes and individuals interchangeably, and decide when we need constraints (as opposed to classes and individuals) when describing a particular domain of interest, is a great advantage. In other words, OWL does not define what needs to be an individual, class or constraint.

Similarly, the use of hierarchical classes and hierarchical property relationships means that many rules that are necessary in predicate logic can be omitted from ontological reasoning languages. For example, the following Prolog rule

```
senior_to(R1, R2) :- directly_senior_to(R1, R2)
```

can be expressed in OWL by defining senior_to as a sub-property of directly_senior_to, and no reasoning rule is necessary to define this. In theory, is should also be possible to eliminate all recursive rules in the reasoning language by defining properties such as senior_to as transitive. However, the ontological building tool that we used (Protégé) does not infer property relationships transitivity, and nor do either of the reasoning languages SWRL or Jena. Additionally, Jena does not infer indirect sub-classes, which are used extensively in the OWL-FULL model ESO-RBAC. The state of the art in ontological reasoning tools needs further development, particularly in OWL-FULL, before the full power of ontological reasoning can be exploited. However, given the generic nature of the semantic web, it should be possible to develop such reasoners; this is a matter for further research.

The inference mechanism in OWL and SWRL/Jena is more thus powerful than that of predicate logic. Static ontologies can be queried quickly; the data obtained by the reasoning process are reusable: they do not have to be recomputed each time unlike in predicate logic and in RDBMS syntax. The ontologies need to be repopulated whenever the information that they are modelling changes. However, this need not necessarily mean that the entire ontology needs

to be rebuilt from scratch each time the data or the RBAC rules change; SO-RBAC and ESO-RBAC are designed so that the reasoning is performed in steps, with each step creating a consistent ontology representing a stage in the reasoning process, and the reasoning process can be partially run from any stage. Alternatively, since the types of information going into particular parts of the class hierarchy are well defined, a partial reasoning process could be re-run by clearing and re-populating those classes where information has changed, and re-running whichever steps of the reasoning process need to be re-run.

Not Vendor Specific

SO-RBAC and ESO-RBAC are not database vendor specific. Section 3 documents a way of implementing some of the features of dynamic RBAC in an RDBMS. However, the syntax of the dynamic RBAC mechanism in particular is specific to the RDBMS used. OWL allows the development of a generic, non-vendor-specific syntax for dynamic RBAC.

SO-RBAC and ESO-RBAC and types of data repositories to which they control access: potentially these can access any data repository and are particularly suited to accessing data in the semantic web.

Permissions and Denials granted through SWRL and Jena are application-independent. They depend on the positioning of individuals in classes in the RBAC ontologies, and these can be queried by any reasoning language using standard reasoning syntax.

Independence of query layer from ontology

The querying layer is also independent of the ontology, which is stored in OWL files. There is no need to load or link the contents of a data file into a database schema, or to load the data into an application environment such as that of Prolog. Any reasoning tool can be used to run a query on an OWL file, which is stored as a plain file on a computer system; there is no requirement to use a specific environment to query an OWL file.

Summary

In summary, the main advantages of OWL over predicate logic in modelling RBAC are as follows:

- the ability to use the ontological class and property hierarchies as part of the model, allowing a natural representation of hierarchical relationships and eliminating the need for certain computations;
- the ability to query static ontologies quickly without recomputation;
- independence of the querying layer from the ontology;
- OWL and reasoning languages are not vendor-specific.

However, the ontology needs to be rebuilt every time the data or permissions change, and the reasoning process is slow and the OWL files are large.

7.2.2 SO-RBAC and ESO-RBAC Models

Our first RBAC model, SO-RBAC, uses OWL-DL, and is based on similar reasoning rules to those used in predicate logic (Prolog). The purpose of SO-RBAC is to demonstrate the feasibility of writing an RBAC model in OWL. However, consequently SO-RBAC does not take full advantage of the flexibility offered by OWL, due to limitations in OWL-DL discussed earlier in this section. Following this proof of concept, we developed ESO-RBAC, which uses OWL-Full, giving much greater freedom to break away from the confines of DL and create a model that is naturally suited to OWL.

The core ESO-RBAC model is reusable across any domain. There are two main classes at the top level of the ESO-RBAC hierarchy, called DATA and RBAC. The DATA super-class contains exclusively domain-specific data, equivalent to the information that might be stored in user-created tables in a relational database. However, because OWL has a class hierarchy, relationships between information and types of information can be defined in a much more flexible, 'object oriented' fashion than is possible in a relational data model.

The RBAC super-class contains the information defining permissions and denials. The class structure of RBAC is mostly *not* domain-specific. The only part that is domain specific is the set of roles, which is defined under the ROLE class under RBAC. All other classes under the RBAC super-class are directly related to the RBAC model. It is in the RBAC super-class that individuals representing user permissions and denials are moved by the SO-RBAC or ESO-RBAC reasoning process. In SO-RBAC these are normal OWL individuals, while in ESO-RBAC they are classes that are treated as individuals by the reasoner.

The re-usability across domains of the RBAC model in Prolog is similar to that in OWL. As in OWL, Prolog allows the reuse of the same rules to reason permissions and denials based on whatever roles, permissions and data are defined by the domain administrator. This is also similar to the separation of user-defined data and meta-data (data dictionary) in a relational database. Therefore, in principle, it should be possible to define RBAC rules in a relational database that is independent of the user-manipulated data. However, in section 3.5 we found that RDBMSs do not implement all of the features of RBAC models discussed in Chapter 2. The only way of implementing certain features, such as path inheritance restrictions, in the RDBMS that we tested would be to implement the RBAC model as a series of normal database tables, rather than using the data dictionary. This approach jeopardises the separation between RBAC data and ordinary data. However, this can be mitigated by defining the RBAC data tables in a separate database schema from other data.

Implementation of SO-RBAC and ESO-RBAC would use an OWL API to determine who would be permitted and denied access to certain data from the application layer, completely independent of the structure and type of data accessed. There are many tools available for populating OWL from any data source, including flat files or a DBMS. In this thesis, the data model was populated using a script in order to prove the concept that SO-RBAC and ESO-RBAC provide a mechanism for creating permissions and denial completely independently of the types of data sources on which we wish to control access. It is very easy to retrieve the content of OWL classes storing individuals relating to permissions and denials from any type of software applications built in integrated development environments that have plug-ins to an OWL API.

7.2.2.1 Reasoning processes

Uniquely, both SO-RBAC and ESO-RBAC infer at the OWL level rather than at the application level. ESO-RBAC exploits the natural hierarchy of OWL and performs all reasoning inside OWL using OWL reasoners. Other ontological RBAC models leave much of the reasoning process to other layers. Our RBAC model is the only one in which the ontological reasoning process is completely automated through SWRL or Jena rule chaining. In particular, we are not aware of any other ontological RBAC model that resolves conflicts between permission and denial by using negation functionality in OWL reasoners to enforce the standard RBAC rule that 'denials override permissions'. Additionally, ESO-RBAC is unique in using the ontological class hierarchy to define some relationships between roles, and the user-role assignments; this eliminates the need to define them explicitly using OWL properties or reasoner functions. The use of the class hierarchy to define RBAC relationships natively allows the use of RBAC with object-

oriented data modelling. It should also be noted that the data on which the RBAC model operates (in the DATA super-class in the (E)SO-RBAC models) could also be defined through the class hierarchy; in ESO-RBAC in particular, this allows for considerable flexibility in defining permission to perform action on object types; for example, defining an RPA relationship on a class of DATA would implicitly define the same relationship on sub-classes of that class. Although object-oriented data models could be created in DL or in relational databases, such a representation would be rather convoluted and unnatural. OWL allows hierarchical relationships among both roles and data to be defined and related to each other, and ESO-RBAC uniquely performs reasoning on such a model.

Pre-requisites in the Reasoning Process

Our reasoning process, which grants correct permissions or denials requires:

i. An ontological model, which stores the semantic essential in the process of granting permissions and denials. Therefore (E)SO-RBAC Ontology should be ready to expand its basic structures into hierarchies and accept its individuals and accommodate required constraints in order to enable the SO-RBAC reasoning process.

ii. Clearly defined steps in the (E)SO-RBAC process which specify which ontological classes and constraints are involved in the process and what would be the outcome of each of its step.

We would like to draw the reader's attention to the dual roles of steps in the reasoning process, because we use reasoning in various stages of the process, but only after pre-conditions for the reasoning have been met, by the same process. However, "meeting pre-conditions" in the proposed process does not necessarily mean that we must use a particular reasoning mechanism for it. We give two examples.

We often populate a selection of ontological classes from existing data sources of a particular domain of interest and use the word "*assert*" (even if the selection of (E)SO-RBAC classes has to be populated at initialization), but we also "*infer*" (as opposed to "*assert*") ontological individuals in a selection of ontological classes through the reasoning process. In both cases these might be pre-conditions for continuing with a particular step of our process of granting permissions and denials.

The same applies to constraints: we sometimes define them as a part of our ontological model (i.e. manual assertions, as a part of ontological initialization is expected) or *infer* constraints through reasoning if they are pre-conditions in the process of granting permissions or denials.

Therefore, assertions and inference are interwoven in the reasoning process, but in its final stage, after we meet all pre-conditions, a chain of SWRL or Jena rules is running in one go and securing permissions or denials.

Characteristics of the Reasoning Process

We draw the reader's attention to a few important characteristics of the (E)SO-RBAC reasoning process.

At each stage the (E)SO-RBAC ontology is in a state where the process can be run from the following step onwards. In other words, it is not necessary to always re-run the (E)SO-RBAC process from the beginning.

A portion of (E)SO-RBAC ontological classes will remain 'empty' until a reasoning process determines which individuals from the asserted classes will be 'moved' (or copied) into (E)SO-RBAC classes which were empty on (E)SO-RBAC initialisation.

It is evident from the process that we perform initial assertions in steps B, C and D. However, assertions continue in stages F (we assert role permissions and role denials) and in stage H when we insert individuals into URA class (in

SO-RBAC) or assign them to ROLE classes (in ESO-RBAC). Therefore we do not limit assertions to the first few steps of the process and consequently they are interwoven with inference.

Reasoning in (E)SO-RBAC process, with SWRL or Jena rules, occurs in steps E, G and J. However, In the (E)SO-RBAC process we perform two types of reasoning.

- The first type is in step E, when we use SWRL or Jena for creating a set of new object properties. All of the object properties for which this is done have ROLE class as both domain and range, as the purpose of this step is to set up all the relationships between roles in the RBAC model.
- The second type of reasoning is performed stages G and H, where we run SWRL or Jena rules in order to move (copy) individuals across (E)SO-RBAC in order to determine permission or denials in particular request, imposed by a user, who has a 'role' and would like to perform an 'activity' upon set of "objects".

Therefore the outcome of our reasoning that a particular user name, which has been moved across the (E)SO-RBAC ontology according to our reasoning process, can be found as an individual of either PERMITTED or DENIED ontological class.

7.2.3 Future Works

Future work would be to further develop dynamic RBAC in the ESO-RBAC model. Section 6.4 demonstrates the use of context constraints, modelled on the system of Strembeck & Neumann [21], in the ESO-RBAC model, and defines a single context constraint applicable to the healthcare domain considered herein. This context constraint ensures that junior nurses can only modify the data of patients in their ward, based on data individuals stored in the DATA super-class as well as user data stored in the RBAC super-class. It would be beneficial to further test the ability of the model to handle complex context constraints, which may not only be internal (depending on data in the ontology on which the ESO-RBAC is operating) but also external, that is, depending on external data, such as environmental factors. This would move towards the goal of creating an ontological RBAC model that could be applied to pervasive, situation-aware systems. In such a scenario, an application would populate the DATA part of the ontology based on environmental triggers, such as ambient temperature, or location or other mutable characteristics of a person. For instance:

- A technician could be permitted to operate a particular equipment depending on the presence of a more senior technician in the same environment.
- In a hospital ward, an emergency situation involving a patient, as determined by the vital signs stored as individuals in the ontology, could trigger a relaxation of the usual restriction where only a nurse or doctor in charge of the patient's ward can access and modify information about that patient, and allow medical staff with suitable expertise (also defined in the ontology) who is present in the ward to help the patient.

We would like to refine this system of context constraints, for example by making it hierarchical. Context constraints are already defined in a particular class in the ESO-RBAC model; making this CONTEXT_CONSTRAINT class hierarchical would further improve the model.

The main difficulty with defining an ontology in OWL-Full is the lack of fully functioning reasoners for it. Although Jena does work with the class-individual duality that is used extensively in ESO-RBAC for describing role-role and user-role relationships, it cannot handle inferred sub-classes or class memberships. That is, Jena can only recognize direct class memberships and sub-classes in the reasoning process. Therefore, a workaround was necessary in

the version of ESO-RBAC presented herein to ensure that indirect sub-classing could be used in reasoning. Hopefully, reasoners will be developed for OWL-Full that do not require such workarounds and allow the ESO-RBAC process to be simplified to eliminate the initial step (step 0) that was found to be necessary when running ESO-RBAC with Jena.

We cannot think of any fundamental changes that would be made to the ESO-RBAC model, which does the job of inferring authorisations of users to perform actions based on user-role and role-permission relationships in an OWL ontology. The class hierarchy for the RBAC information used in ESO-RBAC is very similar to the one that we originally envisaged for a hierarchical RBAC model that could be extended to include dynamic features (context constraints). It evolved as we learnt about how reasoning works in OWL, but the original idea remains intact. We believe that ESO-RBAC has much more potential for future development than the earlier SO-RBAC. When SO-RBAC was developed, the concept of ESO-RBAC was already fully formed. SO-RBAC was developed initially because it was found to be much easier to develop an OWL ontology using OWL-DL, due to the greater capabilities of reasoners such as SWRL for this flavour of OWL. However, because SO-RBAC is developed in OWL-DL, it is tied to its roots in descriptive logic, limiting its ability to fully exploit the object-oriented hierarchical data modelling that is possible in OWL, and thus limiting its usefulness for ontological modelling of RBAC. It was really developed as a proof of concept, to show that it is possible to develop an RBAC model in which user authorization is inferred entirely by the OWL reasoner. We would therefore focus on developing ESO-RBAC by improving and refining its modelling of dynamic and situation-aware access control.

Finally, current context-aware RBAC models, including ESO-RBAC, are based on enumerated contexts. That is, the contexts are stored in rules based on fixed criteria. The permissions or roles are dynamically assigned according to rules, but the rules themselves are static. This is a serious limitation, because in many context-aware systems, it is difficult to know what contexts need to be taken into consideration, or what permissions and roles to assign according to them. Thus, a security model that changes permissions based on **heuristic rules** would be of benefit. This would not only assign the roles and permissions dynamically, but also dynamically generate the rules by which these are assigned according to context changes. The rules are then meta-programmed. The possibility of unpredictable and very frequent context changes, and the complexity or rigidity of RBAC models [47], means that some other access control models, which are not based on roles, have been proposed. These include location (M-ZONES AC [47]) and trust (TRUSTAC [121], TrustBAC [122]). Therefore, future work should consider the semantic modelling of access control in terms of using heuristic rules which control permissions:

(a) dynamically,

(b) according to environments where software applications and their data reside, and

(c) according to situations created by pervasiveness of environments and computational spaces.

Tasks (a)–(c) are very challenging, and solutions based on semantic access control mechanisms, which satisfy (a)–(c) might not be trivial. Therefore in this chapter we start using ontologies and semantic web tools, which could enrich traditional RBAC, and make it applicable to a variety of situations in pervasive computing environments. The aim is to assess whether we can create RBAC model through ontologies which address as far as possible the tasks (a)–(c) above.

References

1: Mavridis, I.; Pangalos, G., *Determining User Authorizations in Distributed Database Systems*, Advances in Informatics 2001 Proc. PCI Conf, 2001

2: Simpson, R. L., *Ensuring Patient Data Privacy, Confidentiality and Security*, Nursing Management, 1994 **25** *7*, pp18–20

3: Huston, T., *Security Issues for implementation of E-Medical Records*, Communications of ACM, 2001 **44** *9*, pp89–94

4: Adams, T.; Budden, M.; Hoare, C.; Sanderson, H., *Lessons from the central Hampshire electronic health record pilot project: issues of data protection and consent*, BMJ, 2004 **328** *7444*, pp871–874

5: *Data Protection Act 1998*, 1998, http://www.legislation.gov.uk/ukpga/1998/29/contents, accessed 16/01/2011

6: Wiederhold, G.; Biello, M.; Sarathy, V.; Qian, X., *Protecting Collaboration*, Proceedings of the National Information Systems Security Conference (NISSC'96), 1996, pp561–569

7: Zhang, L.; Ahn, G. J.; Chu, B. T., *A Role Based Delegation Framework for Healthcare Information Systems*, Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT'02), 2002, pp125–134

8: Anderson, R., *A Security Policy Model for Clinical Information Systems*, Proceedings of the IEEE Symposium on Security and Privacy, 1996, pp30–45

9: Longstaff, J.; Lockyer, M.; Nicholas, J., *A Model of Accountability, Confidentiality and Override for Healthcare and Other Applications*, Proceedings of the ACM Workshop on RBAC, 2000, pp71–76

10: *Trusted Computer Security Evaluation Criteria, DoD 5200.28-STD*, Department of Defense (US), 1985

11: Sandhu, R.; Samarati, P., *Authentication, Access Control and Audits*, ACM Computing Surveys, 1996 **28** *1*, pp241–243

12: Campbell, R. H.; Liu, Z.; Mickunas, M. D.; Naldurg, P.; Yi, S., *Seraphim: Building Dynamic Interoperable Security Architecture For Active Networks*, Open Architectures and Network Architectures and Network Programming, 2000 (OPENARCH 2000), 2000, pp55–64

13: Chang, W.-L., Yuan, S.-T., *Ambient iCare e-Services for Quality Aging: Framework and Roadmap*, Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05), 2005, pp467–470

14: *SWI Prolog*, http://www.swi-prolog.org/, accessed 14/01/2012

15: *Description of W3C Technology Stack Illustration*, 2005, http://www.w3.org/Consortium/techstack-desc.html, accessed 31/03/2012

16: *OWL Web Ontology Language Overview*, 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/, accessed 31/03/2012

17: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, 2004, http://www.w3.org/Submission/SWRL/, accessed 31/03/2012

18: Barker, S.; Stuckey, P., *Flexible Access Control Policy Specification with Constraint Logic Programming*, ACM Information and Systems Security, 2001 **6** *4*, pp501–546

19: Barker, S.; Douglas, P., *Protecting Federated Databases Using A Practical Implementation of a formal RBAC Policy*, Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04), 2004, pp523–

20: Strembeck, M.; Neumann, G., *An Approach to Engineer and Enforce Context Constraints in an RBAC Environment*, SACMAT '03, 2003, pp65–79

21: Strembeck, M.; Neumann, G., *An Integrated Approach to Engineer and Enforce Context Constraints in RBAC*, ACM Informations & Systems Security, 2004 **7** *3*, pp392–427
22: Kataria, P., Macfie, A., Juric, R., Madani, K., *Ontology for Supporting Context Aware Applications for the Intelligent Hospital Ward*, Journal of Integrated Design & Process Science, 2008 **12** *3*, pp35–44
23: Macfie, A., Kataria, P., Koay, N., Dagdeviren, H., Juric, R., Madani, K., *Ontology Based Access Control Derived From Dynamic RBAC and its Context Constraints*, Proceedings of the 11th International Conference on Integrated Design and Process Technology (IDPT'08), 2008
24: *Protégé*, http://protege.stanford.edu/, accessed 31/01/2012
25: Slevin, L.; Macfie, A., *Role Based Access Control for a Medical Database*, 11th IASTED International Conference on Software Engineering and Applications (SEA 2007), 2007, pp226–233
26: Castano, S.; Fugini, M.; Martella, G.; Samarati, P., *Database Security*, Addison-Wesley, 1995
27: Ferraiolo, D.; Kuhn, R., *Role-Based Access Controls*, 15th NIST-NCSC National Computer Security Conference, 1992, pp554–563
28: Samarati, P.; Jajodia, S., *Data Security*, from Webster, J.G., Wiley Encyclopedia of Electrical and Electronics Engineering, John Wiley & Sons, 1999
29: Sandhu, R.; Coyne, E.; Feinstein, H.; Youman, C., *Role Based Access Control Models*, IEEE Computer, 1996 **29** *2*, pp38–47
30: Clark, K., *Negation as Failure*, from H. Gallaire and J. Minker, Eds., Logic and Databases, New York: Plenum Press, 1978, pp293–322
31: Ferraiolo, D.; Kuhn, D.; Chandramouli, R., *Role-Based Access Control*, Artech House, 2003
32: Kern, A.; Walhorn, C., *Rule Support for Role Based Access Control*, Proc. ACM SACMAT, 2005, pp130–138
33: Sandhu, R.; Bhamidipati, V.; Coyne, R.; Ganta, S.; Youman, C., *The ARBAC97 Model for Role-Based Administration of Roles:Preliminary Description and Outline*, IEEE Computer, 1997 **29** *2*, pp38–47
34: Ferraiolo, D. F.; Sandhu, R.; Gavrila, S.; Kyhn, D. R.; Chandramouli, R., *Proposed NIST Standard for Role-Based Access Control*, ACM Transactions on Information Systems Security, 2001 **4** *3*, pp224–274
35: Lupu, E. C.; Marriott, D. A.; Sloman, M. S.; Yialelis, N., *A Policy-Based Role Framework for Access Control*, Proc. ACM SACMAT Workshop on RBAC, 1996
36: Ferraiolo, D.; Barkley, J.; Kuhn, R., *A Role-Based Access Control Model and Reference Implementation Within a Corporate Intranet*, ACM Transactions on Information and System Security, 1999 **2** *1*, pp34–64
37: Sohr, K.; Drouineaud, M.; Ahn, G. J., *Formal Specification of Role-based Security Policies for Clinical Information Systems*, Proceedings of the 2005 ACM symposium on Applied computing (SAC'05), 2005, pp332–339
38: Wilikens, M.; Feriti, S.; Sanna, A.; Masera, M., *A Context-Related Authorization and Access Control Method Based on RBAC: A case study from the health care domain*, Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT'02), 2002, pp117–124
39: Zhang, L.; Ahn, G. J.; Chu, B. T., *A rule-based framework for role based delegation*, Proceedings of the sixth ACM symposium on Access control models and technologies (SAMCAT'01), 2001, pp153–162
40: Potamias, G.; Tsiknakis, M.; Katehakis, D.; Karabela, E.; Moustakis, V.; Orphanoudakis, S., *Role-based Access to Patient Clinical Data: The InterCare Approach in the Region of Crete*, Proc. MIE and GMDS, 2000, pp1074–1079
41: Poole, J.; Barkley, J.; Brady, K.; Cincotta, A.; Salamon, W., *Distributed Communications methods and Role-Based Access Control for use in Healthcare Applications*, Proc. CHIN Summit., 1995

42: Mavridis, I.; Georgiadis, C.; Pangalos, G.; Khair, M., *Access Control Based on Attribute Certificates for Medical Intranet Applications*, Journal of Medical Internet Research, 2001 **3** *1, e9*
43: Moffett, J.; Lupu, E. C., *The Uses of Role Hierarchies in Access Control*, Proc. ACM Workshop on RBAC, 1999, pp153–160
44: Notargiacomo, L., *Role-Based Access Control in ORACLE7 and Trusted ORACLE7*, ACM RBAC Workshop, 1996
45: *PostgreSQL*, http://www.postgresql.org/, accessed 01/04/2012
46: Hu, J.; Weaver, A. C., *Dynamic, Context-Aware Access Control for Distributed Healthcare Applications*, Proceedings of the First Workshop on Pervasive Security, Privacy and Trust (PSPT2004), 2004
47: White, M.; Jennings, B.; van der Meer, S., *User-Centric Adaptive Access Control and Resource Configuration for Ubiquitous Computing Environments*, Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'05), 2005, pp349–
48: Damiani, M. L.; Bertino, E.; Catania, B.; Perlasca, P., *GEO-RBAC: A Spatially Aware RBAC*, ACM Transactions on Information and System Security, 2007 **10** *1*, pp29–37
49: Bertino, E.; Bonatti, P. A.; Ferrari, E., *TRBAC: A Temporal Role-Based Access Control Model*, ACM Transactions on Information and System Security, 2001 **4** *3*, pp191–223
50: Joshi, J. B. D.; Bertino, E.; Latif, U.; Ghafoor, A., *A Generalized Temporal Role-Based Access Control Model*, IEEE Transactions on Knowledge and Data Engineering, 2005 **17** *1*, pp4–23
51: Bacon, J.; Moody, K.; Yao, W., *Access Control and Trust in the Use of Widely Distributed Services*, Software: Practice and Experience, 2003 **33** *4*, pp375–394
52: Bacon, J.; Moody, K.; Yao, W., *A Model of OASIS Role-Based Access Control and Its Support for Active Security*, ACM Transactions on Information and System Security (TISSEC), 2002 **5** *4*, pp492–540
53: Belokosztololszky, A.; Eyers, D. M.; Moody, K., *Policy Contexts: Controlling Information Flow in Parameterized RBAC*, Policy 2003: IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003, pp4–6
54: Tolone, W.; Ahn, G. J.; Pai, T.; Hong, S. P., *Access Control in Collaborative Systems*, ACM Computing Surveys, 2005 **37** *1*, pp29–41
55: Corradi, A.; Montanari, R.; Tibaldi, D., *Context-based Access Control for Ubiquitous Service Provisioning*, Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), 2004, pp444–451
56: Covington, M. J.; Long, W.; Srinivasan, S.; Dey, A. K.; Ahamad, M.; Abowd. G. D., *Securing Context-Aware Applications Using Environment Roles*, Proceedings of the sixth ACM symposium on Access control models and technologies (SACMAT'01), 2001, pp10–20
57: Covington, M. J.; Moyer, M. J.; Ahamad, M., *Generalized Role-Based Access Control for Securing Future Applications*, National Information Systems Security Conference (NISSC'00), 2000
58: Finnigan, P., *Oracle Row Level Security: Part 1*, 2003, http://www.symantec.com/connect/articles/oracle-row-level-security-part-1, accessed 14/01/2012
59: Thomas, R. K., *Team-Based Access Control: A Primitive for Applying Role-Based Access Controls in Collaborative Environments*, Proceedings of the ACM Workshop on Role-Based Access Control, 1997, pp13–19
60: Thomas, R. K., Sandhu, R. S., *Task-Based Authorization Control: A Family of models for Active and Enterprise Oriented Authorization Management*, Proceedings of the IFIP WG11.3 Workshop on Database Security, 1997, pp166–181

61: Bertino, E.; Catania, B.; Ferrari, E.; Perlasca, P., *A Logical Framework for Reasoning about Access Control Models*, ACM Transactions on Information and System Security, 2003 **6** *1*, pp71–127
62: Greco, S.; Leone, N.; Rullo, P., *COMPLEX: An Object-Oriented Logic Programming System*, IEEE Transactions on Knowledge and Data Engineering, 1992 **4** *4*, pp344–359
63: *Datalog*, , http://en.wikipedia.org/wiki/Datalog, accessed 31/10/2012
64: Seitz, L.; Pierson, J. M.; Brunie, L., *Semantic Access Control for Medical Applications in Grid Environments*, 2003
65: Feuerstein, S., *Oracle PL/SQL Programming: Guide to Oracle8i Features*, O'Reilly Media, 1999
66: *Veil*, http://veil.projects.postgresql.org/curdocs/index.html, accessed 14/01/2012
67: Satoshi, H; Kudo, M., *XML Access Control Language: Provisional Authorization for XML Documents*, 2002, http://www.research.ibm.com/trl/projects/xml/xss4j/docs/xacl-spec.html, accessed 07/02/2012
68: *OASIS eXtensible Access Control Markup Language (XACML) TC*, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml, accessed 14/01/2012
69: *MOSQUITO: Mobile Workers' Secure Business Applications in Ubiquitous Environments*
70: Chandramouli, R., *Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks*, 5th ACM workshop on Role-based Access Control (RBAC 2000), 2000, pp11–18
71: Vuong, N.; Smith, G. S.; Deng, Y., *Managing Security Policies in a Distributed Environment Using eXtensible Markup Language (XML)*, SAC '01 Proceedings of the 2001 ACM symposium on Applied computing (SAC'01), 2001, pp405–411
72: Bertino, E.; Castano, S.; Ferrari, S., *Securing XML Documents with Author-X*, IEEE Internet Computing, 2001 **5** *3*, pp21–31
73: Bertino, E.; Correndo, G.; Ferrari, E.; Mella, G., *An Infrastructure for Managing Secure Update Operations on XML Data*, Proceedings of the eighth ACM symposium on Access control models and technologies (SACMAT'03), 2003, pp110–122
74: Bhatti, R.; Joshi, J.; Bertino, E.; Ghafoor, A., *Access Control in Dynamic XML-based Web Services with X-RBAC*, Proceedings of the 2003International Conference on Web Services (ICWM'03), 2003, pp243–249
75: Bhatti, R.; Bertino, E.; Ghafoor, A., *A Trust-Based Context-Aware Access Control Model for Web Services*, Proceedings of the IEEE International Conference on Web Services (ICWS'04), 2004, pp184–191
76: Joshi, J. B. D.; Bhatti, R.; Bertino, E.; Ghafoor, A., *Access Control Language for Multidomain Environments*, IEEE Internet Computing, 2004 **8** *6*, pp40–50
77: Bhatti, R.; Bertino, E.; Ghafoor, A., *X-FEDERATE: A Policy Engineering Framework for Federated Access Management*, IEEE Transactions on Software Engineering, 2006 **323** *5*, pp330–346
78: He, H., Wong, R, *A Role-Based Access Model for XML Repositories*, Proceedings of the First International Conference on Web Information Systems Engineering, 2000, pp138–145
79: Stoupa, K.; Vakali, A., *An XML-Based Language for Access Control Specifications in an RBAC Environment*, IEEE, 2003, pp1717–1722
80: Yang, C.; Zhang, C., *Secure Web-based Applications with XML and RBAC*, IEEE Systems, Man and Cybernetics Society Information Assurance Workshop, 2003, pp276–281
81: Bhatti, R.; Ghafoor, A.; Bertino, E., *X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control*, ACM Transactions on Information and System Security, 2005 **8** *2*, pp191–233
82: Bhatti, R.; Shafiq, B.; Bertino, E.; Ghafoor, A.; Joshi, J., *X-GTRBAC Admin: A Decentralized Administration Model*

for Enterprise-Wide Access Control, ACM Transactions on Information and System Security, 2005 **8** *4*, pp388–423
83: Yang, L.; Ege, R., *Mediation Security Specification and Enforcement for Heterogeneous Databases*, Proceedings of the ACM symposium on Applied computing (SAC'05), 2005, pp354–358
84: Warner, J.; Atluri, V.; Vaidya, J.; Mukkamala, R., *Using Semantics for Automatic Enforcement of Access Control Policies among Dynamic Coalitions*, Proceedings of the twelfth ACM symposium on Access control models and technologies (SACMAT'07), 2007
85: Finance, B.; Medjdoub, S.; Pucheral, P., *The Case for Access Control on XML Relationships*, Proceedings of the 14th ACM international conference on Information and knowledge management (CIKM'05), 2005, pp107–114
86: Al-Bouna, B.; Chbeir, R., *Multimedia-Based Authorization and Access Control Policy Specification*, Proceedings of the 3rd ACM workshop on Secure web services (SWS'06), 2006, pp61–68
87: Chadwick D.W., Otenko A., Ball E, *Implementing role based access controls using X.509 attribute certificates*, IEEE Internet Computing, 2003 **7**, pp62–69
88: *X.509*, http://en.wikipedia.org/wiki/X.509, accessed 06/11/2012
89: Brostoff, S.; Sasse, M. A.; Chadwick, D.; Cunningham, J.; Mbanaso, U.; Otenko, S., "*R-What?" Development of a Role-Based Access Control (RBAC) Policy-Writing Tool for e-Scientists*, Software: Practice and Experience, 2005 **35**, pp835–856
90: McGuiness, D.; Harmelen, F., *OWL Web Ontology Overview/Guide*, 2004, http://www.w3.org/TR/owl-features/, accessed 31/01/2012
91: Beckett D.; Mcbride, B., *RDF/XML Syntax Specification (Revised)*, 2004, http://www.w3.org/TR/REC-rdf-syntax/, accessed 31/01/2012
92: Pan, C-C.; Mitra, P.; Lui, P., *Semantic Access Control for Information Interoperation*, Proceedings of the eleventh ACM symposium on Access control models and technologies (SACMAT'06), 2006, pp237–246
93: Wu, D.; Lin, J.; Dong, Y.; Zhu, M., *Using Semantic Web Technologies to Specify Constraints of RBAC*, Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'05), 2005, pp543–545
94: Wu, D.; Chen, X.; Lin, J.; Zhu, M., *Ontology-Based RBAC Specification for Interoperation in Distributed Environment*, Proceedings of the Asian Semantic Web Conference (ASWC'06), 2006, pp179–190
95: Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M., *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, 2004, http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, accessed 31/01/2012
96: Priebe, T.; Dobmeier, W.; Kamprath, N., *Supporting Attribute-based Access Control with Ontologies*, Proceedings of the First International Conference on Availability, Reliability and Security (ARES'06), 2006, pp465–472
97: *SPARQL Query Language for RDF*, http://www.w3.org/TR/rdf-sparql-query/, accessed 14/01/2012
98: Finin, T.; Joshi, A.; Kagal, L.; Niu, J.; Sandhu, R.; Winsborough, W.; Thuraisingham, B., *Using OWL to Model Role Based Access Control*, Ebiquity Laboratory, University of Maryland, Baltimore, 2008
99: Finin, T.; Joshi, A.; Kagal, L.; Niu, J.; Sandhu, R.; Winsborough, W.; Thuraisingham, B., *ROWLBAC — Representing Role Based Access Control in OWL*, SACMAT'08, 2008, pp73–82
100: Finin, T.; Joshi, A.; Kagal, L.; Niu, J.; Sandhu, R.; Winsborough, W.; Thuraisingham, B., *Role Based Access Control and OWL*, Proceedings of the fourth OWL: Experiences and Directions Workshop, 2008
101: Heilili, N.; Chen, Y.; Zhao, C.; Luo, Z. X., *An OWL-Based Approach for RBAC with Negative Authorization*,

Knowledge, Science, Engineering and Management: Lecture Notes in Computer Science, 2006 **4092** *2006*, pp164–175
102: Cirio, L.; Cruz, I.; Tamassia, R., *A Role and Attribute Based Access Control System Using Semantic Web Technologies*, Proceedings of the 2007 OTM Confederated international conference on the move to meaningful internet systems (OTM'07), 2007, pp1256–1266
103: He, Z.; Wu, L.; Li, H.; Lai, H.; Hong, Z., *Semantics-based Access Control Approach for Web Service*, Journal of Computers, 2011 **6** *6*
104: Alcaraz Calero, J.M.; Martinez Pérez, G.; Gomez Skarmeta, A.F., *Towards an authorisation model for distributed systems based on the Semantic Web*, IET Information security, 2010 **4** *4*, pp411–421
105: *Common Information Model (CIM)*, http://dmtf.org/standards/cim, accessed 24/01/2012
106: Cadenhead, T.; Kantarcioglu, M.; Thuraisingham, B.M, *Scalable and Efficient Reasoning for Enforcing Role-Based Access Control*, Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy (DBSec'10), 2010, pp209–224
107: Coma, C.; Cuppens-Boulahia, N.; Cuppens, F.; Cavalli, A.R., *Context Ontology for Secure Interoperability*, Third International Conference on Availability, Reliability and Security (ARES'08), 2008, pp821–827
108: Cuppens, F.; Miège, A., *Modelling Contexts in the Or-BAC Model*, Proceedings of the 19th Computer Security Applications Conference (ACSAC'03), 2003
109: Toninelli, A; Montanari, R.; Kagal, L.; Lassila, O., *A Semantic Context-Aware Access Control Framework for Secure Collaborations in Pervasive Computing Environments*, International Semantic Web Conference (ISWC'06), 2006, pp473–486
110: O'Connor, M., *Protégé: SWRLTab*, 2007, http://protegewiki.stanford.edu/wiki/SWRLTab, accessed 31/01/2012
111: O'Connor, M.; Das, A., *SQWRL: a Query Language for OWL*, OWL: Experiences and Directions, 6th International Workshop (OWLED'09), 2009
112: *SQWRL*, http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL, accessed 01/02/2012
113: Yuhana, U. L., *SWRL and SQWRL*, 2008, http://yuhanaresearch.wordpress.com/2008/04/23/swrl-and-sqwrl/, accessed 14/01/2011
114: *Pellet: OWL 2 Reasoner for Java*, 2011, http://clarkparsia.com/pellet/, accessed 01/02/2012
115: *Jess, the Rule Engine for the Java Platform*, http://www.jessrules.com/, accessed 01/02/2012
116: *SWRLJessTab*, http://protege.cim3.net/cgi-bin/wiki.pl?SWRLJessTab, accessed 01/02/2012
117: Horridge, M., *Protégé: OWLViz*, 2010, http://protegewiki.stanford.edu/wiki/OWLViz, accessed 01/04/2012
118: *Apache Jena: Welcome to Jena*, http://incubator.apache.org/jena/, accessed 01/02/2012
119: *Jena Framework*, 2011, http://en.wikipedia.org/wiki/Jena_%28framework%29, accessed 01/02/2012
120: *SWRL Language FAQ*, 2011, http://protege.cim3.net/cgi-bin/wiki.pl?SWRLLanguageFAQ, accessed 01/02/2012
121: Almenárez, F.; Marín, A.; Campo, C.; García, C., *TrustAC: Trust-Based Access Control for Pervasive Devices*, SPC 2005, LNCS, 2005, pp225–238
122: Chakraborty, S.; Ray, I., *TrustBAC — Integrating Trust Relationships into the RBAC Model for Access Control in Open Systems*, Proceedings of the eleventh ACM symposium on Access control models and technologies (SACMAT'06), 2006, pp49–58
123: Macfie, A., Juric, R., Madani, K., *Research Issues in Access Control for Pervasive Healthcare*, Proceedings of the 11th International Conference on Integrated Design and Process Technology (IDPT'08), 2008
124: Macfie, A.; Juric, R., *SO-RBAC Reasoning Process*, SDPS 2012

125: Macfie, A., Juric, R., Slevin, L., *Implementing DRBAC for a Medical Database*, 2007

126: Macfie, A., *Implementing Dynamic RBAC for a Medical Database: Test Results Using Oracle*, 2007

Appendices

Appendix I: Publications

This is a full list of the publications generated from this research.

- Kataria, P.; Macfie, A.; Juric, R.; Madani, K. (2008), *Ontology for Supporting Context Aware Applications for the Intelligent Hospital Ward*, in Proceedings of the 11th International Conference on Integrated Design and Process Technology, IDPT 2007 (Taichung, Taiwan, June 1–6, 2008). [22]
- Macfie, A.; Kataria, P.; Koay, N.; Dagdeviren, H.; Juric, R.; Madani, K. (2008), *Ontology Based Access Control Derived From Dynamic RBAC and its Context Constraints*, in Proceedings of the 11th International Conference on Integrated Design and Process Technology, IDPT 2007 (Taichung, Taiwan, June 1–6, 2008). [23]
- Macfie, A.; Juric, R.; Madani, K. (2008), *Research Issues in Access Control for Pervasive Healthcare*, in Proceedings of the 11th International Conference on Integrated Design and Process Technology, IDPT 2007 (Taichung, Taiwan, June 1–6, 2008). [123]
- Slevin, L.; Macfie, A. (2007), *Role Based Access Control for a Medical Database*, in Proceedings of the 11th IASTED Conference on Software Engineering Applications (Cambridge, MA, US, November 19–21, 2007). [25]
- Macfie, A.; Juric, R. (2012), *SO-RBAC Reasoning Process* in Proceedings of the SDPS 2012 Conference, June 2012, Berlin Germany [124]
- Macfie, A.; Juric, R. (2014), *Modeling Dynamic RBAC with OWL and SWRL*, under review for the 47th HICSS Conference *http://www.hicss.hawaii.edu/hicss_47/apahome47.htm*, January 2014
- Macfie, A.; Juric, R.; Paurobally, S. (2014), *Semantic Access Control in Medical Databases*, to be submitted to the Journal of Health Systems, *http://www.palgrave-journals.com/hs/index.html*
- Macfie, A.; Juric, R. (2013), *Enhanced Semantic and Ontology Based RBAC*, under review for the Journal of SDPS *http://www.iospress.nl/journal/journal-of-integrated-design-process-science/*
- Macfie, A.; Juric, R. (2014), *Implementing DRBAC for a Medical Database*, 2014 [125] to be submitted to Advances in Engineering Software, *http://www.sciencedirect.com/science/journal/09659978*

Macfie, A. (2014), *Implementing Dynamic RBAC for a Medical Database: Test Results Using Oracle* [126] to be submitted to the Journal of Software Engineering and Practices

Appendix II: Prolog Rules in Static RBAC

```
% inclusion of equal-status roles (this is done so that day/night duty roles
% do not have to be defined twice)
included_in(R1,R1).
included_in(R1,R2) :- is_a(R1,R2).
included_in(R1,R3) :- is_a(R1,R2),
                      included_in(R2,R3).


% Role hierarchies

senior_to(R1,R1) :- d_s(R1,_).
senior_to(R1,R1) :- d_s(_,R1).
senior_to(R1,R2) :- d_s(R1,R2).
senior_to(R1,R2) :- d_s(R1,R3), senior_to(R3,R2).



% Inheritance paths


inherits_rpa(R1,R1,_,_).

inherits_rpa(R2,R3,P,O) :- senior_to(R1,R2),
                           senior_to(R3,R4),
                           inherits_rpa_path(R1,R4,P,O).

% Access control rules structure

rpa_full(R1,P,O) :- included_in(R1,R2),
                    senior_to(R2,R3),
                    rpa(R3,P,O),
                    inherits_rpa(R2,R3,P,O).

permittable(U,P,O) :- permittable(U,P,O,R).

permittable(U,P,O,R) :- ura(U,R)
                            rpa_full(R,P,O).

permitted(U,P,O) :-    ura(U,R),
                       permitted(U,P,O,R).

permitted(U,P,O,R) :-  currently_active(U,R,_),
                       permittable(U,P,O,R).


d_rpa_full(R1,P,O) :- included_in(R1,R2),
                      senior_to(R3,R2),
                      d_rpa(R3,P,O).


denied(U,P,O) :- ura(U,R),
                 d_rpa_full(R,P,O).


authorizable(U,P,O) :-   ura(U,R),
                               authorizable(U,P,O,R).
authorizable(U,P,O,R) :- permittable(U,P,O,R),
                               not(denied(U,P,O)).

authorized(U,P,O) :-   ura(U,R),
                       authorized(U,P,O,R).

authorized(U,P,O,R) :- permitted(U,P,O,R),
                       not(denied(U,P,O)).
```

Appendix III: Prolog Rules in Dynamic RBAC

```
% inclusion of equal-status roles (this is done so that day/night duty roles
% do not have to be defined twice)
included_in(R1,R1).
included_in(R1,R2) :- is_a(R1,R2).
included_in(R1,R3) :- is_a(R1,R2),
                      included_in(R2,R3).


% Role hierarchies

senior_to(R1,R1) :- d_s(R1,_).
senior_to(R1,R1) :- d_s(_,R1).
senior_to(R1,R2) :- d_s(R1,R2).
senior_to(R1,R2) :- d_s(R1,R3), senior_to(R3,R2).



% Inheritance paths


inherits_rpa(R1,R1,_,_).

inherits_rpa(R2,R3,P,O) :- senior_to(R1,R2),
                           senior_to(R3,R4),
                           inherits_rpa_path(R1,R4,P,O).

% Access control rules structure

rpa_full(R1,P,O) :- included_in(R1,R2),
                    senior_to(R2,R3),
                    rpa(R3,P,O),
                    inherits_rpa(R2,R3,P,O).

permittable(U,P,O) :- permittable(U,P,O,R).

permittable(U,P,O,R) :- ura(U,R)
                            rpa_full(R,P,O).

% currently_active
currently_active(U,R1,D1) :- activate(U,R1,D1,Password),
                    password(U,Password),
                    ura(U,R1),
                    (
                      not(deactivate(U,R1,_));
                      deactivate(U,R1,D2),
                      date_time_stamp(D1,T1),
                      date_time_stamp(D2,T2),
                      T2 < T1
                    ),
                    not(inconsistent_dsd(U,R1,D1)).

permitted(U,P,O) :-    ura(U,R),
                       permitted(U,P,O,R).

permitted(U,P,O,R) :- currently_active(U,R,_),
                       not(fail_context_constraint(U,R,P,O)),
                       permittable(U,P,O,R).


d_rpa_full(R1,P,O) :- included_in(R1,R2),
                      senior_to(R3,R2),
                      d_rpa(R3,P,O).


denied(U,P,O) :- ura(U,R),
                 d_rpa_full(R,P,O).


authorizable(U,P,O) :-   ura(U,R),
                               authorizable(U,P,O,R).
authorizable(U,P,O,R) :- permittable(U,P,O,R),
```

```
                        not(denied(U,P,O)).

authorized(U,P,O) :-    ura(U,R),
                        authorized(U,P,O,R).

authorized(U,P,O,R) :- permitted(U,P,O,R),
                        not(denied(R,P,O)).


% separation of duties

% dsd: dynamic separation of duties
dsd_conflict(R1,R2) :- dsd(R1,R2), !.
dsd_conflict(R1,R2) :- dsd(R2,R1).

% ssd: static separation of duties
% This is not modelled in the Prolog implementation
ssd_conflict(R1,R2) :- ssd(R1,R2), !.
ssd_conflict(R1,R2) :- ssd(R2,R1).

inconsistent_ssd(U,R1) :- ura(U,R1),
                          ssd(R1,R2),
                          ura(U,R2).

inconsistent_dsd(U,R1,D1) :- activate(U,R2,D2,Password),
                             password(U,Password),
                             dsd_conflict(R1,R2),
                             date_time_stamp(D1,T1),
                             (
                               not(deactivate(U,R2,_));
                               deactivate(U,R2,D3),
                               date_time_stamp(D3,T3),
                               T3 < T1
                             ),
                             date_time_stamp(D2,T2),
                             T2 =< T1.



% Evaluation of context constraints: from Strembeck & Neuman with some names changed
% context constraints inherit down the hierarchy
applied_cc(R1,P,O,CC) :-
  associated_cc(R3,P,O,CC),
  senior_to(R3,R2),
  included_in(R1,R2).

applied_cc(R,P,O,CC) :-
  associated_cc(R,P,O,CC).

% whether context constraints are violated: negates context_condition
% violated(ContextConstraint,User,Permission,Object).
violated(CC,U,P,O) :- not(context_condition(CC,U,P,O)).

fail_context_constraint(U,R,P,O) :-
  applied_cc(R,P,O,CC),
  violated(CC,U,P,O).
```

Appendix IV: Prolog Facts in Static RBAC

```
% role(Role).
% object(Object).
% user(Username,LastName,FirstName,Address,DOB).
% password(Username,Password).
% d_s(SeniorRole,JuniorRole).
% is_a(InnerRole,OuterRole).
% inherits_rpa_path(SeniorRole,JuniorRole,Permission,Object).
% rpa(Role,Permission,Object).
% ura(User,Role).
% ssd(Role1,Role2).
% dsd(Role1,Role2).

% activate(User,Role,DateTime,Password).
% DateTime is of form date(Year, Month, Day, Hour, Min, Sec, Offset, TimeZone, DST)
% e.g. date(2006, 8, 23, 08, 15, 0, 0, 'BST', true)
% nurse_ward(User,Ward).



% role
% role(Role).
role(consultant).
role(specialist_registrar).
role(senior_house_officer).
role(senior_house_officer_day).
role(senior_house_officer_night).
role(house_officer).
role(house_officer_day).
role(house_officer_night).

role(specialist_nurse).
role(sister).
role(sister_day).
role(sister_night).
role(staff_nurse).
role(staff_nurse_day).
role(staff_nurse_night).
role(student_nurse).
role(student_nurse_day).
role(student_nurse_night).

role(senior_data_manager).
role(junior_data_manager).
role(receptionist).
role(manager).


role(day_duty).
role(night_duty).




% object
% object(Object).
object(ward(Ward_ID,Type,Ward_Capacity)).
object(room(Room_ID,Ward_ID,Type,Bed_Capacity)).
object(bed(Bed_ID,Room_ID,Type)).
object(patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).
object(diagnosis(Diagnosis_code,Illness_name,Usual_Symptoms)).
object(ae_consultation(Cons_Number,Cons_Date,Cons_Description,Patient_ID,Doctor_ID)).
object(patient_diagnosis(Patient_Diagnosis_Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_
Number,Diagnosis_Code)).



% user(Username,LastName,FirstName,Address,DOB).
user(dr_sugar,'Sugar','Ed','1 Montgomery Ave','12/06/1975').
user(dr_python,'Python','Adam','45 Escort Road','24/01/1950').
user(dr_edmonds,'Edmonds','Sophie','49 Convent Gardens','10/10/1968').
user(dr_bowie,'Bowie','Diane','253 Kings Road','02/03/1962').
```

```
user(dr_peters,'Peters','Peter','59 Monkety Crescent','19/01/1980').
user(dr_davies,'Davies','Sheena','10 Auchtermuchty Way','15/02/1979').
user(dr_williams,'Williams','Lucie','23 Monkswood Drive','15/07/1977').
user(dr_jones,'Jones','John','The Manse, Church Lane','18/07/1977').
user(dr_evans,'Evans','Renate','3 Geering Road','12/03/1970').
user(dr_fish,'Fish','Michael','The Vane, Weatherby','28/12/1955').
user(dr_ghosh,'Ghosh','Chandra','10 Kennington Road','11/07/1959').
user(dr_kellett,'Kellett','James','104 The Vale','15/02/1959').

user(miss_jacobson,'Jacobson','Lucinda','14 The Mansion','01/02/1969').
user(mrs_jones,'Jones','Hannah','13 Consort Road','15/05/1955').
user(mr_kenning,'Kenning','Stephen','10 Roadrunner Crescent','13/01/1977').
user(miss_strand,'Strand','Jasmine','The Lodge, Linden Avenue','15/06/1987').
user(mrs_canning,'Canning','Elizabeth','100 Western Road','22/03/1969').
user(mr_clarkson,'Clarkson''Jeremy','43 Vroom Vroom Road','30/09/1962').
user(miss_lewis,'Lewis','Christine','16 Trent Drive','13/05/1980').

user(miss_jackson,'Jackson','Lisa','56 Restorick Road','12/09/1975','queen').
user(mrs_james,'James','Wendy','40 Transvision Road','07/05/1966','vamp').
user(miss_darch,'Darch','Ruth','31 Finstock Street','21/06/1979','woodstock').
user(mr_lewis,'Lewis','Donald','15 Montana Lane','29/12/1980','bronze').
user(miss_davies,'Davies','Caroline','10 The Avenue','17/09/1971','cruise').

user(mrs_lewis,'Lewis','Charlotte','20 High Road','06/07/1974').
user(mr_davies,'Davies','Jonathan','15 Low Road','14/07/1959').
user(mr_minnow,'Minnow','Robert','5 Montrose Place','08/07/0966').
user(mr_avery,'Avery','Caspar','13 Cod Street','15/08/1981').
user(mr_mctaggart,'McTaggart','James','10 Fortean Street','21/02/1977').



% password(Username,Password).
password(dr_sugar,'desk').
password(dr_python,'chair').
password(dr_edmonds,'window').
password(dr_bowie,'brick').
password(dr_peters,'mother').
password(dr_davies,'tennis').
password(dr_williams,'file').
password(dr_jones,'cricket').
password(dr_evans,'dragon').
password(dr_fish,'cock').
password(dr_ghosh,'onion').
password(dr_kellett,'thadeus').


password(miss_jacobson,'re$t').
password(mrs_jones,'carlena').
password(mr_kenning,'walnut').
password(miss_strand,'c001ie').
password(mrs_canning,'compile').
password(mr_clarkson,'wheeler').
password(miss_lewis,'mcginty').

password(miss_jackson,'queen').
password(mrs_james,'vamp').
password(miss_darch,'woodstock').
password(mr_lewis,'bronze').
password(miss_davies,'cruise').

password(mrs_lewis,'cream').
password(mr_davies,'rookie').
password(mr_minnow,'little_fish').
password(mr_avery,'fern').
password(mr_mctaggart,'jimmy').


% direct seniority
% d_s(SeniorRole,JuniorRole).
d_s(consultant,specialist_registrar).
d_s(specialist_registrar,senior_house_officer).
d_s(senior_house_officer,house_officer).

d_s(specialist_nurse,sister).
```

```
d_s(sister,staff_nurse).
d_s(staff_nurse,student_nurse).

d_s(senior_data_manager,junior_data_manager).

d_s(manager,receptionist).
d_s(manager,consultant).
d_s(manager,specialist_nurse).
d_s(manager,senior_data_manager).


% is_a relationships
% is_a(InnerRole,OuterRole).
is_a(student_nurse_day,student_nurse).
is_a(student_nurse_night,student_nurse).
is_a(staff_nurse_day,staff_nurse).
is_a(staff_nurse_night,staff_nurse).
is_a(sister_day,sister).
is_a(sister_night,sister).

is_a(student_nurse,nurse).
is_a(staff_nurse,nurse).
is_a(sister,nurse).
is_a(specialist_nurse,nurse).

is_a(student_nurse,nurse).
is_a(staff_nurse,nurse).
is_a(sister,nurse).
is_a(specialist_nurse,nurse).

is_a(house_officer_day,house_officer).
is_a(house_officer_night,house_officer).
is_a(senior_house_officer_day,senior_house_officer).
is_a(senior_house_officer_night,senior_house_officer).

is_a(house_officer,doctor).
is_a(senior_house_officer,doctor).
is_a(specialist_registrar,doctor).
is_a(consultant,doctor).

is_a(junior_data_manager,data_manager).
is_a(senior_data_manager,data_manager).

is_a(receptionist,administrator).
is_a(manager,administrator).


is_a(student_nurse_day,day_duty).
is_a(staff_nurse_day,day_duty).
is_a(sister_day,day_duty).
is_a(house_officer_day,day_duty).
is_a(senior_house_officer_day,day_duty).

is_a(student_nurse_night,night_duty).
is_a(staff_nurse_night,night_duty).
is_a(sister_night,night_duty).
is_a(house_officer_night,night_duty).
is_a(senior_house_officer_night,night_duty).


% inheritance paths: currently everything is inherited across whole hierarchies,
% except that manager does not inherit from anyone except receptionist.
% inherits_rpa_path(SeniorRole,JuniorRole,Permission,Object).
inherits_rpa_path(consultant,house_officer,_,_).
inherits_rpa_path(specialist_nurse,student_nurse,_,_).
inherits_rpa_path(senior_data_manager,junior_data_manager,_,_).
inherits_rpa_path(manager,receptionist,_,_).


% rpa
% rpa(Role,Permission,Object).
rpa(house_officer,select,ward(Ward_ID,Type,Ward_Capacity)).
rpa(house_officer,select,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(house_officer,select,bed(Bed_ID,Room_ID,Type)).
```

```
rpa(house_officer,select,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).
rpa(house_officer,select,diagnosis(Diagnoses_code,Illness_name,Usual_Symptoms)).
rpa(house_officer,select,user(Username,LastName,FirstName,Address,DOB)).
rpa(house_officer,select,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,Patient_ID,Doctor_ID)).
rpa(house_officer,select,patient_diagnosis(Patient_Diagnosis_
Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_Number,Diagnosis_Code)).

rpa(senior_house_officer,update,diagnosis(Diagnoses_code,Illness_name,Usual_Symptoms)).
rpa(senior_house_officer,update,ae_consultation(Cons_Number,Cons_Date,Cons_
Description,Patient_ID,Doctor_ID)).
rpa(senior_house_officer,update,patient_diagnosis(Patient_
Diagnosis_Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_Number,Diagnosis_Code)).

rpa(specialist_registrar,insert,patient_diagnosis(Patient_
Diagnosis_Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_Number,Diagnosis_Code)).

rpa(consultant,insert,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,Patient_ID,Doctor_ID)).

rpa(student_nurse,select,ward(Ward_ID,Type,Ward_Capacity)).
rpa(student_nurse,select,room(Room_ID,Ward_ID,Type,bed_capacity)).
rpa(student_nurse,select,bed(Bed_ID,Room_ID,Type)).
rpa(student_nurse,select,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).
rpa(student_nurse,select,user(Username,LastName,FirstName,Address,DOB)).

rpa(staff_nurse,update,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)). %
should be able to put them in a ward(Ward_ID,Type,Ward_Capacity)
rpa(staff_nurse,select,diagnosis(Patient_diagnosis_Number,
Diagnoses_code,Illness_name,Usual_Symptoms)).
rpa(staff_nurse,select,user(Username,LastName,FirstName,Address,DOB)).
rpa(staff_nurse,select,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,Patient_ID,Doctor_ID)).
rpa(staff_nurse,select,patient_diagnosis(Patient_Diagnosis_
Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_Number,Diagnosis_Code)).

rpa(sister,update,patient_diagnosis(Diagnoses_code,Illness_name,Usual_Symptoms)).

rpa(specialist_nurse,update,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,Doctor_ID)).

rpa(junior_data_manager,insert,ward(Ward_ID,Type,Ward_Capacity)).
rpa(junior_data_manager,insert,room(Room_ID,Ward_ID,Type,bed_capacity)).
rpa(junior_data_manager,insert,bed(Bed_ID,Room_ID,Type)).
rpa(junior_data_manager,insert,patient(Patient_ID,Last_Name,
First_Name,Address,DOB,Bed_ID)).
rpa(junior_data_manager,insert,diagnosis(Diagnoses_code,Illness_name,Usual_Symptoms)).
rpa(junior_data_manager,insert,ae_consultation(Cons_Number,
Cons_Date,Cons_Description,Patient_ID,Doctor_ID)).
rpa(junior_data_manager,insert,patient_diagnosis(Patient_
Diagnosis_Number,Diagnosing_Doctor,Diagnosis_Desc,Cons_Number,Diagnosis_Code)).

rpa(receptionist,select,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).

rpa(manager,update,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).
rpa(manager,insert,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID)).


rpa(junior_data_manager,insert,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,delete,password(Username,Password)).



rpa(senior_data_manager,select,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,update,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,delete,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,create,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,drop,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,grant,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,references,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,index,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,alter,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,create_view,ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,show_view,ward(Ward_ID,Type,Ward_Capacity)).
```

```
rpa(senior_data_manager,select,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,update,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,delete,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,create,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,drop,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,grant,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,references,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,index,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,alter,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,create_view,room(Room_ID,Ward_ID,Type,Bed_Capacity)).
rpa(senior_data_manager,show_view,room(Room_ID,Ward_ID,Type,Bed_Capacity)).

rpa(senior_data_manager,select,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,update,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,delete,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,create,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,drop,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,grant,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,references,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,index,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,alter,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,create_view,bed(Bed_ID,Room_ID,Type)).
rpa(senior_data_manager,show_view,bed(Bed_ID,Room_ID,Type)).

rpa(senior_data_manager,select,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,update,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,delete,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,create,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,drop,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,grant,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,references,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,index,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,alter,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,create_view,patient(Patient_ID,Last_Name,First_Name,Address,DOB,
Bed_ID)).
rpa(senior_data_manager,show_view,patient(Patient_ID,Last_Name,First_Name,
Address,DOB,Bed_ID)).

rpa(senior_data_manager,select,diagnoses).
rpa(senior_data_manager,update,diagnoses).
rpa(senior_data_manager,delete,diagnoses).
rpa(senior_data_manager,create,diagnoses).
rpa(senior_data_manager,drop,diagnoses).
rpa(senior_data_manager,grant,diagnoses).
rpa(senior_data_manager,references,diagnoses).
rpa(senior_data_manager,index,diagnoses).
rpa(senior_data_manager,alter,diagnoses).
rpa(senior_data_manager,create_view,diagnoses).
rpa(senior_data_manager,show_view,diagnoses).

rpa(senior_data_manager,select,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,update,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,delete,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,create,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,drop,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,grant,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,references,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,?Patient_ID,Doctor_ID)).
```

```
rpa(senior_data_manager,index,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,alter,ae_consultation(Cons_Number,Cons_Date,Cons_Description,
Patient_ID,?Doctor_ID)).
rpa(senior_data_manager,create_view,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,?Patient_ID,Doctor_ID)).
rpa(senior_data_manager,show_view,ae_consultation(Cons_Number,Cons_Date,
Cons_Description,?Patient_ID,Doctor_ID)).

rpa(senior_data_manager,select,patient_diagnoses).
rpa(senior_data_manager,update,patient_diagnoses).
rpa(senior_data_manager,delete,patient_diagnoses).
rpa(senior_data_manager,create,patient_diagnoses).
rpa(senior_data_manager,drop,patient_diagnoses).
rpa(senior_data_manager,grant,patient_diagnoses).
rpa(senior_data_manager,references,patient_diagnoses).
rpa(senior_data_manager,index,patient_diagnoses).
rpa(senior_data_manager,alter,patient_diagnoses).
rpa(senior_data_manager,create_view,patient_diagnoses).
rpa(senior_data_manager,show_view,patient_diagnoses).

rpa(senior_data_manager,select,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,insert,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,update,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,delete,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,create,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,drop,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,grant,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,references,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,index,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,alter,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,create_view,nurse_ward(Ward_ID,Type,Ward_Capacity)).
rpa(senior_data_manager,show_view,nurse_ward(Ward_ID,Type,Ward_Capacity)).


% access on rbac-related data: reserved for senior data manager
rpa(senior_data_manager,select,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,update,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,delete,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,create,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,drop,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,grant,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,references,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,index,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,alter,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,create_view,user(Username,LastName,FirstName,Address,DOB)).
rpa(senior_data_manager,show_view,user(Username,LastName,FirstName,Address,DOB)).

rpa(senior_data_manager,select,password(Username,Password)).
rpa(senior_data_manager,insert,password(Username,Password)).
rpa(senior_data_manager,update,password(Username,Password)).
rpa(senior_data_manager,delete,password(Username,Password)).
rpa(senior_data_manager,create,password(Username,Password)).
rpa(senior_data_manager,drop,password(Username,Password)).
rpa(senior_data_manager,grant,password(Username,Password)).
rpa(senior_data_manager,references,password(Username,Password)).
rpa(senior_data_manager,index,password(Username,Password)).
rpa(senior_data_manager,alter,password(Username,Password)).
rpa(senior_data_manager,create_view,password(Username,Password)).
rpa(senior_data_manager,show_view,password(Username,Password)).

rpa(senior_data_manager,select,role(Role)).
rpa(senior_data_manager,insert,role(Role)).
rpa(senior_data_manager,update,role(Role)).
rpa(senior_data_manager,delete,role(Role)).
rpa(senior_data_manager,create,role(Role)).
rpa(senior_data_manager,drop,role(Role)).
rpa(senior_data_manager,grant,role(Role)).
rpa(senior_data_manager,references,role(Role)).
rpa(senior_data_manager,index,role(Role)).
rpa(senior_data_manager,alter,role(Role)).
rpa(senior_data_manager,create_view,role(Role)).
rpa(senior_data_manager,show_view,role(Role)).
```

```
rpa(senior_data_manager,select,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,insert,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,update,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,delete,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,create,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,drop,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,grant,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,references,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,index,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,alter,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,create_view,d_s(SeniorRole,JuniorRole)).
rpa(senior_data_manager,show_view,d_s(SeniorRole,JuniorRole)).

rpa(senior_data_manager,select,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,insert,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,update,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,delete,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,create,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,drop,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,grant,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,references,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,index,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,alter,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,create_view,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).
rpa(senior_data_manager,show_view,inherits_rpa_path(SeniorRole,JuniorRole,Permission,
Object)).

rpa(senior_data_manager,select,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,insert,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,update,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,delete,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,create,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,drop,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,grant,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,references,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,index,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,alter,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,create_view,is_a(InnerRole,OuterRole)).
rpa(senior_data_manager,show_view,is_a(InnerRole,OuterRole)).

rpa(senior_data_manager,select,rpa(Role,Permission,Object)).
rpa(senior_data_manager,insert,rpa(Role,Permission,Object)).
rpa(senior_data_manager,update,rpa(Role,Permission,Object)).
rpa(senior_data_manager,delete,rpa(Role,Permission,Object)).
rpa(senior_data_manager,create,rpa(Role,Permission,Object)).
rpa(senior_data_manager,drop,rpa(Role,Permission,Object)).
rpa(senior_data_manager,grant,rpa(Role,Permission,Object)).
rpa(senior_data_manager,references,rpa(Role,Permission,Object)).
rpa(senior_data_manager,index,rpa(Role,Permission,Object)).
rpa(senior_data_manager,alter,rpa(Role,Permission,Object)).
rpa(senior_data_manager,create_view,rpa(Role,Permission,Object)).
rpa(senior_data_manager,show_view,rpa(Role,Permission,Object)).

rpa(senior_data_manager,select,ssd(Role1,Role2)).
rpa(senior_data_manager,insert,ssd(Role1,Role2)).
rpa(senior_data_manager,update,ssd(Role1,Role2)).
rpa(senior_data_manager,delete,ssd(Role1,Role2)).
rpa(senior_data_manager,create,ssd(Role1,Role2)).
rpa(senior_data_manager,drop,ssd(Role1,Role2)).
rpa(senior_data_manager,grant,ssd(Role1,Role2)).
rpa(senior_data_manager,references,ssd(Role1,Role2)).
rpa(senior_data_manager,index,ssd(Role1,Role2)).
rpa(senior_data_manager,alter,ssd(Role1,Role2)).
```

```
rpa(senior_data_manager,create_view,ssd(Role1,Role2)).
rpa(senior_data_manager,show_view,ssd(Role1,Role2)).

rpa(senior_data_manager,select,dsd(Role1,Role2)).
rpa(senior_data_manager,insert,dsd(Role1,Role2)).
rpa(senior_data_manager,update,dsd(Role1,Role2)).
rpa(senior_data_manager,delete,dsd(Role1,Role2)).
rpa(senior_data_manager,create,dsd(Role1,Role2)).
rpa(senior_data_manager,drop,dsd(Role1,Role2)).
rpa(senior_data_manager,grant,dsd(Role1,Role2)).
rpa(senior_data_manager,references,dsd(Role1,Role2)).
rpa(senior_data_manager,index,dsd(Role1,Role2)).
rpa(senior_data_manager,alter,dsd(Role1,Role2)).
rpa(senior_data_manager,create_view,dsd(Role1,Role2)).
rpa(senior_data_manager,show_view,dsd(Role1,Role2)).

rpa(senior_data_manager,select,ura(User,Role)).
rpa(senior_data_manager,insert,ura(User,Role)).
rpa(senior_data_manager,update,ura(User,Role)).
rpa(senior_data_manager,delete,ura(User,Role)).
rpa(senior_data_manager,create,ura(User,Role)).
rpa(senior_data_manager,drop,ura(User,Role)).
rpa(senior_data_manager,grant,ura(User,Role)).
rpa(senior_data_manager,references,ura(User,Role)).
rpa(senior_data_manager,index,ura(User,Role)).
rpa(senior_data_manager,alter,ura(User,Role)).
rpa(senior_data_manager,create_view,ura(User,Role)).
rpa(senior_data_manager,show_view,ura(User,Role)).

rpa(senior_data_manager,select,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,insert,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,update,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,delete,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,create,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,drop,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,grant,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,references,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,index,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,alter,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,create_view,d_rpa(Role,Permission,Object)).
rpa(senior_data_manager,show_view,d_rpa(Role,Permission,Object)).


rpa(Role,Permission,password(Username,Password)) :-
  role(Role),
  (Permission == select; Permission == update).


% ura
% ura(User,Role).
ura(dr_peters,house_officer_day).
ura(dr_davies,house_officer_night).
ura(dr_williams,house_officer_day).
ura(dr_jones,house_officer_night).
ura(dr_fish,house_officer_night).
ura(dr_ghosh,senior_house_officer_day).
ura(dr_edmonds,senior_house_officer_night).
ura(dr_bowie,senior_house_officer_day).
ura(dr_python,specialist_registrar).
ura(dr_kellett,specialist_registrar).
ura(dr_sugar,consultant).
ura(dr_evans,consultant).

ura(miss_strand,student_nurse_day).
ura(miss_strand,student_nurse_night).

ura(mrs_lewis,staff_nurse_day).
ura(mr_davies,staff_nurse_day).

ura(mr_kenning,staff_nurse_night).
ura(mr_minnow,staff_nurse_night).

ura(mr_avery,sister_day).
ura(mrs_jones,sister_day).
```

```
ura(miss_jackson,sister_night).
ura(mrs_jones,sister_night).

ura(miss_jacobson,specialist_nurse).
ura(mr_mctaggart,specialist_nurse).

ura(mr_clarkson,junior_data_manager).
ura(miss_lewis,junior_data_manager).

ura(mrs_canning,senior_data_manager).

ura(miss_darch,receptionist).
ura(mrs_james,manager).

% junior data managers who are also something else
ura(miss_strand,junior_data_manager).
ura(miss_darch,junior_data_manager).


% receptionists who are also something else
ura(dr_peters,receptionist).
ura(dr_evans,receptionist).



% attach nurses to wards (probably needs its own data table)
% nurse_ward(User,Ward).
nurse_ward(miss_strand,ward1).
nurse_ward(miss_strand,ward2).

nurse_ward(mrs_lewis,ward1).
nurse_ward(mr_davies,ward2).

nurse_ward(mr_kenning,ward1).
nurse_ward(mr_minnow,ward1).

nurse_ward(mr_avery,ward1).
nurse_ward(mrs_jones,ward2).

nurse_ward(miss_jackson,ward1).

deactivate(null,null,0).
drpa(null,null,null).
```

Appendix V: Context Constraints in Static RBAC

```
% context_condition(Name,User,Permission,Object).
% associated_cc(Role,Permission,Object,ContextConstraint).


% context conditions
% context_condition(Name,User,Permission,Object).
context_condition(nurse_in_same_ward_as_patient,Nurse,P,patient(Patient_ID,Last_Name,
First_Name,Address,DOB,Bed_ID)) :-
  bed(Bed_ID,Room_ID,_),
  room(Room_ID,Ward_ID,_,_),
  nurse_ward(Nurse,Ward_ID).

context_condition(staff_nurse_or_sister_active_for_2_hours,U,P,O):-
  active(Nurse,R1,D),
  nurse_ward(Nurse,W),
  nurse_ward(U,W),
  date_time_stamp(D,T),
  get_time(Stamp),
  Stamp - T >= 60*60*2, % number of seconds representing 2 hours
  included_in(R1,R2),
  (R2 = sister; R2 = staff_nurse).

context_condition(patient_treated_by_doctor,Doctor_ID,P,patient(Patient_ID,Last_Name,
First_Name,Address,DOB,Bed_ID)) :-
  ae_consultation(_,_,_,Patient_ID,Doctor_ID).

context_condition(patient_treated_by_doctor,Doctor_ID,P,patient(Patient_ID,Last_Name,
First_Name,Address,DOB,Bed_ID)) :-
  ae_consultation(Cons_Number,_,_,Patient_ID,_),
  patient_diagnosis(_,Doctor_ID,_,Cons_Number,_).

% temporal conditions
context_condition(day_duty,U,P,O) :-
  get_time(Stamp),
  stamp_date_time(Stamp,D,local),
  D = date(_, _, _, H, _, _, _, _, _),
  H >= 9, H < 21.

context_condition(night_duty,U,P,O) :-
  get_time(Stamp),
  stamp_date_time(Stamp,D,local),
  D = date(_, _, _, H, _, _, _, _, _),
  (H >= 21; H < 9).

context_condition(office_hours,U,P,O) :-
  get_time(Stamp),
  stamp_date_time(Stamp,D,local),
  D = date(_, _, _, H, _, _, _, _, _),
  H >= 9, H < 17,
  format_time(codes(Day), '%a', D),
  (Day = "Mon"; Day = "Tue"; Day = "Wed"; Day = "Thu"; Day = "Fri").



context_condition(user_is_same_as_logged_in,U,P,password(Username,Password)) :-
  password(Username,Password),
  U = Username.



% association of context constraints with roles and permissions
% associated_cc(Role,Permission,Object,ContextConstraint).

associated_cc(sister,_,patient(Patient_ID,Last_Name,First_Name,Address,DOB,Bed_ID),
nurse_in_same_ward_as_patient).
associated_cc(senior_house_officer,_,patient(Patient_ID,Last_Name,First_Name,Address,
DOB,Bed_ID),patient_treated_by_doctor).

associated_cc(student_nurse,_,_,staff_nurse_or_sister_currently_active_for_2_hours).

associated_cc(junior_data_manager,_,_,office_hours).
```

```
associated_cc(receptionist,_,_,office_hours).

associated_cc(day_duty,_,_,day_duty).
associated_cc(night_duty,_,_,night_duty).

associated_cc(Role,_,password(Username,Password),user_is_same_as_logged_in) :-
  role(Role),
  Role \== senior_data_manager.


% ssd(Role1,Role2).
% dsd(Role1,Role2).


% static separation of duties
% ssd(Role1,Role2).
% a senior_data_manager cannot be anything else
ssd(senior_data_manager,X):-
  role(X),
  X \== senior_data_manager.


% dynamic separation of duties
% dsd(Role1,Role2).
dsd(consultant,receptionist).
% a junior data manager cannot be simultaneously logged in as anything else
dsd(junior_data_manager,X):-
  role(X),
  X \== junior_data_manager.
```

Appendix VI: RBAC and database diagrams

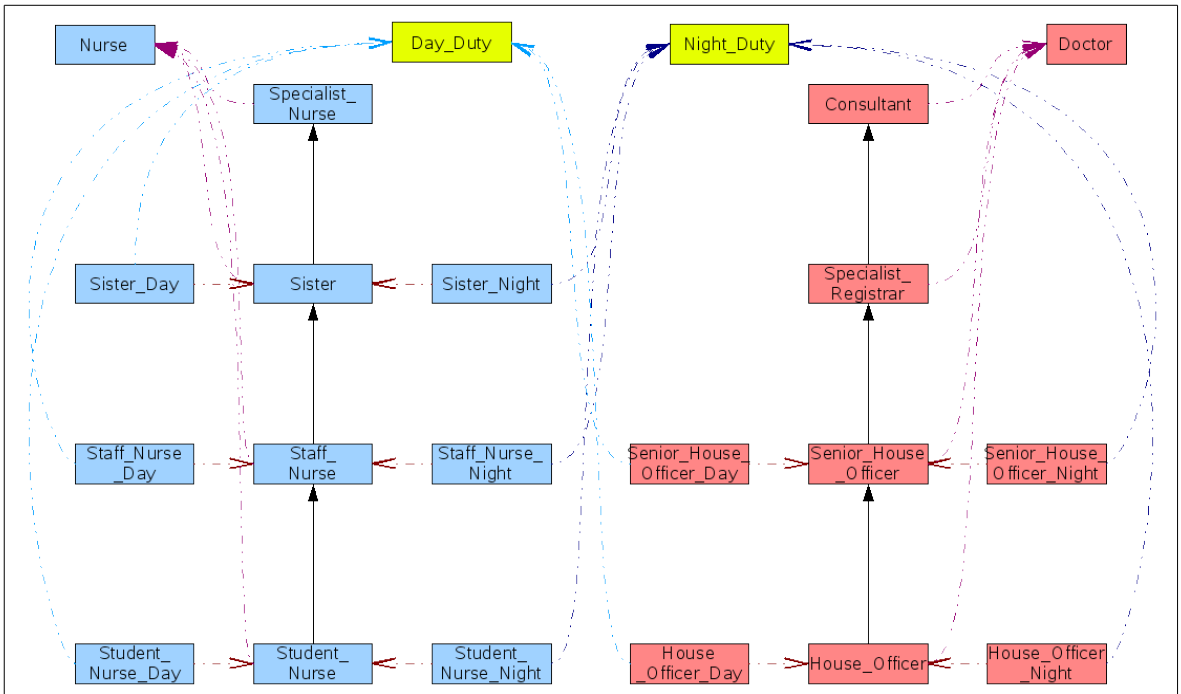

Figure 115: Role Inclusion in Hospital Database. Solid lines represent `d_s` relationships; dotted lines represent `is_a` relationships.

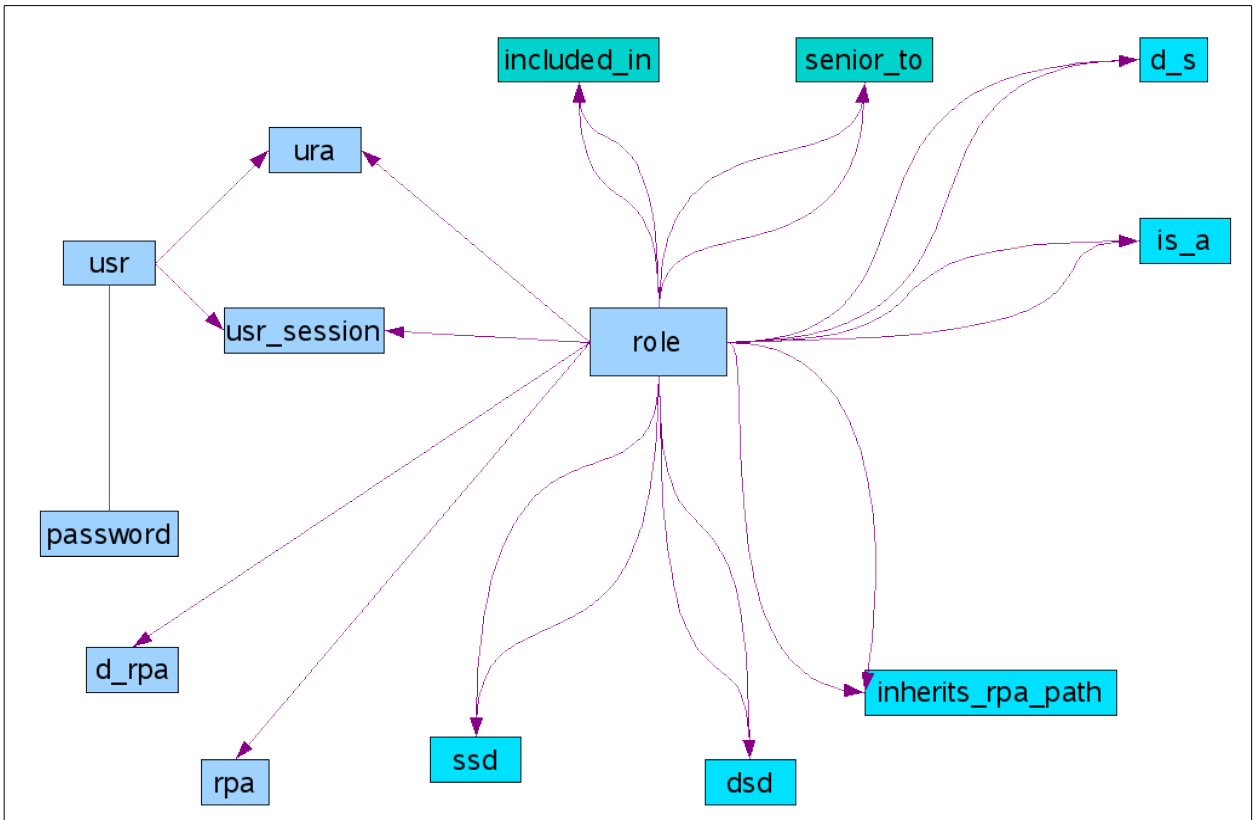

Figure 116: ERD of RBAC schema: tables only. Blue boxes are tables. Cyan boxes are tables linking pairs of roles. Jade boxes are tables populated by triggers to form the results of recursive rules.

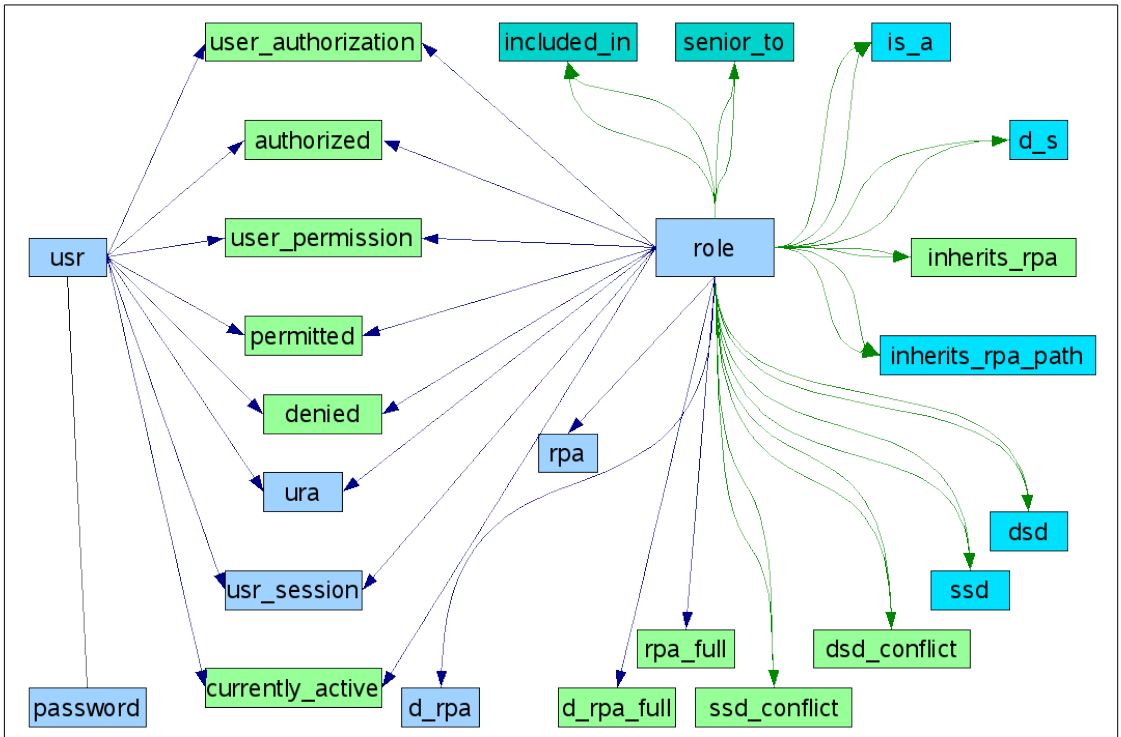

Figure 117: ERD of RBAC data: tables and views. Blue boxes are tables. Cyan boxes are tables linking pairs of roles. Jade boxes are tables populated by triggers to form the results of recursive rules. Green boxes are views. An arrow represents the 'many' end of a 1:many relationship. Double-relationships, where an object has two relationships with another object, are in green; the rest are in blue.

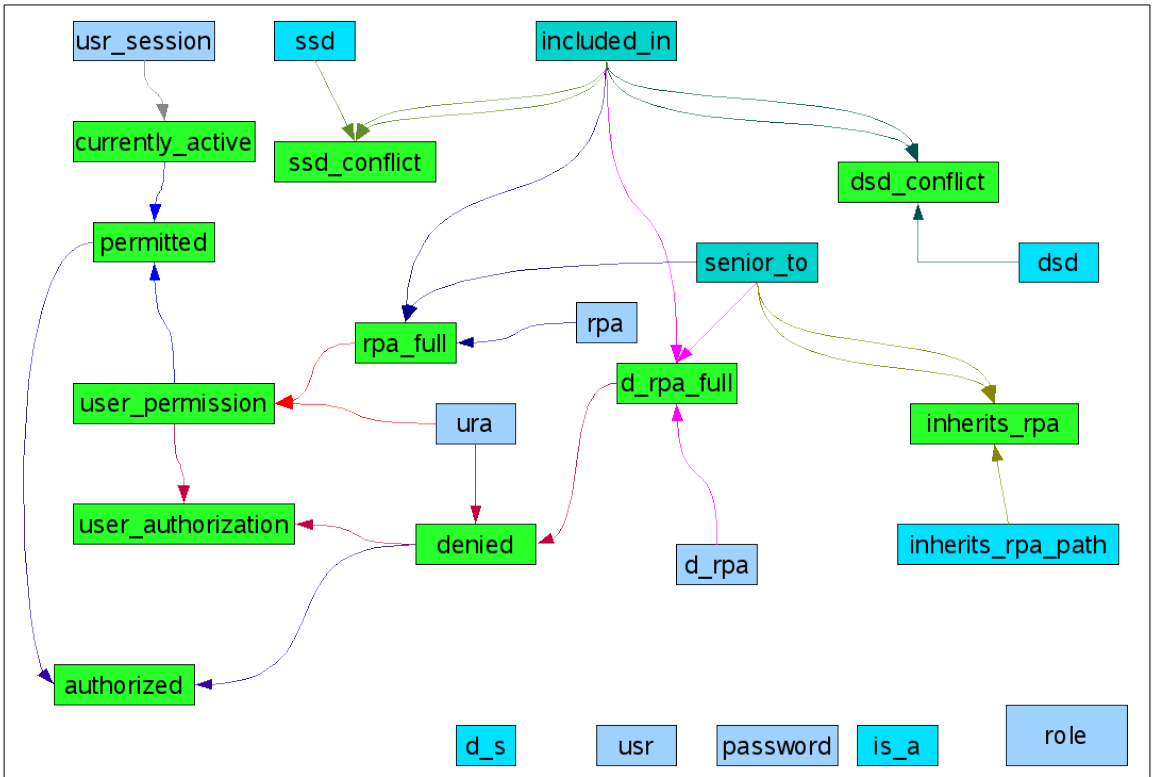

Figure 118: Formation of views from constituent objects, as determined by ***CREATE VIEW*** *statements. Arrows point to view formed. All arrows representing objects forming a specific view have the same colour. Some tables do not participate in any* ***CREATE VIEW*** *statements.*

Appendix VII: Oracle Database: Data Description

Table 28: Roles and permissions in Hospital database

Role	***Directly Senior to (Inherits from)***	***Permissions***
House_Officer		*Read*: Ward, Room, Bed, Patient, Diagnosis, User, AE_Consultation, Patient_Diagnosis
Senior_House_Officer	*House_Officer*	*Update*: Diagnosis, AE_Consultation, Patient_Diagnosis
Specialist_Registrar	*Senior_House_Officer*	*Insert*: Patient_Diagnosis
Consultant	*Specialist_Registrar*	*Insert*: AE_Consultation
Student_Nurse		*Read*: Ward, Room, Bed, Patient, User
Staff_Nurse	*Student_Nurse*	*Update*: Patient *Read* Diagnosis, User, AE_Consultation, Patient_Diagnosis
Sister	*Staff_Nurse*	*Update*: Patient_Diagnosis
Specialist_Nurse	*Sister*	*Update*: AE_Consultation
Junior_Data_Manager		*Insert*: Ward, Room, Bed, Patient, Diagnosis, AE_Consultation, Patient_Diagnosis
Senior_Data_Manager	*Junior_Data_Manager*	Complete access to entire database
Receptionist		*Read*: Patient
Manager	*Receptionist*	*Update*: Patient *Insert*: Patient

```
usr( user_id , last_name, first_name, address, date_of_birth )
password( user_id, password )
role( role )
d_s( senior_role, junior_role )
inherits_rpa_path( senior_role, junior_role, action, object )
is_a( inner_role, outer_role )
ura( usr, role )
rpa( role, action, object )
d_rpa( role, action, object )
usr_session( usr, role, start_time, end_time )
dsd( role1, role2 )
ssd( role1, role2 )
senior_to( senior_role, junior_role )
included_in( inner_role, outer_role )
-- temporary tables
senior_to_staging( senior_role, junior_role )
included_in_staging( inner_role, outer_role )
```

Text 57: Schema for RBAC model, listing tables.

```
inherits_rpa ( senior_role, junior_role, action, object )
rpa_full ( role, action, object, senior_role, junior_role )
permittable ( usr, object, action, role )
currently_active ( usr, role, start_time )
permitted ( usr, object, action, role )
d_rpa_full ( role, action, object, senior_role, junior_role )
denied ( usr, action, object, role )
authorizable ( usr, object, action, role )
authorized ( usr, object, action, role )
dsd_conflict ( role1, role2 )
ssd_conflict ( role1, role2 )
```

Text 58: Schema for RBAC model, listing views.

Table 29: Triggers for modelling static RBAC

Trigger Condition	***Action***
AFTER INSERT ON role	Apply `included_in(R1,R1)` by inserting the appropriate entry in the **included_in** table.
AFTER INSERT ON d_s	Populate **senior_to** with any new values to be added as a result of new entry in **d_s**.
AFTER UPDATE ON d_s	**DELETE FROM senior_to**, and repopulate it.
AFTER DELETE ON d_s	**DELETE FROM senior_to**, and repopulate it.
AFTER INSERT ON is_a	Populate **included_in** with any new values to be added as a result of new entry in **is_a**.
AFTER UPDATE ON is_a	**DELETE FROM included_in**, and repopulate it.
AFTER DELETE ON is_a	**DELETE FROM included_in**, and repopulate it.
BEFORE INSERT ON ura	Prevent insertion if new entry would cause SSD conflict.
BEFORE UPDATE ON ura	Prevent update if it would cause SSD conflict.
BEFORE INSERT ON session	Prevent insertion if new entry would cause DSD conflict.
BEFORE UPDATE ON session	Prevent any such update except one that ends a session (i.e. set **end_time** to the current time)
BEFORE INSERT ON password	Encrypt password (in some implementations). The password is stored in the **password** table in encrypted form.

Table 30: Triggers for RBAC enforcement mechanism

Trigger Condition	*Function(s) Called (if any)*	*Action*
`AFTER INSERT OR UPDATE ON role`	`add_role`	`CREATE :new.role`
`AFTER UPDATE OR DELETE ON role`	`drop_role`	`DROP :old.role`
`BEFORE UPDATE ON role`		Prevent operation
`AFTER INSERT OR UPDATE ON d_s`	`grant_role`	`GRANT :new.senior_role TO :new.junior_role`
`AFTER UPDATE OR DELETE ON d_s`	`revoke_role`	`REVOKE :old.senior_role FROM :old.junior_role`
`AFTER INSERT OR UPDATE ON is_a`	`grant_role`	`GRANT new.outer_role TO new.inner_role`
`AFTER UPDATE OR DELETE ON is_a`	`revoke_role`	`REVOKE old.outer_role FROM old.inner_role`
`AFTER INSERT ON rpa`	`grant_priv`	`GRANT privilege ON object TO role`
`AFTER DELETE ON rpa`	`revoke_priv`	`REVOKE privilege ON object FROM role`
`BEFORE UPDATE ON rpa`		Prevent this operation
`AFTER DELETE ON ura`	`revoke_role`	`REVOKE` ***`role`*** `FROM` ***`user`*** if `user` is currently active in it (otherwise, it would not be assigned)
`AFTER INSERT ON ura`	`grant_role`	`GRANT` ***`role`*** `TO` ***`user`*** if `user` is active in role (this shouldn't happen)
`BEFORE INSERT ON ura`		Prevent insertion if SSD conflict exists
`BEFORE UPDATE ON ura`		Prevent this operation
`BEFORE UPDATE ON usr`		Prevent operation
`BEFORE UPDATE ON usr`		Prevent operation
`BEFORE UPDATE ON password`		Prevent operation if attempting to modify `user_id`
`AFTER INSERT ON password`	`create_user`	`CREATE new.user`
`AFTER DELETE ON password`	`drop_user`	`DROP old.user`
`AFTER INSERT ON usr_session`	`grant_role`	`GRANT role TO user`
`AFTER UPDATE ON usr_session`	`revoke_role`	`REVOKE role FROM user`
`BEFORE INSERT ON usr_session`		1. Query `currently_active` to determine whether an active session with this user and role already exists. 2. Check for DSD violations (user active in another role that conflicts with this role). 3. Check for user not assigned to role in `ura`. If any of these are true, then prevent insertion.
`BEFORE UPDATE ON usr_session`		Prevent operation unless it is to deactivate session by modifying `end_time`
`BEFORE DELETE ON usr_session`		Prevent operation

Table 31: Number of unique **rpa_full** *rows by role*

Role	*Unique rpa_full rows*	*Users Assigned*	*permittable*
day_duty	0	0	0
night_duty	0	0	0
nurse	0	0	0
student_nurse	5	0	0
student_nurse_d	5	1	5
student_nurse_n	5	1	5
staff_nurse	9	0	0
staff_nurse_d	9	2	18
staff_nurse_n	9	2	18
sister	10	0	0
sister_d	10	2	20
sister_n	10	2	20
specialist_nurse	13	2	26
doctor	0	0	0
house_officer	8	0	0
house_officer_d	8	2	16
house_officer_n	8	3	24
snr_house_officer	11	0	0
snr_house_officer_d	11	2	22
snr_house_officer_n	11	1	11
specialist_registrar	12	2	24
consultant	13	2	26
administrator	0	0	0
receptionist	1	3	3
manager	8	1	8
data_manager	0	0	0
jnr_data_manager	7	(not tested)	(not tested)
snr_data_manager	153	(not tested)	(not tested)

Appendix VIII: SQL Code for Static RBAC

Tables

```
CREATE TABLE usr (
  user_id VARCHAR(10),
  last_name  VARCHAR(50),
  first_name VARCHAR(50),
  address   VARCHAR(50),
  date_of_birth       DATE,
    Primary Key (user_id)
);

CREATE TABLE password (
  user_id VARCHAR(10),
  password VARCHAR(41),
    Primary Key (user_id),
    FOREIGN KEY (user_id) REFERENCES usr (user_id)
);


CREATE TABLE role (
  role VARCHAR(64),
    Primary Key (role)
);

-- dummy role mimicking anonymous variable _ in Prolog
INSERT INTO role( role ) VALUES ( '_' );

-- direct seniority
CREATE TABLE d_s (
  senior_role  VARCHAR(64) NOT NULL,
  junior_role  VARCHAR(64) NOT NULL,
    Primary Key (senior_role,junior_role),
    FOREIGN KEY (senior_role) REFERENCES role(role),
    FOREIGN KEY (junior_role) REFERENCES role(role)
);

-- rpa path inheritance
CREATE TABLE inherits_rpa_path (
  senior_role  VARCHAR(64) NOT NULL,
  junior_role  VARCHAR(64) NOT NULL,
  action VARCHAR(64),
  object VARCHAR(64),
    Primary Key (senior_role,junior_role,action,object),
    FOREIGN KEY (senior_role) REFERENCES role(role),
    FOREIGN KEY (junior_role) REFERENCES role(role),
    CONSTRAINT check_action_inherits_rpa_path
    CHECK ( action IN ( 'select', 'insert', 'update', 'delete', 'alter', '_' ) )
);

CREATE TABLE is_a (
  inner_role VARCHAR(64) NOT NULL,
  outer_role VARCHAR(64) NOT NULL,
    Primary Key (inner_role,outer_role),
    FOREIGN KEY (inner_role) REFERENCES role(role),
    FOREIGN KEY (outer_role) REFERENCES role(role)
);

CREATE TABLE ura(
  usr VARCHAR(16) NOT NULL,
  role VARCHAR(64) NOT NULL,
    PRIMARY KEY (usr, role),
    FOREIGN KEY (usr) REFERENCES usr(user_id),
    FOREIGN KEY (role) REFERENCES role(role)
);

CREATE TABLE rpa(
  role VARCHAR(64) NOT NULL,
```

```
  action VARCHAR(16) NOT NULL,
  object VARCHAR(64) NOT NULL,
    PRIMARY KEY (role, object, action),
    FOREIGN KEY (role) REFERENCES role(role),
    CONSTRAINT check_action_rpa
    CHECK ( action IN ( 'select', 'insert', 'update', 'delete', 'alter' ) )
);


CREATE TABLE d_rpa(
  role VARCHAR(64) NOT NULL,
  action VARCHAR(16) NOT NULL,
  object VARCHAR(64) NOT NULL,
    PRIMARY KEY (role, object, action),
    FOREIGN KEY (role) REFERENCES role(role),
    CONSTRAINT check_action_d_rpa
    CHECK ( action IN ( 'select', 'insert', 'update', 'delete', 'alter' ) )
);


CREATE TABLE usr_session(
  usr VARCHAR(16) NOT NULL,
  role VARCHAR(64) NOT NULL,
  start_time TIMESTAMP NOT NULL,
  end_time TIMESTAMP,
    FOREIGN KEY (usr) REFERENCES usr(user_id),
    FOREIGN KEY (role) REFERENCES role(role)
);
-- dynamic separation of duties
CREATE TABLE dsd(
  role1 VARCHAR(64) NOT NULL,
  role2 VARCHAR(64) NOT NULL,
    PRIMARY KEY (role1, role2),
    FOREIGN KEY (role1) REFERENCES role(role),
    FOREIGN KEY (role2) REFERENCES role(role)
);
-- static separation of duties
CREATE TABLE ssd(
  role1 VARCHAR(64) NOT NULL,
  role2 VARCHAR(64) NOT NULL,
    PRIMARY KEY (role1, role2),
    FOREIGN KEY (role1) REFERENCES role(role),
    FOREIGN KEY (role2) REFERENCES role(role)
);

-- the following tables can only be updated by triggers
-- senior role
CREATE TABLE senior_to(
  senior_role  VARCHAR(64) NOT NULL,
  junior_role  VARCHAR(64) NOT NULL,
--    Primary Key (senior_role,junior_role),
    FOREIGN KEY (senior_role) REFERENCES role(role),
    FOREIGN KEY (junior_role) REFERENCES role(role)
);


CREATE TABLE included_in (
  inner_role VARCHAR(64) NOT NULL,
  outer_role VARCHAR(64) NOT NULL,
    Primary Key (inner_role,outer_role),
    FOREIGN KEY (inner_role) REFERENCES role(role),
    FOREIGN KEY (outer_role) REFERENCES role(role)
);


CREATE TABLE log (
  txt VARCHAR(512)
);


-- temporary tables
```

```
CREATE TABLE senior_to_staging(
  senior_role  VARCHAR(64) NOT NULL,
  junior_role  VARCHAR(64) NOT NULL
);

CREATE TABLE included_in_staging(
  inner_role  VARCHAR(64) NOT NULL,
  outer_role  VARCHAR(64) NOT NULL
);
```

Views

```
CREATE VIEW inherits_rpa AS
-- inherits_rpa(R2,R3,P,O) :- senior_to(R1,R2),
--                            senior_to(R3,R4),
--                            inherits_rpa_path(R1,R4,P,O).
  SELECT DISTINCT s1.junior_role AS senior_role, s2.senior_role AS junior_role, action, object
    FROM senior_to s1, senior_to s2, inherits_rpa_path
    WHERE s1.senior_role = inherits_rpa_path.senior_role
    AND s2.junior_role = inherits_rpa_path.junior_role;


CREATE VIEW rpa_full AS -- all permissions to all roles, both explicit and implicit (by
inheritance)
-- rpa_full(R1,P,O) :- included_in(R1,R2),
--                     senior_to(R2,R3),
--                    rpa(R3,P,O),
--                    inherits_rpa(R2,R3,P,O).
  SELECT DISTINCT included_in.inner_role AS role, action, object, senior_role, junior_role
FROM rpa, included_in, senior_to
    WHERE included_in.outer_role = senior_to.senior_role
    AND senior_to.junior_role = rpa.role
    AND (
      (senior_to.senior_role,senior_to.junior_role) IN
        (SELECT senior_role,junior_role FROM inherits_rpa WHERE action = '_' AND object = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,action) IN
        (SELECT senior_role,junior_role,action FROM inherits_rpa WHERE object = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,object) IN
        (SELECT senior_role,junior_role,object FROM inherits_rpa WHERE action = '_')
      OR
      (senior_to.senior_role,senior_to.junior_role,action,object) IN
        (SELECT senior_role,junior_role,action,object FROM inherits_rpa)
    )
;

CREATE VIEW permittable AS
-- permittable(U,P,O) :- ura(U,R),
--                       permittable(U,P,O,R).
-- permittable(U,P,O,R) :- rpa_full(R,P,O).
  SELECT DISTINCT usr, object, action, ura.role AS role FROM ura, rpa_full
    WHERE ura.role = rpa_full.role;

CREATE VIEW currently_active AS
-- currently_active(U,R1,D1)
-- This behaves differently from its equivalent in Prolog,
-- hence the full predicate is not shown.
-- Unlike in Prolog, it does not test for DSD inconsistency,
-- since this is already done when inserting a row in active.
-- active is equivalent to both activate and deactivate in Prolog.
  SELECT DISTINCT usr, role, start_time FROM usr_session
  WHERE usr_session.start_time < SYSTIMESTAMP
  AND (usr_session.end_time > SYSTIMESTAMP or usr_session.end_time is null);

CREATE VIEW permitted AS
-- permitted(U,P,O) :-    ura(U,R),
--                        permitted(U,P,O,R).
-- permitted(U,P,O,R1) :- included_in(R0,R1),
--                        currently_active(U,R0,_),
--                        not(fail_context_constraint(U,R0,P,O)),
```

```
--                        permittable(U,P,O,R1).
  SELECT DISTINCT permittable.usr AS usr, object, action, permittable.role AS role FROM
permittable, currently_active
    WHERE permittable.usr = currently_active.usr and permittable.role = currently_active.role;

CREATE VIEW d_rpa_full AS
-- d_rpa_full(R1,P,O) :- included_in(R3,R2),
--                       senior_to(R1,R2),
--                       d_rpa(R1,P,O).
  SELECT DISTINCT included_in.inner_role AS role, action, object, senior_role, junior_role
FROM d_rpa, included_in, senior_to
    WHERE included_in.outer_role = senior_to.junior_role
    AND d_rpa.role = senior_to.senior_role;

CREATE VIEW denied AS
-- denied(U,P,O) :-    ura(U,R),
--                     denied(U,P,O,R).
--
-- denied(U,P,O,R) :- d_rpa_full(R,P,O).
  SELECT DISTINCT usr, action, object, ura.role AS role FROM ura, d_rpa_full
    WHERE ura.role = d_rpa_full.role;

CREATE VIEW authorizable AS
-- usr_authorization(U,P,O) :-   ura(U,R),
--                               usr_authorization(U,P,O,R).
-- usr_authorization(U,P,O,R) :- usr_permission(U,P,O,R),
--                                not(denied(U,P,O,R)).
  SELECT DISTINCT usr, object, action, role FROM permittable
  WHERE (usr, object, action, role) NOT IN
      (SELECT usr, object, action, role FROM denied);

CREATE VIEW authorized AS
-- authorized(U,P,O) :-   ura(U,R),
--                        authorized(U,P,O,R).
-- authorized(U,P,O,R) :- permitted(U,P,O,R),
--                        not(denied(R,P,O)).
  SELECT DISTINCT usr, object, action, role FROM permitted
  WHERE (usr, object, action, role) NOT IN
      (SELECT usr, object, action, role FROM denied);

CREATE VIEW dsd_conflict AS
-- dsd_conflict(R1,R3) :- included_in(R1,R2),
--                        included_in(R3,R4),
--                        dsd(R2,R4), !.
--                        dsd(R4,R2).
-- Entries in dsd with '_' are expanded to all roles.
  SELECT i1.inner_role AS role1, i2.inner_role AS role2 FROM dsd, included_in i1, included_in
i2
    WHERE i1.outer_role = dsd.role1 AND i2.outer_role = dsd.role2
    UNION
  SELECT i2.inner_role AS role1,i1.inner_role AS role2 FROM dsd, included_in i1, included_in
i2
    WHERE i1.outer_role = dsd.role1 AND i2.outer_role = dsd.role2
    UNION
  SELECT i1.inner_role AS role1,role AS role2 FROM dsd, role, included_in i1
    WHERE i1.outer_role = dsd.role1 AND dsd.role1 <> role.role AND dsd.role2 = '_'
    UNION
  SELECT role AS role1,i1.inner_role AS role2 FROM dsd, role, included_in i1
    WHERE i1.outer_role = dsd.role1 AND dsd.role1 <> role.role AND dsd.role2 = '_'
    UNION
  SELECT i2.inner_role AS role1, role AS role2 FROM dsd, role, included_in i2
    WHERE i2.outer_role = dsd.role2 AND dsd.role2 <> role.role AND dsd.role1 = '_'
    UNION
  SELECT role AS role1, i2.inner_role AS role2 FROM dsd, role, included_in i2
    WHERE i2.outer_role = dsd.role2 AND dsd.role2 <> role.role AND dsd.role1 = '_'
;

CREATE VIEW ssd_conflict AS
-- ssd_conflict(R1,R3) :- included_in(R1,R2),
--                        included_in(R3,R4),
--                        ssd(R2,R4), !.
```

```
--                        ssd(R4,R2).
-- Entries in ssd with '_' are expanded to all roles.
  SELECT i1.inner_role AS role1, i2.inner_role AS role2 FROM ssd, included_in i1, included_in
i2
    WHERE i1.outer_role = ssd.role1 AND i2.outer_role = ssd.role2
    UNION
  SELECT i2.inner_role AS role1,i1.inner_role AS role2 FROM ssd, included_in i1, included_in
i2
    WHERE i1.outer_role = ssd.role1 AND i2.outer_role = ssd.role2
    UNION
  SELECT i1.inner_role AS role1,role AS role2 FROM ssd, role, included_in i1
    WHERE i1.outer_role = ssd.role1 AND ssd.role1 <> role.role AND ssd.role2 = '_'
    UNION
  SELECT role AS role1,i1.inner_role AS role2 FROM ssd, role, included_in i1
    WHERE i1.outer_role = ssd.role1 AND ssd.role1 <> role.role AND ssd.role2 = '_'
    UNION
  SELECT i2.inner_role AS role1, role AS role2 FROM ssd, role, included_in i2
    WHERE i2.outer_role = ssd.role2 AND ssd.role2 <> role.role AND ssd.role1 = '_'
    UNION
  SELECT role AS role1, i2.inner_role AS role2 FROM ssd, role, included_in i2
    WHERE i2.outer_role = ssd.role2 AND ssd.role2 <> role.role AND ssd.role1 = '_'
;
```

Triggers

```
CREATE OR REPLACE TRIGGER role_after_all
AFTER INSERT OR UPDATE OR DELETE ON role
FOR EACH ROW
BEGIN

     -- Update also needs to modify d_s tables etc. CASCADE UPDATE?

     IF UPDATING OR DELETING THEN
         DELETE FROM included_in
             WHERE inner_role = :old.role
             AND outer_role = :old.role;
        drop_role(full_db_user(:old.role));
     END IF;



     IF UPDATING OR INSERTING THEN
     -- Insert an entry for the new role in included_in,
--       corresponding to the rule
--       included_in(R1,R1).
         INSERT INTO included_in(inner_role,outer_role)
         VALUES (:new.role,:new.role);
         create_role(full_db_user(:new.role));
    END IF;


END;


/



CREATE OR REPLACE TRIGGER role_before_update
BEFORE UPDATE ON role
FOR EACH ROW
BEGIN
    RAISE_APPLICATION_ERROR(-20000, 'You cannot change the name of a role once created.');
END;


/


CREATE OR REPLACE TRIGGER d_s_after_all
AFTER INSERT OR UPDATE OR DELETE ON d_s
```

```
FOR EACH ROW
BEGIN
    IF (DELETING OR UPDATING) THEN
        revoke_role(full_db_user(:old.junior_role),full_db_user(:old.senior_role));
    END IF;

    IF (UPDATING OR INSERTING) THEN
        grant_role(full_db_user(:new.junior_role),full_db_user(:new.senior_role));
    insert into log values ( 'GRANT ' || full_db_user(:new.junior_role)
         || ' TO ' || full_db_user(:new.senior_role) );
    END IF;
END;


/


CREATE OR REPLACE TRIGGER d_s_after_all_sl
AFTER INSERT OR UPDATE OR DELETE ON d_s
BEGIN
    IF (DELETING OR UPDATING) THEN
      delete_senior_to();
    END IF;

    insert_senior_to();
END;


/

CREATE OR REPLACE TRIGGER senior_to_after_insert_sl
AFTER INSERT ON senior_to
BEGIN

    recourse_senior_to();
END;


/


CREATE OR REPLACE TRIGGER is_a_after_all
AFTER INSERT OR UPDATE OR DELETE ON is_a
FOR EACH ROW
BEGIN
   IF (DELETING OR UPDATING) THEN
       revoke_role(full_db_user(:old.outer_role),full_db_user(:old.inner_role));
    END IF;

    IF (UPDATING OR INSERTING) THEN
       grant_role(full_db_user(:new.outer_role),full_db_user(:new.inner_role));
    END IF;

END;

/


CREATE OR REPLACE TRIGGER is_a_after_all_sl
AFTER INSERT OR UPDATE OR DELETE ON is_a
BEGIN
    insert_included_in();
END;


/


CREATE OR REPLACE TRIGGER included_in_after_insert_sl
AFTER INSERT ON included_in
BEGIN
```

```
    recourse_included_in();
END;

/

CREATE OR REPLACE TRIGGER rpa_after_insert
AFTER INSERT ON rpa
FOR EACH ROW
BEGIN
    grant_priv(:new.action, quote_ident(:new.object), full_db_user(:new.role) );
END;


/

CREATE OR REPLACE TRIGGER rpa_after_delete
AFTER DELETE ON rpa
FOR EACH ROW
BEGIN
    revoke_priv(:old.action, quote_ident(:old.object), full_db_user(:old.role) );
END;


/

CREATE OR REPLACE TRIGGER rpa_before_update
BEFORE UPDATE ON rpa
FOR EACH ROW
BEGIN

    raise_application_error(-20000, 'You cannot alter an existing user-role assignment.');

END;


/

CREATE OR REPLACE TRIGGER ura_before_insert
BEFORE INSERT ON ura
FOR EACH ROW
DECLARE
  v_conflicting_roles INTEGER;
BEGIN
    -- Check for static separation of duty
    -- rules that conflict with this attempt to
    -- insert a user-role assignment.
    -- If any are found, then the ura cannot be inserted.

    -- Implements rule
    -- inconsistent_ssd(U,R1) :- ura(U,R1),
    --                           ssd(R1,R2),
    --                           ura(U,R2).



    SELECT count(*) INTO v_conflicting_roles FROM ssd_conflict, ura
        WHERE ssd_conflict.role1 = ura.role
        AND ssd_conflict.role2 = :new.role
        AND ura.usr = :new.usr;

    IF v_conflicting_roles > 0 THEN
        RAISE_APPLICATION_ERROR(-20000, 'Conflicting roles: cannot assign ' || :new.usr || '
to ' || :new.role || '.');
    END IF;

END;

/
```

```
CREATE OR REPLACE TRIGGER ura_before_update
BEFORE UPDATE ON ura
FOR EACH ROW
BEGIN

    raise_application_error(-20000, 'You cannot alter an existing user-role assignment.');

END;

/

CREATE OR REPLACE TRIGGER ura_after_delete
AFTER DELETE ON ura
FOR EACH ROW
DECLARE
    v_active_usr INTEGER;
BEGIN

    SELECT count(*) INTO v_active_usr FROM currently_active
        WHERE usr = :old.usr;

    IF v_active_usr > 0 THEN
        revoke_role(full_db_user(:old.role),full_db_user(:old.usr));
    END IF;
END;

/

CREATE OR REPLACE TRIGGER ura_after_insert
AFTER INSERT ON ura
FOR EACH ROW
DECLARE
    v_active_usr INTEGER;
BEGIN

    SELECT count(*) INTO v_active_usr FROM currently_active
      WHERE usr = :new.usr;

    IF v_active_usr > 0 THEN -- this shouldn't happen: user should not be active in role
unless ura already there
        grant_role(full_db_user(:new.role),full_db_user(:new.usr));
    END IF;
END;

/


CREATE OR REPLACE TRIGGER password_after_insert
AFTER INSERT ON password
FOR EACH ROW
BEGIN
    create_user( full_db_user(:new.User_Id), :new.password);
END;

/

CREATE OR REPLACE TRIGGER password_after_delete
AFTER DELETE ON password
FOR EACH ROW
BEGIN
    drop_user(full_db_user(:old.User_Id));
END;

/


CREATE OR REPLACE TRIGGER usr_session_after_insert
AFTER INSERT ON usr_session
FOR EACH ROW
BEGIN
    grant_role(full_db_user(:new.role),full_db_user(:new.usr));
```

```
END;

/


CREATE OR REPLACE TRIGGER usr_session_after_update
AFTER UPDATE ON usr_session
FOR EACH ROW
BEGIN
    IF (:new.end_time >= SYSDATE AND :old.end_time is null) THEN
        revoke_role(full_db_user(:new.role),full_db_user(:new.usr));
    END IF;
END;

/


CREATE OR REPLACE TRIGGER usr_session_before_insert
BEFORE INSERT ON usr_session
FOR EACH ROW
DECLARE
    v_conflicting_roles INTEGER;
    v_current_sessions INTEGER;
    v_ura INTEGER;
BEGIN

    -- Check for existing active session with this user and role.
    -- If exists, cannot activate a new one.

    SELECT count(*) INTO v_current_sessions
      FROM currently_active
      WHERE usr = :new.usr AND role = :new.role;

    if v_current_sessions > 0 THEN
        raise_application_error(-20000, 'User already active in role: cannot activate ' ||
:new.usr || ' as ' || :new.role || '.');
    END IF;

    -- Check for dynamic separation of duty
    -- rules that conflict with this attempt to
    -- activate a user with a role.
    -- If any are found, then the activation cannot take place.

    -- Implements rule
    -- inconsistent_dsd(U,R1,D1) :- activate(U,R2,D2,Password),
    --                              password(U,Password),
    --                              dsd_conflict(R1,R2),
    --                              date_time_stamp(D1,T1),
    --                              (
    --                                not(deactivate(U,R2,_));
    --                                deactivate(U,R2,D3),
    --                                date_time_stamp(D3,T3),
    --                                T3 < T1
    --                              ),
    --                              date_time_stamp(D2,T2),
    --                              T2 =< T1.


    :new.start_time := CURRENT_TIMESTAMP;

    SELECT count(*) INTO v_conflicting_roles
        FROM dsd_conflict, usr_session
        WHERE dsd_conflict.role1 = usr_session.role
        AND dsd_conflict.role2 = :new.role
        AND usr_session.usr = :new.usr
        AND usr_session.start_time < CURRENT_TIMESTAMP
        AND (usr_session.end_time > CURRENT_TIMESTAMP or usr_session.end_time is null);

    SELECT count(*) INTO v_ura FROM ura
        WHERE usr = :new.usr AND role = :new.role;
```

```
    IF v_conflicting_roles > 0 THEN
        raise_application_error(-20000, 'Conflicting roles: cannot activate ' || :new.usr || '
as ' || :new.role || '.');
    END IF;

    IF v_conflicting_roles > 0 OR v_ura = 0 THEN
        raise_application_error(-20000, 'Not assigned to role: cannot activate ' || :new.usr
|| ' as ' || :new.role || '.');
    END IF;
 END;


/


CREATE OR REPLACE TRIGGER usr_session_before_update
BEFORE UPDATE ON usr_session
FOR EACH ROW
BEGIN
   IF :new.usr <> :old.usr OR :new.role <> :old.role OR :new.start_time <> :old.start_time
OR :new.end_time < SYSDATE THEN
        RAISE_APPLICATION_ERROR(-20000, 'You cannot update a session once created, except to
end it.');
    END IF;
END;

/

CREATE OR REPLACE TRIGGER usr_session_before_delete
BEFORE DELETE ON usr_session
FOR EACH ROW
BEGIN
    RAISE_APPLICATION_ERROR(-20000, 'You cannot delete a session once created.');
END;

/

CREATE OR REPLACE TRIGGER password_before_update
BEFORE UPDATE ON password
FOR EACH ROW
BEGIN
    IF :new.user_id <> :old.user_id THEN
        RAISE_APPLICATION_ERROR(-20000, 'You cannot change a user''s ID once the user has been
created.');
    END IF;
END;

/

CREATE OR REPLACE TRIGGER usr_before_update
BEFORE UPDATE ON usr
FOR EACH ROW
BEGIN
    IF :new.user_id <> :old.user_id THEN
        RAISE_APPLICATION_ERROR(-20000, 'You cannot change a user''s ID once the user has been
created.');
    END IF;
END;

/
```

Functions

```
CREATE OR REPLACE FUNCTION is_part_of( p_inner_role VARCHAR, p_outer_role VARCHAR )
RETURN BOOLEAN
IS
  v_num_rows1 INT;
  v_num_rows2 INT;
BEGIN
  SELECT COUNT(*) INTO v_num_rows1 FROM included_in, senior_to WHERE
    p_inner_role = included_in.inner_role AND
```

```
    included_in.outer_role = senior_to.junior_role AND
    senior_to.senior_role = p_outer_role;
  SELECT COUNT(*) INTO v_num_rows2 FROM included_in WHERE
    p_inner_role = included_in.inner_role AND
    included_in.outer_role = p_outer_role;
  RETURN (v_num_rows1 + v_num_rows2 > 0);
END;


/

-- replace single with double quote for safe passage.
-- this is equivalent to a predefined function in Postgres
CREATE OR REPLACE FUNCTION quote_ident( username VARCHAR )
RETURN VARCHAR
IS
BEGIN
  RETURN replace(username, '''', '''''');
END;

/

CREATE OR REPLACE FUNCTION get_username
RETURN VARCHAR
IS
  v_username VARCHAR(64);
BEGIN
  SELECT user INTO v_username FROM DUAL;
  RETURN v_username;

END;

/


CREATE OR REPLACE FUNCTION get_schema
RETURN VARCHAR
IS
  v_username VARCHAR(64);
  v_schema VARCHAR(64);
BEGIN

  v_username := get_username;

  SELECT DEFAULT_TABLESPACE INTO v_schema
  FROM sys.dba_users WHERE username  = v_username;

  RETURN v_schema;

END;

/

CREATE OR REPLACE FUNCTION get_usr
RETURN VARCHAR
IS
  v_username VARCHAR(64);
BEGIN
  select user into v_username from dual;
  return nls_lower(replace(v_username, get_schema() || '1_', ''));
END;

/

CREATE OR REPLACE FUNCTION full_db_user( username VARCHAR )
RETURN VARCHAR
IS
  v_schema VARCHAR(64);
BEGIN
    v_schema := get_schema();
    RETURN v_schema || '1_' || quote_ident(username);
```

```
END;

/

CREATE OR REPLACE PROCEDURE insert_senior_to
IS
BEGIN
  -- Add entries to table senior_to according to each rule,
  -- if they are not already there.


  -- senior_to(R1,R1) :- d_s(R1,_).
  INSERT INTO senior_to(
    SELECT DISTINCT senior_role,senior_role FROM d_s
      WHERE (senior_role,senior_role) NOT IN
      (SELECT senior_role,junior_role from senior_to)
  );

  -- senior_to(R1,R1) :- d_s(_,R1).
  INSERT INTO senior_to(
    SELECT DISTINCT junior_role,junior_role FROM d_s
      WHERE (junior_role,junior_role) NOT IN
      (SELECT senior_role,junior_role from senior_to)
  );

  -- senior_to(R1,R2) :- d_s(R1,R2).
  INSERT INTO senior_to(
    SELECT DISTINCT senior_role,junior_role FROM d_s
      WHERE (senior_role,junior_role) NOT IN
      (SELECT senior_role,junior_role from senior_to)
  );

END;

/

CREATE OR REPLACE PROCEDURE recourse_senior_to
IS
  v_rows INT;
BEGIN

  DELETE FROM senior_to_staging;

  -- senior_to(R1,R2) :- d_s(R1,R3), senior_to(R3,R2).
  INSERT INTO senior_to_staging(
    (SELECT DISTINCT senior_to.senior_role, d_s.junior_role
      FROM d_s JOIN senior_to
      ON d_s.senior_role = senior_to.junior_role
    )
--      WHERE (senior_to.senior_role, d_s.junior_role) NOT IN
      MINUS
      (SELECT senior_role,junior_role from senior_to)
  );

  SELECT COUNT(*) INTO v_rows FROM senior_to_staging;
  IF (v_rows > 0 ) THEN
    INSERT INTO senior_to (
      (SELECT * FROM senior_to_staging)
      MINUS
      (SELECT senior_role,junior_role from senior_to)
    );
  END IF;

  DELETE FROM senior_to_staging;

END;

/

CREATE OR REPLACE PROCEDURE delete_senior_to
IS
```

```
BEGIN
  DELETE FROM senior_to
    WHERE junior_role <> senior_role
    AND (
      junior_role NOT IN (SELECT junior_role FROM d_s)
      OR senior_role NOT IN (SELECT senior_role FROM d_s)
    )
    OR junior_role = senior_role
    AND (
      junior_role NOT IN (SELECT junior_role FROM d_s)
      AND senior_role NOT IN (SELECT senior_role FROM d_s)
    )
  ;
END;

/

CREATE OR REPLACE PROCEDURE insert_included_in
IS
BEGIN

    -- Add entries to table included_in according to each rule,
    -- if they are not already there.


    -- included_in(R1,R2) :- is_a(R1,R2).
    INSERT INTO included_in(
        SELECT DISTINCT is_a.inner_role,is_a.outer_role
            FROM is_a WHERE (inner_role,outer_role) NOT IN
            (SELECT inner_role,outer_role from included_in)
    );
END;

/

CREATE OR REPLACE PROCEDURE recourse_included_in
IS
  v_rows INT;
BEGIN

  DELETE FROM included_in_staging;

  -- included_in(R1,R3) :- is_a(R1,R2), included_in(R2,R3).
  INSERT INTO included_in_staging(
    (SELECT DISTINCT included_in.inner_role, is_a.outer_role
    FROM is_a JOIN included_in
    ON is_a.inner_role = included_in.outer_role)
    MINUS
    (SELECT inner_role,outer_role from included_in)
  );

  SELECT COUNT(*) INTO v_rows FROM included_in_staging;
  IF (v_rows > 0 ) THEN
    INSERT INTO included_in (SELECT * FROM included_in_staging);
  END IF;

  DELETE FROM included_in_staging;

END;


/

CREATE OR REPLACE PROCEDURE grant_role( role1 VARCHAR, role2 VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    EXECUTE IMMEDIATE 'GRANT ' || role1
         || ' TO ' || role2;
END;
```

```
/


CREATE OR REPLACE PROCEDURE revoke_role( role1 VARCHAR, role2 VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    EXECUTE IMMEDIATE 'REVOKE ' || role1
         || ' FROM ' || role2;
END;


/

CREATE OR REPLACE PROCEDURE create_user( user_id VARCHAR, password VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'CREATE USER ' || user_id || ' IDENTIFIED BY "' || password  || '" DEFAULT
TABLESPACE ' || get_schema();
  EXECUTE IMMEDIATE 'GRANT CREATE SESSION TO ' || user_id;
  EXECUTE IMMEDIATE 'GRANT EXECUTE ANY PROCEDURE TO ' || user_id;
  EXECUTE IMMEDIATE 'GRANT EXECUTE ON DBMS_RLS TO ' || user_id;
  EXECUTE IMMEDIATE 'GRANT EXECUTE ON DBMS_SESSION TO ' || user_id;
  EXECUTE IMMEDIATE 'GRANT ADMINISTER DATABASE TRIGGER TO ' || user_id;
  EXECUTE IMMEDIATE 'GRANT EXECUTE ON "' || get_schema() || '"."SET_CONTEXT" TO ' || user_id;
END;

/

CREATE OR REPLACE PROCEDURE drop_user( user_id VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    EXECUTE IMMEDIATE 'DROP USER ' || user_id;
END;

/


CREATE OR REPLACE PROCEDURE create_role( role VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'CREATE ROLE ' || role;
END;

/

CREATE OR REPLACE PROCEDURE drop_role( role VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP ROLE ' || role;
END;

/

CREATE OR REPLACE PROCEDURE grant_priv( action VARCHAR, object VARCHAR, role VARCHAR )
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'GRANT ' || action
         || ' ON ' ||object
         || ' TO ' || role;
END;

/

CREATE OR REPLACE PROCEDURE revoke_priv( action VARCHAR, object VARCHAR, role VARCHAR )
IS
```

```
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    EXECUTE IMMEDIATE 'REVOKE ' || action
         || ' ON ' || object
         || ' FROM ' || role;
END;

/
```

Appendix IX: SQL Code for Dynamic RBAC: Generic

Tables

```
CREATE TABLE tbl_rows (
  row_id VARCHAR(256),
  object VARCHAR(64)
);
```

Views 1

These are run *before* setting up context constraints for databases to which RBAC model is applied.

```
CREATE VIEW permittable_by_row AS
  SELECT usr, permittable.object as object, action, role, row_id
    FROM permittable, tbl_rows
    WHERE permittable.object = tbl_rows.object;

CREATE VIEW permitted_by_row AS
  SELECT usr, permitted.object as object, action, role, row_id
    FROM permitted, tbl_rows
    WHERE permitted.object = tbl_rows.object;

CREATE VIEW authorizable_by_row AS
  SELECT usr, authorizable.object as object, action, role, row_id
    FROM authorizable, tbl_rows
    WHERE authorizable.object = tbl_rows.object;

CREATE VIEW authorized_by_row AS
  SELECT usr, authorized.object as object, action, role, row_id
    FROM authorized, tbl_rows
    WHERE authorized.object = tbl_rows.object;
```

Views 2

These are run *after* setting up context constraints for databases to which RBAC model is applied, as this is when `fails_context_constraints` is set up.

```
CREATE VIEW permittable_cc AS
  SELECT usr, object, row_id, action, role FROM permittable_by_row
    MINUS
  SELECT usr, object, row_id, action, role FROM fails_context_constraints
;

CREATE VIEW permitted_cc AS
  SELECT usr, object, row_id, action, role FROM permitted_by_row
    MINUS
  SELECT usr, object, row_id, action, role FROM fails_context_constraints
;

CREATE VIEW authorizable_cc AS
  SELECT usr, object, row_id, action, role FROM authorizable_by_row
    MINUS
  SELECT usr, object, row_id, action, role FROM fails_context_constraints
;

CREATE VIEW authorized_cc AS
  SELECT usr, object, row_id, action, role FROM authorized_by_row
    MINUS
  SELECT usr, object, row_id, action, role FROM fails_context_constraints
;
```

Triggers

```
-- security context

CREATE OR REPLACE TRIGGER cc_logon_trigger
```

```
AFTER LOGON ON DATABASE
DECLARE
  v_latest_logon VARCHAR(30);
  v_fulluser VARCHAR(30);
  v_user VARCHAR(30) := get_usr();
  v_schema VARCHAR(30) := get_schema();
  v_grant VARCHAR(255);
  v_role VARCHAR(30) := '';

  CURSOR c_get_roles IS
    SELECT role FROM currently_active WHERE usr = (SELECT get_usr FROM DUAL);
BEGIN

  -- get user
  OPEN c_get_roles;

  LOOP
    FETCH c_get_roles INTO v_role;
    EXIT WHEN c_get_roles%NOTFOUND;

--     set context constraints
    set_context.set_cc(v_role);
--     set_context.set_day_duty;
  END LOOP;
  CLOSE c_get_roles;

END;

/
```

Appendix X: SQL Code for Dynamic RBAC: Hospital Database

Tables

```
CREATE TABLE patient_bed ( -- needed for nurse_in_same_ward_as_patient
  patient_id VARCHAR(256),
  bed_id VARCHAR(64)
);
```

Views

These correspond to context constraints.

```
-- context constraints

CREATE VIEW nurse_patient AS
  SELECT permittable_by_row.usr as usr, action, object, row_id
    FROM permittable_by_row, nurse_ward, room, bed, patient
    WHERE object = 'patient'
    AND permittable_by_row.usr = nurse_ward.usr
    AND nurse_ward.ward = room.ward_id
    AND room.room_id = bed.room_id
    AND bed.bed_id = patient.bed_id
    AND patient.patient_id = permittable_by_row.row_id
    ORDER BY usr, action, row_id;

CREATE VIEW patient_doctor AS
  SELECT usr, action, object, row_id
    FROM permittable_by_row, ae_consultation
    WHERE object = 'patient'
    AND permittable_by_row.usr = ae_consultation.doctor_id
    AND ae_consultation.patient_id = permittable_by_row.row_id
  UNION
  SELECT usr, action, object, row_id
    FROM permittable_by_row, ae_consultation, patient_diagnosis
    WHERE object = 'patient'
    AND permittable_by_row.usr = patient_diagnosis.diagnosing_doctor
    AND patient_diagnosis.cons_number = ae_consultation.cons_number
    AND ae_consultation.patient_id = permittable_by_row.row_id;

CREATE VIEW day_duty AS
  SELECT usr, action, object, row_id
    FROM permittable_by_row
    WHERE TO_CHAR (SYSDATE, 'HH24') >= 9 AND TO_CHAR (SYSDATE, 'HH24') < 21;

CREATE VIEW night_duty AS
  SELECT usr, action, object, row_id
    FROM permittable_by_row
    WHERE TO_CHAR (SYSDATE, 'HH24') < 9 OR TO_CHAR (SYSDATE, 'HH24') >= 21;

CREATE VIEW weekend_duty AS
  SELECT usr, action, object, row_id
    FROM permittable_by_row
    WHERE  TO_CHAR (SYSDATE, 'D') = 7  OR TO_CHAR (SYSDATE, 'D') = 1;

CREATE VIEW office_hours AS
  SELECT usr, action, object, row_id
    FROM permittable_by_row
    WHERE (
      TO_CHAR (SYSDATE, 'D') >= 2  AND TO_CHAR (SYSDATE, 'D') <= 6
    ) AND (
      TO_CHAR (SYSDATE, 'HH24') >= 9 AND TO_CHAR (SYSDATE, 'HH24') < 17
    );

CREATE VIEW staff_nurse_sister_2h AS
  SELECT DISTINCT permittable_by_row.usr, action, object, row_id
    FROM permittable_by_row,included_in,currently_active
    WHERE currently_active.role = included_in.inner_role
```

```
    AND (included_in.outer_role = 'sister' OR included_in.outer_role = 'staff_nurse')
    AND currently_active.start_time <  SYSDATE - 2/24;

-- application
CREATE VIEW applies_nurse_patient AS
  SELECT DISTINCT role, object FROM nurse_patient, included_in, senior_to, ura
  WHERE object = 'patient'
  AND nurse_patient.usr = ura.usr
  AND ura.role = included_in.inner_role
  AND included_in.outer_role = senior_to.junior_role
  AND senior_to.senior_role = 'sister';

CREATE VIEW applies_patient_doctor AS
  SELECT DISTINCT role, object FROM patient_doctor, included_in, senior_to, ura
  WHERE object = 'patient'
  AND patient_doctor.usr = ura.usr
  AND ura.role = included_in.inner_role
  AND included_in.outer_role = senior_to.junior_role
  AND senior_to.senior_role = 'snr_house_officer';

CREATE VIEW applies_day_duty AS
  SELECT DISTINCT inner_role AS role FROM is_a
  WHERE outer_role = 'day_duty';

CREATE VIEW applies_night_duty AS
  SELECT DISTINCT inner_role AS role FROM is_a
  WHERE outer_role = 'night_duty';

CREATE VIEW applies_office_hours AS
  SELECT DISTINCT role FROM role
  WHERE (
    role = 'office_hours'
  );

CREATE VIEW applies_staff_nurse_sister_2h AS
  SELECT DISTINCT inner_role AS role FROM included_in
  WHERE outer_role = 'student_nurse';

CREATE VIEW fails_nurse_patient AS
  SELECT DISTINCT ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM nurse_patient
  ) AND (ura.role, object) IN (
    SELECT role, object FROM applies_nurse_patient
  );

CREATE VIEW fails_patient_doctor AS
  SELECT DISTINCT ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM patient_doctor
  ) AND (ura.role, object) IN (
    SELECT role, object FROM applies_patient_doctor
  );

CREATE VIEW fails_day_duty AS
  SELECT DISTINCT  ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM day_duty
  ) AND ura.role IN (
    SELECT role FROM applies_day_duty
  );

CREATE VIEW fails_night_duty AS
  SELECT DISTINCT ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
```

```
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM night_duty
  ) AND ura.role IN (
    SELECT role FROM applies_night_duty
  );

CREATE VIEW fails_office_hours AS
  SELECT DISTINCT ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM office_hours
  ) AND ura.role IN (
    SELECT role FROM applies_office_hours
  );

CREATE VIEW fails_staff_nurse_sister_2h AS
  SELECT DISTINCT ura.usr AS usr, object, row_id, action, ura.role AS role FROM
permittable_by_row, ura
  WHERE permittable_by_row.usr = ura.usr
  AND (ura.usr, action, object, row_id) NOT IN (
    SELECT usr, action, object, row_id FROM staff_nurse_sister_2h
  ) AND ura.role IN (
    SELECT role FROM applies_staff_nurse_sister_2h
  );


CREATE VIEW fails_context_constraints AS
  SELECT usr, object, row_id, action, role FROM fails_nurse_patient
    UNION
  SELECT usr, object, row_id, action, role FROM fails_patient_doctor
    UNION
  SELECT usr, object, row_id, action, role FROM fails_day_duty
    UNION
  SELECT usr, object, row_id, action, role FROM fails_night_duty
    UNION
  SELECT usr, object, row_id, action, role FROM fails_office_hours
    UNION
  SELECT usr, object, row_id, action, role FROM fails_staff_nurse_sister_2h
;
```

Triggers

```
CREATE OR REPLACE TRIGGER patient_insert
AFTER INSERT ON patient
FOR EACH ROW
BEGIN

  INSERT INTO patient_bed(patient_id, bed_id) values( :new.patient_id, :new.bed_id );
  INSERT INTO tbl_rows(row_id, object) values( :new.patient_id, 'patient' );

END;

/

CREATE OR REPLACE TRIGGER patient_update
AFTER UPDATE ON patient
FOR EACH ROW
BEGIN

  UPDATE patient_bed SET patient_id = :new.patient_id, bed_id = :new.bed_id WHERE patient_id =
:old.patient_id AND bed_id = :old.bed_id;
  UPDATE tbl_rows SET row_id = :new.patient_id WHERE row_id = :old.patient_id AND object =
'patient';

END;

/

CREATE OR REPLACE TRIGGER patient_delete
```

```
AFTER DELETE ON patient
FOR EACH ROW
BEGIN

  DELETE FROM patient_bed WHERE patient_id = :old.patient_id AND bed_id = :old.bed_id;
  DELETE FROM tbl_rows WHERE row_id = :old.patient_id AND object = 'patient';

END;

/

CREATE OR REPLACE TRIGGER ward_insert
AFTER INSERT ON ward
FOR EACH ROW
BEGIN

  INSERT INTO tbl_rows(row_id, object) values( :new.ward_id, 'ward' );

END;

/

CREATE OR REPLACE TRIGGER room_insert
AFTER INSERT ON room
FOR EACH ROW
BEGIN

  INSERT INTO tbl_rows(row_id, object) values( :new.room_id, 'room' );

END;

/

CREATE OR REPLACE TRIGGER bed_insert
AFTER INSERT ON bed
FOR EACH ROW
BEGIN

  INSERT INTO tbl_rows(row_id, object) values( :new.bed_id, 'bed' );

END;

/

CREATE OR REPLACE TRIGGER diagnosis_insert
AFTER INSERT ON diagnosis
FOR EACH ROW
BEGIN

  INSERT INTO tbl_rows(row_id, object) values( :new.diagnosis_code, 'diagnosis' );

END;

/

CREATE OR REPLACE TRIGGER ae_consultation_insert
AFTER INSERT ON ae_consultation
FOR EACH ROW
BEGIN

  INSERT INTO tbl_rows(row_id, object) values( :new.cons_number, 'ae_consultation' );

END;

/

CREATE OR REPLACE TRIGGER patient_diagnosis_insert
AFTER INSERT ON patient_diagnosis
FOR EACH ROW
BEGIN
```

```
  INSERT INTO tbl_rows(row_id, object) values( :new.patient_diagnosis_number,
'patient_diagnosis' );

END;

/

CREATE OR REPLACE TRIGGER ward_update
AFTER UPDATE ON ward
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.ward_id WHERE row_id = :old.ward_id AND object = 'ward';

END;

/

CREATE OR REPLACE TRIGGER room_update
AFTER UPDATE ON room
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.room_id WHERE row_id = :old.room_id AND object = 'room';

END;

/

CREATE OR REPLACE TRIGGER bed_update
AFTER UPDATE ON bed
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.bed_id WHERE row_id = :old.bed_id AND object = 'bed';

END;

/

CREATE OR REPLACE TRIGGER diagnosis_update
AFTER UPDATE ON diagnosis
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.diagnosis_code WHERE row_id = :old.diagnosis_code AND
object = 'diagnosis';

END;

/

CREATE OR REPLACE TRIGGER ae_consultation_update
AFTER UPDATE ON ae_consultation
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.cons_number WHERE row_id = :old.cons_number AND object =
'ae_consultation';

END;

/

CREATE OR REPLACE TRIGGER patient_diagnosis_update
AFTER UPDATE ON patient_diagnosis
FOR EACH ROW
BEGIN

  UPDATE tbl_rows SET row_id = :new.patient_diagnosis_number WHERE row_id =
:old.patient_diagnosis_number AND object = 'patient_diagnosis';
```

```
END;

/

CREATE OR REPLACE TRIGGER ward_delete
AFTER DELETE ON ward
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.ward_id AND object = 'ward';

END;

/

CREATE OR REPLACE TRIGGER room_delete
AFTER DELETE ON room
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.room_id AND object = 'room';

END;

/

CREATE OR REPLACE TRIGGER bed_delete
AFTER DELETE ON bed
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.bed_id AND object = 'bed';

END;

/

CREATE OR REPLACE TRIGGER diagnosis_delete
AFTER DELETE ON diagnosis
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.diagnosis_code AND object = 'diagnosis';

END;

/

CREATE OR REPLACE TRIGGER ae_consultation_delete
AFTER DELETE ON ae_consultation
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.cons_number AND object = 'ae_consultation';

END;

/

CREATE OR REPLACE TRIGGER patient_diagnosis_delete
AFTER DELETE ON patient_diagnosis
FOR EACH ROW
BEGIN

  DELETE FROM tbl_rows WHERE row_id = :old.patient_diagnosis_number AND object =
'patient_diagnosis';

END;-

/
```

Appendix XI: Oracle VPD Context for Hospital Database

Head

```
CREATE OR REPLACE PACKAGE set_context
IS
  PROCEDURE set_cc( p_role VARCHAR );

  PROCEDURE set_day_duty;
  PROCEDURE set_night_duty;
  PROCEDURE set_office_hours;
  PROCEDURE set_staff_sister_active_2_h;
  PROCEDURE set_nurse_ward;
  PROCEDURE set_patient_doctor;

  PROCEDURE set_denials( p_role VARCHAR );
END;
/
```

Body

```
CREATE OR REPLACE PACKAGE BODY set_context
AS

  PROCEDURE set_cc( p_role VARCHAR )
  IS
  BEGIN
    IF( is_part_of(p_role,'day_duty' )) THEN
      set_day_duty;
    END IF;
    IF( is_part_of(p_role,'night_duty' )) THEN
      set_night_duty;
    END IF;
    IF( is_part_of(p_role,'sister')) THEN
      set_nurse_ward;
    END IF;
    IF( is_part_of(p_role, 'snr_house_officer')) THEN
      set_patient_doctor;
    END IF;
    IF( is_part_of(p_role, 'office_hours')) THEN
      set_office_hours;
    END IF;
    IF( is_part_of(p_role, 'student_nurse' )) THEN
      set_staff_sister_active_2_h;
    END IF;
    set_denials(p_role);
  END;

  PROCEDURE set_day_duty
  IS
  BEGIN
    dbms_session.set_context('hosp', 'day_duty', 'y');
  END;

  PROCEDURE set_night_duty
  IS
  BEGIN
    dbms_session.set_context('hosp', 'night_duty', 'y');
  END;

  PROCEDURE set_office_hours
  IS
  BEGIN
    dbms_session.set_context('hosp', 'office_hours', 'y');
  END;

  PROCEDURE set_staff_sister_active_2_h
  IS
  BEGIN
```

```
    dbms_session.set_context('hosp', 'staff_sister_active_2_h', 'y');
  END;

  PROCEDURE set_nurse_ward
  IS
  BEGIN
    dbms_session.set_context('hosp', 'nurse_in_same_ward_as_patient', 'y');
  END;

  PROCEDURE set_patient_doctor
  IS
  BEGIN
    dbms_session.set_context('hosp', 'patient_treated_by_doctor', 'y');
  END;


  PROCEDURE set_denials( p_role VARCHAR )
  IS
    v_action VARCHAR(64);
    v_object VARCHAR(64);
  CURSOR c_get_denials IS
    SELECT action, object FROM d_rpa_full WHERE role = p_role;
  BEGIN

    OPEN c_get_denials;

    LOOP
      FETCH c_get_denials INTO v_action, v_object;
      EXIT WHEN c_get_denials%NOTFOUND;

      -- set context constraints
      dbms_session.set_context('hosp', 'denied_' || v_object || '_' || v_action, 'y');
    END LOOP;

    CLOSE c_get_denials;

  END;

end;
/
```

Appendix XII: Oracle VPD Policy for Hospital Database

Adding

```
begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT',
            policy_name => 'CC_PATIENT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT',
            policy_name => 'CC_PATIENT_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT',
            policy_name => 'CC_PATIENT_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT',
            policy_name => 'CC_PATIENT_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT',
            policy_name => 'CC_PATIENT_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
```

```
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'BED',
            policy_name => 'CC_BED',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'BED',
            policy_name => 'CC_BED_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'BED',
            policy_name => 'CC_BED_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'BED',
            policy_name => 'CC_BED_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'BED',
            policy_name => 'CC_BED_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
```

```
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'ROOM',
            policy_name => 'CC_ROOM',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'ROOM',
            policy_name => 'CC_ROOM_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'ROOM',
            policy_name => 'CC_ROOM_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'ROOM',
            policy_name => 'CC_ROOM_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'ROOM',
            policy_name => 'CC_ROOM_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
```

```
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'WARD',
            policy_name => 'CC_WARD',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'WARD',
            policy_name => 'CC_WARD_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'WARD',
            policy_name => 'CC_WARD_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'WARD',
            policy_name => 'CC_WARD_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'WARD',
            policy_name => 'CC_WARD_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
```

```
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'DIAGNOSIS',
            policy_name => 'CC_DIAGNOSIS',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'DIAGNOSIS',
            policy_name => 'CC_DIAGNOSIS_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'DIAGNOSIS',
            policy_name => 'CC_DIAGNOSIS_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'DIAGNOSIS',
            policy_name => 'CC_DIAGNOSIS_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'DIAGNOSIS',
            policy_name => 'CC_DIAGNOSIS_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/
```

```
begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'AE_CONSULTATION',
            policy_name => 'CC_AE_CONSULTATION',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'AE_CONSULTATION',
            policy_name => 'CC_AE_CONSULTATION_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'AE_CONSULTATION',
            policy_name => 'CC_AE_CONSULTATION_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'AE_CONSULTATION',
            policy_name => 'CC_AE_CONSULTATION_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'AE_CONSULTATION',
            policy_name => 'CC_AE_CONSULTATION_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/
```

```
begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT_DIAGNOSIS',
            policy_name => 'CC_PATIENT_DIAGNOSIS',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT_DIAGNOSIS',
            policy_name => 'CC_PATIENT_DIAGNOSIS_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT_DIAGNOSIS',
            policy_name => 'CC_PATIENT_DIAGNOSIS_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT_DIAGNOSIS',
            policy_name => 'CC_PATIENT_DIAGNOSIS_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'PATIENT_DIAGNOSIS',
            policy_name => 'CC_PATIENT_DIAGNOSIS_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/
```

```
begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'USR',
            policy_name => 'CC_USR',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC',
            statement_types => 'select, insert, update, delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/


begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'USR',
            policy_name => 'CC_USR_SELECT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_SELECT',
            statement_types => 'select',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'USR',
            policy_name => 'CC_USR_INSERT',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_INSERT',
            statement_types => 'insert',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'USR',
            policy_name => 'CC_USR_UPDATE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_UPDATE',
            statement_types => 'update',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/

begin
    dbms_rls.add_policy(
            object_schema => 'HOSP',
            object_name => 'USR',
            policy_name => 'CC_USR_DELETE',
            function_schema => 'HOSP',
            policy_function => 'POLICY.CC_DELETE',
            statement_types => 'delete',
            update_check => TRUE,
            enable => TRUE,
            static_policy => FALSE);
end;
/
```

Dropping

```
begin
    dbms_rls.drop_policy('HOSP','PATIENT','CC_PATIENT');
end;
/

begin
    dbms_rls.drop_policy('HOSP','PATIENT','CC_PATIENT_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT','CC_PATIENT_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT','CC_PATIENT_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT','CC_PATIENT_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','BED','CC_BED');
end;
/

begin
    dbms_rls.drop_policy('HOSP','BED','CC_BED_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','BED','CC_BED_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','BED','CC_BED_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','BED','CC_BED_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','ROOM','CC_ROOM');
end;
/

begin
    dbms_rls.drop_policy('HOSP','ROOM','CC_ROOM_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','ROOM','CC_ROOM_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','ROOM','CC_ROOM_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','ROOM','CC_ROOM_DELETE');
end;
/


begin
```

```
    dbms_rls.drop_policy('HOSP','WARD','CC_WARD');
end;
/

begin
    dbms_rls.drop_policy('HOSP','WARD','CC_WARD_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','WARD','CC_WARD_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','WARD','CC_WARD_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','WARD','CC_WARD_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','DIAGNOSIS','CC_DIAGNOSIS');
end;

begin
    dbms_rls.drop_policy('HOSP','DIAGNOSIS','CC_DIAGNOSIS_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','DIAGNOSIS','CC_DIAGNOSIS_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','DIAGNOSIS','CC_DIAGNOSIS_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','DIAGNOSIS','CC_DIAGNOSIS_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','AE_CONSULTATION','CC_AE_CONSULTATION');
end;
/

begin
    dbms_rls.drop_policy('HOSP','AE_CONSULTATION','CC_AE_CONSULTATION_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','AE_CONSULTATION','CC_AE_CONSULTATION_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','AE_CONSULTATION','CC_AE_CONSULTATION_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','AE_CONSULTATION','CC_AE_CONSULTATION_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','PATIENT_DIAGNOSIS','CC_PATIENT_DIAGNOSIS');
end;
/
```

```
begin
    dbms_rls.drop_policy('HOSP','PATIENT_DIAGNOSIS','CC_PATIENT_DIAGNOSIS_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT_DIAGNOSIS','CC_PATIENT_DIAGNOSIS_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT_DIAGNOSIS','CC_PATIENT_DIAGNOSIS_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','PATIENT_DIAGNOSIS','CC_PATIENT_DIAGNOSIS_DELETE');
end;
/


begin
    dbms_rls.drop_policy('HOSP','USR','CC_USR');
end;
/

begin
    dbms_rls.drop_policy('HOSP','USR','CC_USR_SELECT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','USR','CC_USR_INSERT');
end;
/
begin
    dbms_rls.drop_policy('HOSP','USR','CC_USR_UPDATE');
end;
/
begin
    dbms_rls.drop_policy('HOSP','USR','CC_USR_DELETE');
end;
/
```

Appendix XIII: Hospital Database CREATE TABLE statements

```
CREATE TABLE ward
(
ward_id       VARCHAR(10),
type          VARCHAR(10),
ward_capacity VARCHAR(10),
primary key (ward_id)
);


CREATE TABLE room
(
room_id      VARCHAR(10),
ward_id      VARCHAR(10),
type         VARCHAR(10),
bed_capacity VARCHAR(10),
primary key (room_id),
Foreign Key (ward_id) references ward(ward_id)
);

CREATE TABLE bed
(
bed_id     VARCHAR(10),
room_id    VARCHAR(10),
type       VARCHAR(10),
primary key (bed_id),
Foreign Key (room_id) references room(room_id)
);

CREATE TABLE patient
(
patient_id VARCHAR(50),
last_name   VARCHAR(50),
first_name  VARCHAR(50),
address    VARCHAR(50),
date_of_birth        VARCHAR(10),
bed_id    Varchar(10),
Primary Key (patient_id),
Foreign key (bed_id) references bed(bed_id)
);


CREATE TABLE diagnosis
(
diagnosis_code  VARCHAR(10),
illness_name     VARCHAR(50),
usual_symptoms  Varchar(200),
Primary Key (diagnosis_code)
);


CREATE TABLE ae_consultation
(
cons_number      VARCHAR(10),
cons_date        VARCHAR(10),
cons_description VARCHAR(100),
patient_id       VARCHAR(50),
doctor_id        VARCHAR(16),
Primary Key (cons_number),
Foreign Key (patient_id) references patient(patient_id),
Foreign Key (doctor_id) references usr(user_id)
);


CREATE TABLE patient_diagnosis
(
patient_diagnosis_number VARCHAR(10),
diagnosing_doctor VARCHAR(16),
diagnosis_desc   VARCHAR(100),
```

```
cons_number       VARCHAR(10),
diagnosis_code    VARCHAR(10),
Primary Key (patient_diagnosis_number),
Foreign Key (diagnosing_doctor) references usr(user_id),
Foreign Key (cons_number) references ae_consultation(cons_number),
Foreign Key (diagnosis_code) references diagnosis(diagnosis_code)
);


CREATE TABLE nurse_ward
(
    usr VARCHAR(16) NOT NULL,
    ward  VARCHAR(10) NOT NULL,
    PRIMARY KEY (usr, ward),
    FOREIGN KEY (usr) REFERENCES usr(user_id),
    FOREIGN KEY (ward) REFERENCES ward(ward_id)
);
```

Appendix XIV: Test Script for RBAC Enforcement

```
ALTER SESSION SET CURRENT_SCHEMA="HOSP";

@format

select REGEXP_SUBSTR(user, 'U[0-9]+') "User" FROM DUAL;

SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';
SELECT * FROM all_users WHERE username LIKE 'HOSP1_%' ORDER BY USERNAME;
SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM
ALL_TAB_PRIVS WHERE TABLE_SCHEMA = 'HOSP';

/* select data */
SELECT user_id "User_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM usr;
SELECT * FROM ward;
SELECT * FROM room;
SELECT * FROM bed;
SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address
"Address", date_of_birth FROM patient;
SELECT diagnosis_code "Diag_Code", illness_name "Illness_Name", usual_symptoms
"Usual_Symptoms" FROM diagnosis;
SELECT cons_number "Cons_Num", cons_date, cons_description "Cons_Description", patient_id
"Patient_ID", doctor_id "Doctor_ID" FROM ae_consultation;
SELECT patient_diagnosis_number "PD_Num", diagnosing_doctor "Doctor_ID", diagnosis_desc
"Diagnosis_Desc", cons_number "Cons_Num", diagnosis_code "Diag_Code" FROM patient_diagnosis;
SELECT * FROM nurse_ward;

SELECT * FROM password;
SELECT * FROM role;
SELECT * FROM d_s;
SELECT * FROM senior_to;
SELECT * FROM ura;
SELECT * FROM rpa;
SELECT role "Role", action "Action", object "Object", senior_role "Senior Role", junior_role
"Junior Role" FROM rpa_full;
SELECT * FROM d_rpa;
SELECT role "Role", action "Action", object "Object", senior_role "Senior Role", junior_role
"Junior Role"  FROM d_rpa_full;
SELECT * FROM denied;
SELECT * FROM dsd;
SELECT * FROM dsd_conflict;
SELECT * FROM ssd;
SELECT * FROM ssd_conflict;
SELECT * FROM is_a;
SELECT * FROM included_in;
SELECT senior_role "Senior Role", junior_role "Junior Role", action "Action", object "Object"
FROM inherits_rpa;
SELECT senior_role "Senior Role", junior_role "Junior Role", action "Action", object "Object"
FROM inherits_rpa_path;
SELECT usr "User_ID", role "Role", start_time "Start_Time", end_time "End_Time" FROM
usr_session;
SELECT * FROM currently_active;
SELECT * FROM authorizable;
SELECT * FROM permittable;
SELECT * FROM permitted;
SELECT * FROM authorized;


/* insert data */
UPDATE patient SET date_of_birth = TO_DATE('1979-12-12', 'YYYY-MM-DD') WHERE patient_id =
12345;
UPDATE ward SET ward_capacity = 15 WHERE ward_id = 'ward2';
UPDATE room SET bed_capacity = 3 WHERE room_id = 'Room1H';
UPDATE bed SET type='Electric' WHERE bed_id = 'Bed001';
UPDATE usr SET date_of_birth = ('1976-06-15', 'YYYY-MM-DD') WHERE user_id = 'u0019';

UPDATE diagnosis SET usual_symptoms = usual_symptoms || ', with foaming at the mouth.' WHERE
diagnosis_code = 'diag003';
```

```
UPDATE ae_consultation SET cons_description = 'Diarrhea and Vomiting' WHERE cons_number =
'c00022';
UPDATE patient_diagnosis SET diagnosis_desc = 'Coronary Heart Disease' WHERE
patient_diagnosis_number = 'pd00008';


/* insert data */
INSERT INTO usr(
  user_id,
  last_name,
  first_name,
  address,
  date_of_birth
) VALUES (
  'u0030',
  'Juric',
  'Radmila',
  'University of Westminster',
  TO_DATE('1960-05-17', 'YYYY-MM-DD')
);

INSERT INTO ward (
  ward_id,
  type,
  ward_capacity
) VALUES (
  'ward3',
  'Operating',
  12
);

INSERT INTO room (
  room_id,
  ward_id,
  type,
  bed_capacity
) VALUES (
  'Room1G',
  'ward3',
  'Public',
  2
);

INSERT INTO bed (
  bed_id,
  room_id,
  type
) VALUES (
  'Bed023',
  'Room1G',
  'Electric'
);

INSERT INTO patient (
  patient_id,
  last_name,
  first_name,
  address,
  date_of_birth,
  bed_id
) VALUES (
  12367,
  'Christ',
  'Jesus H.',
  'The Stables, The Inn, Bethlehem',
  TO_DATE('0000-12-25', 'YYYY-MM-DD'),
  'Bed023'
);

INSERT INTO diagnosis (
  diagnosis_code,
```

```
  illness_name,
  usual_symptoms
) VALUES (
  'diag010',
  'Asthma',
  'Wheezing, lack of breath'
);

INSERT INTO ae_consultation (
  cons_number,
  cons_date,
  cons_description,
  patient_id,
  doctor_id
) VALUES (
  'c00023',
  TO_DATE('2007-08-17', 'YYYY-MM-DD'),
  'Diarrhea',
  12365,
  'u0001'
);


INSERT INTO patient_diagnosis (
  patient_diagnosis_number,
  diagnosing_doctor,
  diagnosis_desc,
  cons_number,
  diagnosis_code
) VALUES (
  'pd00023',
  'u0010',
  'Stomach infection',
  'c00023',
  'diag002'
);


/*
Pretend to delete data.
This only tests whether the tables are
accessible for condition.
the WHERE clause always fails, so
nothing actually gets deleted.
*/

DELETE FROM ae_consultation WHERE 0 <> 0;
DELETE FROM authorized WHERE 0 <> 0;
DELETE FROM bed WHERE 0 <> 0;
DELETE FROM diagnosis WHERE 0 <> 0;
DELETE FROM patient_diagnosis WHERE 0 <> 0;
DELETE FROM patient WHERE 0 <> 0;
DELETE FROM nurse_ward WHERE 0 <> 0;
DELETE FROM room WHERE 0 <> 0;
DELETE FROM ward WHERE 0 <> 0;


DELETE FROM currently_active WHERE 0 <> 0;
DELETE FROM authorizable WHERE 0 <> 0;
DELETE FROM permittable WHERE 0 <> 0;
DELETE FROM authorized WHERE 0 <> 0;
DELETE FROM permitted WHERE 0 <> 0;
DELETE FROM denied WHERE 0 <> 0;
DELETE FROM dsd_conflict WHERE 0 <> 0;
DELETE FROM ssd_conflict WHERE 0 <> 0;
DELETE FROM d_rpa_full WHERE 0 <> 0;
DELETE FROM rpa_full WHERE 0 <> 0;
DELETE FROM inherits_rpa WHERE 0 <> 0;

DELETE FROM dsd WHERE 0 <> 0;
DELETE FROM inherits_rpa_path WHERE 0 <> 0;
```

```
DELETE FROM is_a WHERE 0 <> 0;
DELETE FROM included_in WHERE 0 <> 0;
DELETE FROM d_s WHERE 0 <> 0;
DELETE FROM senior_to WHERE 0 <> 0;
DELETE FROM usr_session WHERE 0 <> 0;
DELETE FROM ssd WHERE 0 <> 0;
DELETE FROM usr WHERE 0 <> 0;
DELETE FROM role WHERE 0 <> 0;
DELETE FROM password WHERE 0 <> 0;
DELETE FROM ura WHERE 0 <> 0;
DELETE FROM d_rpa WHERE 0 <> 0;
DELETE FROM rpa WHERE 0 <> 0;
```

Appendix XV: Hospital Database RBAC INSERT Statements

```
connect hosp/hosp

INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0001','Sugar','Ed','1 Montgomery Ave',TO_DATE('12/06/1975', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0002','Python','Adam','45 Escort Road',TO_DATE('24/01/1950', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0003','Edmonds','Sophie','49 Convent Gardens',TO_DATE('10/10/1968', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0004','Bowie','Diane','253 Kings Road',TO_DATE('02/03/1962', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0005','Peters','Peter','59 Monkety Crescent',TO_DATE('19/01/1980', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0006','Davies','Sheena','10 Auchtermuchty Way',TO_DATE('15/02/1979', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0007','Williams','Lucie','23 Monkswood Drive',TO_DATE('15/07/1977', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0008','Jones','John','The Manse, Church Lane',TO_DATE('18/07/1977', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0009','Evans','Renate','3 Geering Road',TO_DATE('12/03/1970', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0010','Fish','Michael','The Vane, Weatherby',TO_DATE('28/12/1955', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0011','Ghosh','Chandra','10 Kennington Road',TO_DATE('11/07/1959', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0012','Kellett','James','104 The Vale',TO_DATE('15/02/1959', 'DD/MM/YYYY' ) );

INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0013','Jacobson','Lucinda','14 The Mansion',TO_DATE('01/02/1969', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0014','Jones','Hannah','13 Consort Road',TO_DATE('15/05/1955', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0015','Kenning','Stephen','10 Roadrunner Crescent',TO_DATE('13/01/1977', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0016','Strand','Jasmine','The Lodge, Linden Avenue',TO_DATE('15/06/1987',
'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0017','Canning','Elizabeth','100 Western Road',TO_DATE('22/03/1969', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0018','Clarkson','Jeremy','43 Vroom Vroom Road',TO_DATE('30/09/1962', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0019','Lewis','Christine','16 Trent Drive',TO_DATE('13/05/1980', 'DD/MM/YYYY' ) );

INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0020','Jackson','Lisa','56 Restorick Road',TO_DATE('12/09/1975', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0021','James','Wendy','40 Transvision Road',TO_DATE('07/05/1966', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0022','Darch','Ruth','31 Finstock Street',TO_DATE('21/06/1979', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0023','Lewis','Donald','15 Montana Lane',TO_DATE('29/12/1980', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0024','Davies','Caroline','10 The Avenue',TO_DATE('17/09/1971', 'DD/MM/YYYY' ) );

INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0025','Lewis','Charlotte','20 High Road',TO_DATE('06/07/1974', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0026','Davies','Jonathan','15 Low Road',TO_DATE('14/07/1959', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0027','Minnow','Robert','5 Montrose Place',TO_DATE('08/07/0966', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0028','Avery','Caspar','13 Cod Street',TO_DATE('15/08/1981', 'DD/MM/YYYY' ) );
INSERT INTO usr( user_id, last_name, first_name, address, date_of_birth ) VALUES
( 'u0029','McTaggart','James','10 Fortean Street',TO_DATE('21/02/1977', 'DD/MM/YYYY' ) );
```

```
INSERT INTO password( user_id, password) VALUES ( 'u0001','desk' );
INSERT INTO password( user_id, password) VALUES ( 'u0002','chair' );
INSERT INTO password( user_id, password) VALUES ( 'u0003','window' );
INSERT INTO password( user_id, password) VALUES ( 'u0004','brick' );
INSERT INTO password( user_id, password) VALUES ( 'u0005','mother' );
INSERT INTO password( user_id, password) VALUES ( 'u0006','tennis' );
INSERT INTO password( user_id, password) VALUES ( 'u0007','file' );
INSERT INTO password( user_id, password) VALUES ( 'u0008','cricket' );
INSERT INTO password( user_id, password) VALUES ( 'u0009','dragon' );
INSERT INTO password( user_id, password) VALUES ( 'u0010','cock' );
INSERT INTO password( user_id, password) VALUES ( 'u0011','onion' );
INSERT INTO password( user_id, password) VALUES ( 'u0012','thadeus' );


INSERT INTO password( user_id, password) VALUES ( 'u0013','re$t' );
INSERT INTO password( user_id, password) VALUES ( 'u0014','carlena' );
INSERT INTO password( user_id, password) VALUES ( 'u0015','walnut' );
INSERT INTO password( user_id, password) VALUES ( 'u0016','c001ie' );
INSERT INTO password( user_id, password) VALUES ( 'u0017','compile' );
INSERT INTO password( user_id, password) VALUES ( 'u0018','wheeler' );
INSERT INTO password( user_id, password) VALUES ( 'u0019','mcginty' );

INSERT INTO password( user_id, password) VALUES ( 'u0020','queen' );
INSERT INTO password( user_id, password) VALUES ( 'u0021','vamp' );
INSERT INTO password( user_id, password) VALUES ( 'u0022','woodstock' );
INSERT INTO password( user_id, password) VALUES ( 'u0023','bronze' );
INSERT INTO password( user_id, password) VALUES ( 'u0024','cruise' );

INSERT INTO password( user_id, password) VALUES ( 'u0025','cream' );
INSERT INTO password( user_id, password) VALUES ( 'u0026','rookie' );
INSERT INTO password( user_id, password) VALUES ( 'u0027','little_fish' );
INSERT INTO password( user_id, password) VALUES ( 'u0028','fern' );
INSERT INTO password( user_id, password) VALUES ( 'u0029','jimmy' );


INSERT INTO role( role ) VALUES ( 'nurse' );
INSERT INTO role( role ) VALUES ( 'doctor' );
INSERT INTO role( role ) VALUES ( 'data_manager' );
INSERT INTO role( role ) VALUES ( 'administrator' );

INSERT INTO role( role ) VALUES ( 'consultant' );
INSERT INTO role( role ) VALUES ( 'specialist_registrar' );
INSERT INTO role( role ) VALUES ( 'snr_house_officer' );
INSERT INTO role( role ) VALUES ( 'snr_house_officer_d' );
INSERT INTO role( role ) VALUES ( 'snr_house_officer_n' );
INSERT INTO role( role ) VALUES ( 'house_officer' );
INSERT INTO role( role ) VALUES ( 'house_officer_d' );
INSERT INTO role( role ) VALUES ( 'house_officer_n' );

INSERT INTO role( role ) VALUES ( 'specialist_nurse' );
INSERT INTO role( role ) VALUES ( 'sister' );
INSERT INTO role( role ) VALUES ( 'sister_d' );
INSERT INTO role( role ) VALUES ( 'sister_n' );
INSERT INTO role( role ) VALUES ( 'staff_nurse' );
INSERT INTO role( role ) VALUES ( 'staff_nurse_d' );
INSERT INTO role( role ) VALUES ( 'staff_nurse_n' );
INSERT INTO role( role ) VALUES ( 'student_nurse' );
INSERT INTO role( role ) VALUES ( 'student_nurse_d' );
INSERT INTO role( role ) VALUES ( 'student_nurse_n' );

INSERT INTO role( role ) VALUES ( 'snr_data_manager' );
INSERT INTO role( role ) VALUES ( 'jnr_data_manager' );
INSERT INTO role( role ) VALUES ( 'receptionist' );
INSERT INTO role( role ) VALUES ( 'manager' );

INSERT INTO role( role ) VALUES ( 'day_duty' );
INSERT INTO role( role ) VALUES ( 'night_duty' );
INSERT INTO role( role ) VALUES ( 'office_hours' );
```

```
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'consultant','specialist_registrar' );
INSERT INTO d_s( senior_role, junior_role ) VALUES
( 'specialist_registrar','snr_house_officer' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'snr_house_officer','house_officer' );

INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'specialist_nurse','sister' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'sister','staff_nurse' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'staff_nurse','student_nurse' );

INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'snr_data_manager','jnr_data_manager' );

INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'manager','receptionist' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'manager','consultant' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'manager','snr_data_manager' );
INSERT INTO d_s( senior_role, junior_role ) VALUES ( 'manager','specialist_nurse' );



INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'consultant','house_officer','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'specialist_nurse','student_nurse','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'snr_data_manager','jnr_data_manager','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','receptionist','_','_' );

INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','house_officer','select','ae_consultation');
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','staff_nurse','_','patient_diagnosis');
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','staff_nurse','select','_');

-- s/junior/jnr, s/senior/snr, s/night/n, s/day/d due to restriction in length of role names
in Oracle
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'student_nurse_d','student_nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'student_nurse_n','student_nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'staff_nurse_d','staff_nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'staff_nurse_n','staff_nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'sister_d','sister' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'sister_n','sister' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'student_nurse','nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'staff_nurse','nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'sister','nurse' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'specialist_nurse','nurse' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'house_officer_d','house_officer' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'house_officer_n','house_officer' );
INSERT INTO is_a( inner_role, outer_role ) VALUES
( 'snr_house_officer_d','snr_house_officer' );
INSERT INTO is_a( inner_role, outer_role ) VALUES
( 'snr_house_officer_n','snr_house_officer' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'house_officer','doctor' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'snr_house_officer','doctor' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'specialist_registrar','doctor' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'consultant','doctor' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'jnr_data_manager','data_manager' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'snr_data_manager','data_manager' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'receptionist','administrator' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'manager','administrator' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'student_nurse_d','day_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'staff_nurse_d','day_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'sister_d','day_duty' );
```

```
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'house_officer_d','day_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'snr_house_officer_d','day_duty' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'student_nurse_n','night_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'staff_nurse_n','night_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'sister_n','night_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'house_officer_n','night_duty' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'snr_house_officer_n','night_duty' );

INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'receptionist','office_hours' );
INSERT INTO is_a( inner_role, outer_role ) VALUES ( 'jnr_data_manager','office_hours' );


INSERT INTO dsd( role1, role2 ) VALUES ( 'jnr_data_manager', '_' );
INSERT INTO dsd( role1, role2 ) VALUES ( 'receptionist', 'nurse' );
INSERT INTO dsd( role1, role2 ) VALUES ( 'administrator', 'doctor' );
INSERT INTO dsd( role1, role2 ) VALUES ( 'day_duty', 'night_duty' );


INSERT INTO ssd( role1, role2 ) VALUES ( 'snr_data_manager', '_' );
INSERT INTO ssd( role1, role2 ) VALUES ( 'manager', 'consultant' );
INSERT INTO ssd( role1, role2 ) VALUES ( 'doctor', 'nurse' );



INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'usr' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'house_officer','select', 'patient_diagnosis'
);

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_house_officer','update', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_house_officer','update',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_house_officer','update',
'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'specialist_registrar','insert',
'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'consultant','insert', 'ae_consultation' );


INSERT INTO rpa( role, action, object ) VALUES ( 'student_nurse','select', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'student_nurse','select', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'student_nurse','select', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'student_nurse','select', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'student_nurse','select', 'usr' );

INSERT INTO rpa( role, action, object ) VALUES ( 'staff_nurse','update', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'staff_nurse','select', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'staff_nurse','select', 'usr' );
INSERT INTO rpa( role, action, object ) VALUES ( 'staff_nurse','select', 'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'staff_nurse','select',
'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'sister','update', 'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'specialist_nurse','update',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'specialist_nurse','update', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'specialist_nurse','insert', 'diagnosis' );


INSERT INTO rpa( role, action, object ) VALUES ( 'receptionist','select', 'patient' );
```

```
INSERT INTO rpa( role, action, object ) VALUES ( 'manager','insert', 'patient' );


INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert',
'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'ward' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'room' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'bed' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'patient' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'ae_consultation'
);

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'patient_diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'patient_diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'patient_diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter',
'patient_diagnosis' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'nurse_ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'nurse_ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'nurse_ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'nurse_ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'nurse_ward' );


INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'usr' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'usr' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'usr' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'usr' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'password' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'password' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'password' );
```

```
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'password' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'password' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'role' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'role' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'role' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'role' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'role' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'd_s' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'd_s' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'd_s' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'd_s' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'd_s' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'inherits_rpa_path' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'inherits_rpa_path' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'inherits_rpa_path' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'inherits_rpa_path' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter',
'inherits_rpa_path' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'is_a' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'is_a' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'is_a' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'is_a' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'is_a' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'rpa' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'ssd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'ssd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'ssd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'ssd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'ssd' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'dsd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'dsd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'dsd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'dsd' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'dsd' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'ura' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'ura' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'ura' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'ura' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'ura' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'd_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'd_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'd_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'd_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'd_rpa' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'usr_session' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'usr_session' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'usr_session' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'usr_session' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'usr_session' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'senior_to' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'senior_to' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'senior_to' );
```

```
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'senior_to' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'senior_to' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'included_in' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'included_in' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'included_in' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'included_in' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','alter', 'included_in' );


INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'currently_active' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'currently_active' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'currently_active' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'currently_active' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'authorizable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'authorizable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'authorizable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'authorizable' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'permittable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'permittable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'permittable' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'permittable' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'authorized' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'authorized' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'authorized' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'authorized' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'permitted' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'permitted' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'permitted' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'permitted' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'denied' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'denied' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'denied' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'denied' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'dsd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'dsd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'dsd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'dsd_conflict' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'ssd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'ssd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'ssd_conflict' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'ssd_conflict' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'd_rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'd_rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'd_rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'd_rpa_full' );
```

```
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select', 'rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert', 'rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update', 'rpa_full' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete', 'rpa_full' );

INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','select',
'inherits_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','insert',
'inherits_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','update',
'inherits_rpa' );
INSERT INTO rpa( role, action, object ) VALUES ( 'snr_data_manager','delete',
'inherits_rpa' );



INSERT INTO ura( usr, role ) VALUES ( 'u0005','house_officer_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0006','house_officer_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0007','house_officer_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0008','house_officer_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0010','house_officer_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0011','snr_house_officer_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0003','snr_house_officer_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0004','snr_house_officer_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0002','specialist_registrar' );
INSERT INTO ura( usr, role ) VALUES ( 'u0012','specialist_registrar' );
INSERT INTO ura( usr, role ) VALUES ( 'u0001','consultant' );
INSERT INTO ura( usr, role ) VALUES ( 'u0009','consultant' );

INSERT INTO ura( usr, role ) VALUES ( 'u0016','student_nurse_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0016','student_nurse_n' );

INSERT INTO ura( usr, role ) VALUES ( 'u0025','staff_nurse_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0026','staff_nurse_d' );

INSERT INTO ura( usr, role ) VALUES ( 'u0015','staff_nurse_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0027','staff_nurse_n' );

INSERT INTO ura( usr, role ) VALUES ( 'u0028','sister_d' );
INSERT INTO ura( usr, role ) VALUES ( 'u0014','sister_d' );

INSERT INTO ura( usr, role ) VALUES ( 'u0020','sister_n' );
INSERT INTO ura( usr, role ) VALUES ( 'u0014','sister_n' );

INSERT INTO ura( usr, role ) VALUES ( 'u0013','specialist_nurse' );
INSERT INTO ura( usr, role ) VALUES ( 'u0029','specialist_nurse' );

INSERT INTO ura( usr, role ) VALUES ( 'u0018','jnr_data_manager' );
INSERT INTO ura( usr, role ) VALUES ( 'u0019','jnr_data_manager' );

INSERT INTO ura( usr, role ) VALUES ( 'u0017','snr_data_manager' );

INSERT INTO ura( usr, role ) VALUES ( 'u0022','receptionist' );
INSERT INTO ura( usr, role ) VALUES ( 'u0021','manager' );

INSERT INTO ura( usr, role ) VALUES ( 'u0016','jnr_data_manager' );
INSERT INTO ura( usr, role ) VALUES ( 'u0022','jnr_data_manager' );


INSERT INTO ura( usr, role ) VALUES ( 'u0005','receptionist' );
INSERT INTO ura( usr, role ) VALUES ( 'u0009','receptionist' );

-- denials
insert into d_rpa( role, action, object ) VALUES ( 'snr_house_officer','select', 'ward' );
insert into d_rpa( role, action, object ) VALUES ( 'sister','select', 'usr' );
insert into d_rpa( role, action, object ) VALUES ( 'staff_nurse','select', 'usr' );

insert into d_rpa( role, action, object ) VALUES ( 'snr_house_officer','select', 'bed' );
insert into d_rpa( role, action, object ) VALUES ( 'house_officer','select', 'usr' );

insert into d_rpa( role, action, object ) VALUES ( 'night_duty','select', 'patient' );
```

```
insert into d_rpa( role, action, object ) VALUES ( 'administrator','update', 'patient' );

insert into d_rpa( role, action, object ) VALUES ( 'nurse','update', 'patient' );
insert into d_rpa( role, action, object ) VALUES ( 'nurse','update', 'ward' );

insert into d_rpa( role, action, object ) VALUES ( 'office_hours','insert', 'ward' );
insert into d_rpa( role, action, object ) VALUES ( 'office_hours','update', 'bed' );

insert into d_rpa( role, action, object ) VALUES ( 'snr_house_officer_d', 'update',
'ae_consultation' );

insert into d_rpa( role, action, object ) VALUES ( 'house_officer_n', 'select', 'diagnosis' );

-- permissions
insert into rpa( role, action, object ) VALUES ( 'day_duty','select', 'usr' );
insert into rpa( role, action, object ) VALUES ( 'doctor','select', 'patient' );
insert into rpa( role, action, object ) VALUES ( 'administrator','insert', 'patient' );
insert into rpa( role, action, object ) VALUES ( 'administrator','insert', 'usr' );
insert into rpa( role, action, object ) VALUES ( 'data_manager','insert', 'usr' );
insert into rpa( role, action, object ) VALUES ( 'administrator','update', 'patient' );
```

Appendix XVI: Hospital Database Data INSERT Statements

```
connect hosp/hosp

INSERT INTO ward( ward_id, type, ward_capacity ) VALUES ( 'ward1', 'Operating', '10' );
INSERT INTO ward( ward_id, type, ward_capacity ) VALUES ( 'ward2', 'Hemotology', '12' );


INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room1O', 'ward1', 'Public',
'4' );
INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room2O', 'ward1', 'Public',
'4' );
INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room3O', 'ward1',
'Private', '2' );
INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room1H', 'ward2', 'Public',
'4' );
INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room2H', 'ward2', 'Public',
'4' );
INSERT INTO room( room_id, ward_id, type, bed_capacity ) VALUES ( 'Room3H', 'ward2',
'Private', '4' );


INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed001', 'Room1O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed002', 'Room1O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed003', 'Room1O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed004', 'Room1O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed005', 'Room2O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed006', 'Room2O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed007', 'Room2O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed008', 'Room2O', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed009', 'Room3O', 'Electric' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed010', 'Room3O', 'Electric' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed011', 'Room1H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed012', 'Room1H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed013', 'Room1H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed014', 'Room1H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed015', 'Room2H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed016', 'Room2H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed017', 'Room2H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed018', 'Room2H', 'Normal' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed019', 'Room3H', 'Electric' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed020', 'Room3H', 'Electric' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed021', 'Room3H', 'Electric' );
INSERT INTO bed( bed_id, room_id, type ) VALUES ( 'Bed022', 'Room3H', 'Electric' );


INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12345', 'Smith', 'John', '33 Oak  Street', TO_DATE( '12/12/197O', 'DD/MM/YYYY' ),
'Bed001' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12354', 'Davies', 'Kenneth', '405 Kingston Road', TO_DATE( '13/03/1980',
'DD/MM/YYYY' ), 'Bed002' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12353', 'Williams', 'Louise', '15 Wellstone Street', TO_DATE( '31/05/1955',
'DD/MM/YYYY' ), 'Bed003' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12352', 'McDonald', 'Ronald', '23 Portobello Road', TO_DATE( '15/06/1977',
'DD/MM/YYYY' ), 'Bed004' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12355', 'Wilkinson', 'Matthew', '15 Touchwood Lane', TO_DATE( '15/02/195O',
'DD/MM/YYYY' ), 'Bed005' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12356', 'Matthewman', 'Wendy', '23a Tisbury Road', TO_DATE( '12/12/199O',
'DD/MM/YYYY' ), 'Bed006' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12357', 'Kenwood', 'Robert', '14 Minster Lane', TO_DATE( '15/09/1966',
'DD/MM/YYYY' ), 'Bed007' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
```

```
VALUES ( '12358', 'Constantine', 'Frederick', '1 The Avenue', TO_DATE( '14/03/1933',
'DD/MM/YYYY' ), 'Bed008' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12347', 'Fowler', 'Robert', '443 Sidney Gardens', TO_DATE( '11/10/1984',
'DD/MM/YYYY' ), 'Bed009' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12359', 'Kelly', 'Yasmin', '14 Crusader Road', TO_DATE( '15/02/1982',
'DD/MM/YYYY' ), 'Bed010' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12349', 'Jones', 'Julia', '12 Oakley Road', TO_DATE( '11/11/1971', 'DD/MM/YYYY' ),
'Bed011' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12346', 'King', 'Steve', '44 Fulham Broadway', TO_DATE( '11/02/1945',
'DD/MM/YYYY' ), 'Bed012' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12350', 'Cole', 'Katherine', '22 Bridge  Road', TO_DATE( '09/08/1950',
'DD/MM/YYYY' ), 'Bed013' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12351', 'Robinson', 'Tim', '11 Horsenden Lane', TO_DATE( '08/07/1960',
'DD/MM/YYYY' ), 'Bed014' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12360', 'James', 'Timothy', '16 Bender Lane', TO_DATE( '01/06/1944', 'DD/MM/YYYY' ),
'Bed015' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12361', 'David', 'Frances', '177 Calder Pass', TO_DATE( '02/07/1966',
'DD/MM/YYYY' ), 'Bed016' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12362', 'Treville', 'Marcus', '103 Stanford Drive', TO_DATE( '22/01/1988',
'DD/MM/YYYY' ), 'Bed017' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12363', 'Mckenzie', 'Angus', '100 Creswood Road', TO_DATE( '21/03/1969',
'DD/MM/YYYY' ), 'Bed018' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12348', 'Philips', 'Cindy', '10 Brentworth Road', TO_DATE( '04/03/1977',
'DD/MM/YYYY' ), 'Bed019' );

INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12364', 'Churchill', 'Winston', '88 Kenwood Drive', TO_DATE( '13/05/1966',
'DD/MM/YYYY' ), 'Bed020' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12365', 'Bhatti', 'Salima', '10 Firewood Lane', TO_DATE( '12/06/1979',
'DD/MM/YYYY' ), 'Bed021' );
INSERT INTO patient( patient_id, last_name, first_name, address, date_of_birth, bed_id )
VALUES ( '12366', 'Dijkstra', 'Ravi', '17 Strongwood Close', TO_DATE( '14/08/1955',
'DD/MM/YYYY' ), 'Bed022' );


INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag001',
'Appendicitis', 'Pain in the iliac fossa on the right side. Loss of appetite and sometimes
vomiting occur,although this is rarely severe. There may be constipation or diarrhoea.' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag002',
'Food Poisoning', 'Nausea,vomiting,diarrhoea and stomach pain' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag003',
'Epilepsy', 'Recurrent fits or seizures' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag004',
'Heart Attack', 'Extreme pain in the left hand side of the chest' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag005',
'Gastroesophageal reflux disease', 'burning pain behind the breastbone,a taste of acid in the
back of the throat or mouth' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag006',
'Pubic Lice', 'Intense itching in the affected area,black powder in underwear,brown eggs on
the hair' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag007',
'Dementia', 'Memory loss is a very common symptom,in particular,short-term memory loss' );
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag008', 'Sun
Allergy', 'Painful skin when outside in the sun' );
```

```
INSERT INTO diagnosis( diagnosis_code, illness_name, usual_symptoms ) VALUES ( 'diag009',
'Eczema', 'Itchy skin' );


INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00001', TO_DATE( '01/02/2006', 'DD/MM/YYYY' ), 'Stomach pains', '12345', 'u0001' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00002', TO_DATE( '24/01/2006', 'DD/MM/YYYY' ), 'Extreme case of diarrhea', '12346',
'u0002' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00003', TO_DATE( '14/12/2005', 'DD/MM/YYYY' ), 'Faints a lot', '12347', 'u0003' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00004', TO_DATE( '03/04/2006', 'DD/MM/YYYY' ), 'Itching on and around groin',
'12348', 'u0004' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00005', TO_DATE( '20/01/2006', 'DD/MM/YYYY' ), 'Forgetting things that he always
used to remember', '12349', 'u0005' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00006', TO_DATE( '26/11/2005', 'DD/MM/YYYY' ), 'Extreme pain in left side',
'12350', 'u0001' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00007', TO_DATE( '30/04/2006', 'DD/MM/YYYY' ), 'Chest pains', '12351', 'u0002' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00008', TO_DATE( '30/04/2006', 'DD/MM/YYYY' ), 'Chest pains', '12352', 'u0007' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00009', TO_DATE( '01/09/2005', 'DD/MM/YYYY' ), 'Extreme pain in left side',
'12353', 'u0008' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00010', TO_DATE( '15/06/2006', 'DD/MM/YYYY' ), 'In pain when steps outside',
'12354', 'u0010' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00011', TO_DATE( '25/07/2005', 'DD/MM/YYYY' ), 'Itching all over body', '12355',
'u0011' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00012', TO_DATE( '20/08/2006', 'DD/MM/YYYY' ), 'Chest pains', '12356', 'u0012' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00013', TO_DATE( '21/09/2005', 'DD/MM/YYYY' ), 'Itching in left lower leg',
'12357', 'u0009' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00014', TO_DATE( '22/03/2006', 'DD/MM/YYYY' ), 'Memory loss', '12358', 'u0001' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00015', TO_DATE( '29/07/2006', 'DD/MM/YYYY' ), 'Regular fits', '12359', 'u0002' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00016', TO_DATE( '12/01/2006', 'DD/MM/YYYY' ), 'Itching on and around groin',
'12360', 'u0003' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00017', TO_DATE( '28/02/2006', 'DD/MM/YYYY' ), 'Stomach pains', '12361', 'u0004' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00018', TO_DATE( '10/03/2006', 'DD/MM/YYYY' ), 'Chest pains', '12362', 'u0012' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00019', TO_DATE( '15/04/2006', 'DD/MM/YYYY' ), 'Occasional fits', '12363',
'u0009' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00020', TO_DATE( '12/05/2006', 'DD/MM/YYYY' ), 'Extreme stomach pains', '12364',
'u0007' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00021', TO_DATE( '05/02/2006', 'DD/MM/YYYY' ), 'Severe memory loss', '12365',
'u0011' );
INSERT INTO ae_consultation( cons_number, cons_date, cons_description, patient_id, doctor_id )
VALUES ( 'c00022', TO_DATE( '16/08/2006', 'DD/MM/YYYY' ), 'Diarrhea', '12366', 'u0010' );


INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00001', 'u0001', 'Patient has been diagnosed with
Appendicitis', 'c00001', 'diag001' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00002', 'u0002', 'Patient has been diagnosed with
food poisoning', 'c00002', 'diag002' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00003', 'u0003', 'Patient has been diagnosed with
epilepsy', 'c00003', 'diag003' );
```

```
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00004', 'u0004', 'Patient has been diagnosed with
pubic lice', 'c00004', 'diag006' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00005', 'u0005', 'Patient has been diagnosed with
dementia', 'c00005', 'diag007' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00006', 'u0001', 'Patient has had a heart attack',
'c00006', 'diag004' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00007', 'u0002', 'patient has been diagnosed with
gastroesophageal reflux disease', 'c00007', 'diag005' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00008', 'u0007', '', 'c00008', 'diag005' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00009', 'u0001', '', 'c00009', 'diag004' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00010', 'u0010', '', 'c00010', 'diag008' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00011', 'u0011', '', 'c00011', 'diag009' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00012', 'u0012', '', 'c00012', 'diag005' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00013', 'u0009', '', 'c00013', 'diag009' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00014', 'u0001', '', 'c00014', 'diag007' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00015', 'u0011', '', 'c00015', 'diag003' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00016', 'u0003', '', 'c00016', 'diag006' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00017', 'u0004', '', 'c00017', 'diag001' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00018', 'u0012', '', 'c00018', 'diag005' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00019', 'u0009', '', 'c00019', 'diag003' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00020', 'u0007', '', 'c00020', 'diag001' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00021', 'u0011', '', 'c00021', 'diag007' );
INSERT INTO patient_diagnosis( patient_diagnosis_number, diagnosing_doctor, diagnosis_desc,
cons_number, diagnosis_code ) VALUES ( 'pd00022', 'u0010', '', 'c00022', 'diag002' );



INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0016', 'ward1' );
INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0016', 'ward2' );

INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0025', 'ward1' );
INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0026', 'ward2' );

INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0015', 'ward1' );
INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0027', 'ward1' );

INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0028', 'ward1' );
INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0014', 'ward2' );

INSERT INTO nurse_ward( usr, ward ) VALUES ( 'u0020', 'ward1' );
```

Appendix XVII: Discussion of Testing and Output

Role Permissions and Denials (rpa and d_rpa)

These produced the same data for each Condition, as expected. The permissions and denials associated with roles do not change according to user activity. The output of **rpa** and **d_rpa** is described by type of role, in the following order:

1. Temporal RBAC Roles: *day_duty* and *night_duty*
2. Job Roles: Data Managers
3. Job Roles: Doctors
4. Job Roles: Nurses
5. Job Roles: Administrators

1 Temporal RBAC Roles: day_duty and night_duty

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'day_duty' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'day_duty' order by role, action, object;

no rows selected
```

*Output 1: **rpa** and **rpa_full** results for day_duty and night_duty.*

No rows were produced, as would be expected (Output 1). *day_duty* has no permissions assigned to it. Additionally, it is not inside any other role, either via a hierarchy or inclusion, so has no implicit role assignments either. It is a container role for all day-duty roles, such as *house_officer_d* and *staff_nurse_d*, so that the temporal context constraints associated with day-duty roles can be applied easily, The role *night_duty* works analogously for night-duty roles.

2 Job Roles: Data Managers

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'data_manager' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'data_manager' order by role, action, object;

no rows selected
```

*Output 2: **rpa** and **rpa_full** results for data_manager.*

Again, *data_manager* is a container role, with no permissions directly assigned (Output 2). All roles assigned to users are within one of *data_manager*, *doctor*, *nurse* and *administrator.* These specify the type of role, but do not have any users directly assigned to them. Permissions could be assigned to these roles as a way of saying "all users of this type can do X", but the model implemented does not use this facility.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'jnr_data_manager' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
jnr_data_manager              |insert          |ae_consultation
jnr_data_manager              |insert          |bed
jnr_data_manager              |insert          |diagnosis
jnr_data_manager              |insert          |patient
jnr_data_manager              |insert          |patient_diagnosis
jnr_data_manager              |insert          |room
jnr_data_manager              |insert          |ward
7 rows selected.
```

*Output 3: **rpa** results for jnr_data_manager.*

```
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'ward' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'room' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'bed' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'patient' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'diagnosis' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'ae_consultation' );
INSERT INTO rpa( role, action, object ) VALUES ( 'jnr_data_manager','insert', 'patient_diagnosis' );
```

*Code 62: **INSERT** statements into **rpa** for jnr_data_manager.*

The role *jnr_data_manager* has permissions directly assigned to it, as shown by Output 3. These can be inferred from the appropriate **INSERT INTO rpa** statements (Code 62).

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'jnr_data_manager' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
jnr_data_manager              |insert          |ae_consultation
jnr_data_manager              |insert          |bed
jnr_data_manager              |insert          |diagnosis
jnr_data_manager              |insert          |patient
jnr_data_manager              |insert          |patient_diagnosis
jnr_data_manager              |insert          |room
jnr_data_manager              |insert          |ward

7 rows selected.
```

*Output 4: **rpa_full** results for jnr_data_manager.*

The query on **rpa_full** produced the same data rows as the **rpa** query (Output 4). This is because although *jnr_data_manager* is contained within *data_manager* via an **is_a** (inclusion) relationship, *data_manager* has no rows in **rpa**.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'snr_data_manager' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
snr_data_manager              |alter           |ae_consultation
...
146 rows selected.

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'snr_data_manager' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
snr_data_manager              |alter           |ae_consultation
...
153 rows selected.
```

*Output 5: Partial **rpa** and **rpa_full** results for snr_data_manager.*

The queries on **rpa** and **rpa_full** for the role *snr_data_manager* produced different results (Output 5). This is because *snr_data_manager* is contained within *jnr_data_manager* via a **d_s** (directly senior) relationship (as well as being contained within *data_manager*). Most rows returned are omitted in Output 5 to save space. The query on **rpa_full** for *snr_data_manager* returns 153 roles: the 146 directly assigned to *snr_data_manager* in **rpa**, and the 7 inherited from *jnr_data_manager*.

The large number of rows involved mean that this is perhaps not the best example. For a better example of static permission inheritance, consider roles of type *doctor*.

3 Job Roles: Doctors

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'doctor' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'doctor' order by role, action, object;

no rows selected
```

Output 6: ***rpa*** *and* ***rpa_full*** *results for* ***doctor****.*

The *doctor* role has no permissions assigned to it (Output 6). However, if it did have any, then they would be inherited directly by all roles contained within it by an **is_a** relationship, which are *house_officer*, *senior_house_officer*, *specialist_registrar* and *consultant*.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'house_officer' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
house_officer                  |select          |ae_consultation
house_officer                  |select          |bed
house_officer                  |select          |diagnosis
house_officer                  |select          |patient
house_officer                  |select          |patient_diagnosis
house_officer                  |select          |room
house_officer                  |select          |usr
house_officer                  |select          |ward

8 rows selected.
```

Output 7: ***rpa*** *results for* *house_officer*.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'house_officer' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
house_officer                  |select          |ae_consultation
house_officer                  |select          |bed
house_officer                  |select          |diagnosis
house_officer                  |select          |patient
house_officer                  |select          |patient_diagnosis
house_officer                  |select          |room
house_officer                  |select          |usr
house_officer                  |select          |ward

8 rows selected.
```

Output 8: ***rpa_full*** *results for* *house_officer*.

For *house_officer*, **rpa** returns the 8 permissions directly assigned to it (Output 7). **rpa_full** returns the same 8 permissions (Output 8), since *house_officer* does not inherit any permissions from elsewhere.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'house_officer_d' order by role, action, object;

no rows selected
```

*Output 9: **rpa** results for house_officer_d.*

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'house_officer_d' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
house_officer_d                |select          |ae_consultation
house_officer_d                |select          |bed
house_officer_d                |select          |diagnosis
house_officer_d                |select          |patient
house_officer_d                |select          |patient_diagnosis
house_officer_d                |select          |room
house_officer_d                |select          |usr
house_officer_d                |select          |ward

8 rows selected.
```

*Output 10: **rpa_full** results for house_officer_d.*

house_officer_d refers to a "house officer on day duty". The role thus inherits permissions from both *house_officer* and *day_duty*. The **rpa** query on *house_officer_d* produces no rows, since no permissions are directly assigned to it (Output 9). However, *rpa_full* retrieves the 8 permissions that *house_officer_d* inherits from *house_officer* (it inherits none from *day_duty*) (Output 10).

house_officer_n refers to a "house officer on night duty", and thus inherits permissions from both *house_officer* and *night_duty*, analogously to *house_officer_d* (data not shown).

The role *snr_house_officer* inherits permissions from house_officer via a **d_s** assignment.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'snr_house_officer' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
snr_house_officer              |select          |ae_consultation
snr_house_officer              |select          |bed
snr_house_officer              |select          |diagnosis
snr_house_officer              |select          |patient
snr_house_officer              |select          |patient_diagnosis
snr_house_officer              |select          |room
snr_house_officer              |select          |usr
snr_house_officer              |select          |ward
snr_house_officer              |update          |ae_consultation
snr_house_officer              |update          |diagnosis
snr_house_officer              |update          |patient_diagnosis

11 rows selected.
```

*Output 11: **rpa_full** results for snr_house_officer.*

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'snr_house_officer' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
snr_house_officer              |update          |ae_consultation
snr_house_officer              |update          |diagnosis
snr_house_officer              |update          |patient_diagnosis
```

*Output 12: **rpa** results for snr_house_officer.*

snr_house_officer has 3 permissions directly assigned to it, as given by the 3 rows returned from **rpa** (Output 11). **rpa_full** returns these 3 rows, plus the 8 representing permissions inherited from *house_officer* (Output 12). Like *house_officer*, *snr_house_officer* also inherits from doctor.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'snr_house_officer_d' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'snr_house_officer_d' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
snr_house_officer_d           |select          |ae_consultation
...

11 rows selected.
```

Output 13: ***rpa*** *and* ***rpa_full*** *results for* *snr_house_officer_d*.

snr_house_officer_d inherits from *snr_house_officer* as *house_officer_d* inherits from *house_officer*, and again has no permissions directly assigned to it (Output 13). The rows returned by **rpa_full** for *snr_house_officer_d* are not all shown in Output 13, since they are exactly the same as the ones returned for *snr_house_officer*.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'specialist_registrar' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
specialist_registrar          |insert          |patient_diagnosis
```

Output 14: ***rpa*** *results for* *specialist_registrar*.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'specialist_registrar' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
specialist_registrar          |insert          |patient_diagnosis
specialist_registrar          |select          |ae_consultation
specialist_registrar          |select          |bed
specialist_registrar          |select          |diagnosis
specialist_registrar          |select          |patient
specialist_registrar          |select          |patient_diagnosis
specialist_registrar          |select          |room
specialist_registrar          |select          |usr
specialist_registrar          |select          |ward
specialist_registrar          |update          |ae_consultation
specialist_registrar          |update          |diagnosis
specialist_registrar          |update          |patient_diagnosis

12 rows selected.
```

Output 15: ***rpa_full*** *results for* *specialist_registrar*.

specialist_registrar inherits from *snr_house_officer*, and also has 1 permission assigned directly. Thus **rpa** retrieves 1 row (Output 14), and **rpa_full** retrieves 1+11=12 rows. (Output 15) Unlike *house_officer* and *senior_ house_officer*, this role has no *day_duty* or *night_duty* divisions.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'consultant' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
consultant                    |insert          |ae_consultation
```

*Output 16: **rpa** results for `consultant`.*

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'consultant' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
consultant                    |insert          |ae_consultation
consultant                    |insert          |patient_diagnosis
consultant                    |select          |ae_consultation
consultant                    |select          |bed
consultant                    |select          |diagnosis
consultant                    |select          |patient
consultant                    |select          |patient_diagnosis
consultant                    |select          |room
consultant                    |select          |usr
consultant                    |select          |ward
consultant                    |update          |ae_consultation
consultant                    |update          |diagnosis
consultant                    |update          |patient_diagnosis

13 rows selected.
```

*Output 17: **rpa_full** results for `consultant`.*

`consultant` inherits from `specialist_registrar`, and has one permission directly assigned, as again indicated by the rows retrived by **rpa** (Output 16) and **rpa_full** (Output 17).

4 Job Roles: Nurses

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'student_nurse' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
student_nurse                 |select          |bed
student_nurse                 |select          |patient
student_nurse                 |select          |room
student_nurse                 |select          |usr
student_nurse                 |select          |ward
```

*Output 18: **rpa** results for `student_nurse`.*

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'student_nurse' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
student_nurse                 |select          |bed
student_nurse                 |select          |patient
student_nurse                 |select          |room
student_nurse                 |select          |usr
student_nurse                 |select          |ward
```

*Output 19: **rpa_full** results for `student_nurse`.*

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'nurse' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'nurse' order by role, action, object;

no rows selected
```

Output 20: ***rpa*** *and* ***rpa_full*** *results for* `nurse`.

`student_nurse` has 5 permissions directly assigned (Output 18 and Output 19). Although it inherits directly from `nurse`, this has no permissions assigned to it (Output 20). `student_nurse_d` and `student_nurse_n` inherit directly from `student_nurse` via is_a relationships. They also respectively inherit from `day_duty` and `night_duty` (data not shown).

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'staff_nurse' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
staff_nurse                    |select          |ae_consultation
staff_nurse                    |select          |diagnosis
staff_nurse                    |select          |patient_diagnosis
staff_nurse                    |select          |usr
staff_nurse                    |update          |patient
```

Output 21: ***rpa*** *results for* `staff_nurse`.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'staff_nurse' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
staff_nurse                    |select          |ae_consultation
staff_nurse                    |select          |bed
staff_nurse                    |select          |diagnosis
staff_nurse                    |select          |patient
staff_nurse                    |select          |patient_diagnosis
staff_nurse                    |select          |room
staff_nurse                    |select          |usr
staff_nurse                    |select          |usr
staff_nurse                    |select          |ward
staff_nurse                    |update          |patient

10 rows selected.
```

Output 22: ***rpa_full*** *results for* `staff_nurse`.

As shown in Output 21 and Output 22, `staff_nurse` inherits directly from `student_nurse` (as well as from `nurse`). Day-duty and night-duty roles `staff_nurse_d` and `staff_nurse_n` also exist (not shown). Note that because **usr** is defined as selectable for both `student_nurse` and `staff_nurse`, the query on **rpa_full** in Output 22 displays it twice (once for `student_nurse`, and once for `staff_nurse`). Using **select distinct** would prevent this duplication.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'sister' order by role, action, object;

Role                           |Action          |Object
-------------------------------|----------------|-------------------------------
sister                         |update          |patient_diagnosis
```

Output 23: ***rpa*** *results for* `sister`.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'sister' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
sister                        |select          |ae_consultation
sister                        |select          |bed
sister                        |select          |diagnosis
sister                        |select          |patient
sister                        |select          |patient_diagnosis
sister                        |select          |room
sister                        |select          |usr
sister                        |select          |usr
sister                        |select          |ward
sister                        |update          |patient
sister                        |update          |patient_diagnosis

11 rows selected.
```

Output 24: ***rpa_full*** *results for* `sister`.

`sister` inherits directly from `staff_nurse` (Output 23 and Output 24). Day-duty and night-duty roles `sister_d` and `sister_n` also exist (not shown).

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'specialist_nurse' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
specialist_nurse              |insert          |diagnosis
specialist_nurse              |select          |ae_consultation
specialist_nurse              |select          |bed
specialist_nurse              |select          |diagnosis
specialist_nurse              |select          |patient
specialist_nurse              |select          |patient_diagnosis
specialist_nurse              |select          |room
specialist_nurse              |select          |usr
specialist_nurse              |select          |usr
specialist_nurse              |select          |ward
specialist_nurse              |update          |ae_consultation
specialist_nurse              |update          |diagnosis
specialist_nurse              |update          |patient
specialist_nurse              |update          |patient_diagnosis

14 rows selected.
```

Output 25: ***rpa_full*** *results for* `specialist_nurse`.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'specialist_nurse' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
specialist_nurse              |insert          |diagnosis
specialist_nurse              |update          |ae_consultation
specialist_nurse              |update          |diagnosis
```

Output 26: ***rpa*** *results for* `specialist_nurse`.

Output 25 and Output 26 show how `specialist_nurse` inherits directly from `sister`. There are no day-duty or night-duty roles for `specialist_nurse`.

5 Job Roles: Administrators

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'receptionist' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
receptionist                  |select          |patient
```

Output 27: ***rpa*** *results for* `receptionist`.

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'receptionist' order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
receptionist                  |select          |patient
```

Output 28: ***rpa_full*** *results for* `receptionist`.

```
SQL> select role "Role", action "Action", object "Object" from rpa
where role = 'administrator' order by role, action, object;

no rows selected

SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'administrator'
order by role, action, object;

no rows selected
```

Output 29: ***rpa*** *and* ***rpa_full*** *results for* `administrator`.

Output 27 and Output 28 show the **rpa** and **rpa_full** results for *receptionist*. This is the most junior role in the *administrator* hierarchy, inheriting only from the (empty) role *administrator* (Output 29).

```
SQL> select role "Role", action "Action", object "Object" from rpa_full
where role = 'manager'
order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
manager                       |insert          |patient
manager                       |update          |patient
```

Output 30: ***rpa*** *results for* `manager`.

```
SQL> select role "Role", action "Action", object "Object"
from rpa_full where role = 'manager'
order by role, action, object;

Role                          |Action          |Object
------------------------------|----------------|------------------------------
manager                       |insert          |patient
manager                       |select          |ae_consultation
manager                       |select          |ae_consultation
manager                       |select          |diagnosis
manager                       |select          |patient
manager                       |select          |patient_diagnosis
manager                       |select          |usr
manager                       |update          |patient
manager                       |update          |patient_diagnosis

9 rows selected.
```

Output 31: ***rpa_full*** *results for* `manager`.

```
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'consultant','house_officer','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'specialist_nurse','student_nurse','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'snr_data_manager','jnr_data_manager','_','_' );
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','receptionist','_','_' );
```

Code 63: Some **inherits_rpa_path statements** *that apply to role* manager.

manager has two permissions directly assigned to it (Output 30). However, its inheritance situation is complex. *manager* inherits not only from *receptionist*, but also from the most senior roles of the other hierarchies, namely *snr_data_manager*, *consultant* and *specialist_nurse*. However, it does not inherit all of the permissions of these roles, due to path inheritance rules defined in Code 63. See Output 31.

```
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','house_officer','select','ae_consultation');
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','staff_nurse','_','patient_diagnosis');
INSERT INTO inherits_rpa_path( senior_role,junior_role,action,object ) VALUES
( 'manager','staff_nurse','select','_');
```

Code 64: Further **inherits_rpa_path statements** *that apply to role* manager.

These statements mean that permissions are inherited from *house_officer* as far as consultant; from *student_nurse* up to *specialist_nurse*; from *jnr_data_manager* up to *snr_data_manager*, and from *receptionist* up to *manager*. Therefore, the role *manager* inherits permissions only from *receptionist*, and does not inherit from any permissions from the other hierarchies. However, exceptions to this are also defined in Code 64.

Thus, the role *manager* inherits **select** permission on **ae_consultation** from the role *house_officer*. It also inherits all permissions related to the table **patient_diagnosis**, as well as all **select** permissions, from *staff_nurse*. These, together with the roles directly assigned and inherited from *receptionist*, yield the 9 rows (8 unique) returned by **rpa_full** for *manager*.

Queries on **d_rpa** and **d_rpa_full** returned no rows (not shown), because no denials were assigned in this model.

Static User Permissions and Authorizations (permittable, authorizable, permitted and authorized)

The results of these tests are given in terms of the Conditions in 3.4.1, with output explained only for some roles, rather than for all roles, to avoid repetition.

No Users Activated

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'nurse'
ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorizable where role = 'nurse'
ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from permitted where role = 'nurse'
ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorized where role = 'nurse'
ORDER BY usr, object, action;

no rows selected
```

Output 32: ***permittable****,* ***authorizable****,* ***permitted*** *and* ***authorized*** *results for* `nurse`.

As explained previously, `nurse` is a container role. Since it has neither users nor permissions assigned to it, queries on **permission** and **authorization** views for it return no rows (Output 32).

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'student_nurse' ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorizable where role = 'student_nurse' ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from permitted where role = 'student_nurse' ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorized where role = 'student_nurse' ORDER BY usr, object, action;

no rows selected
```

Output 33: ***permittable****,* ***authorizable****,* ***permitted*** *and* ***authorized*** *results for* `student_nurse`.

Again, no rows are returned. This is because although permissions are assigned to `student_nurse`, no users are directly assigned (Output 33).

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'student_nurse_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016           |bed                            |select          |student_nurse_d
u0016           |patient                        |select          |student_nurse_d
u0016           |room                           |select          |student_nurse_d
u0016           |usr                            |select          |student_nurse_d
u0016           |ward                           |select          |student_nurse_d
```

Output 34: ***permittable*** *results for* `student_nurse_d`.

permittable returns 5 rows (Output 34). This is because 1 user (**u0016**) is assigned to the role *student_nurse_d*, which has 5 permissions assigned to it: 1×5=5.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from authorizable where role = 'student_nurse_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016            |bed                            |select          |student_nurse_d
u0016            |patient                        |select          |student_nurse_d
u0016            |room                           |select          |student_nurse_d
u0016            |usr                            |select          |student_nurse_d
u0016            |ward                           |select          |student_nurse_d
```

Output 35: ***authorizable*** *results for* *student_nurse_d*.

authorizable returns the same rows as **permittable** (Output 35). This is the case throughout the test, because no denials are assigned.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permitted where role = 'student_nurse_d' ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorized where role = 'student_nurse_d' ORDER BY usr, object, action;

no rows selected
```

Output 36: ***permitted*** *and* ***authorized*** *results for* *student_nurse_d*.

permitted and **authorized** return no rows for this role (Output 36). This is because the user is not active. This is the case throughout this part of the test, because no users are yet active.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'student_nurse_n'
ORDER BY usr, object, action;
User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016            |bed                            |select          |student_nurse_n
u0016            |patient                        |select          |student_nurse_n
u0016            |room                           |select          |student_nurse_n
u0016            |usr                            |select          |student_nurse_n
u0016            |ward                           |select          |student_nurse_n
```

Output 37: ***permittable*** *results for* *student_nurse_n*.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from authorizable where role = 'student_nurse_n'
ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016            |bed                            |select          |student_nurse_n
u0016            |patient                        |select          |student_nurse_n
u0016            |room                           |select          |student_nurse_n
u0016            |usr                            |select          |student_nurse_n
u0016            |ward                           |select          |student_nurse_n
```

Output 38: ***authorizable*** *results for* *student_nurse_n*.

permittable (Output 37) and **authorizable** (Output 38) return the same rows for role *student_nurse_n* as for *student_nurse_d*, because the same user (**u0016**) is assigned to both.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'staff_nurse_d'
ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0025            |ae_consultation                |select          |staff_nurse_d
u0025            |bed                            |select          |staff_nurse_d
u0025            |diagnosis                      |select          |staff_nurse_d
u0025            |patient                        |select          |staff_nurse_d
u0025            |patient                        |update          |staff_nurse_d
u0025            |patient_diagnosis              |select          |staff_nurse_d
u0025            |room                           |select          |staff_nurse_d
u0025            |usr                            |select          |staff_nurse_d
u0025            |ward                           |select          |staff_nurse_d
u0026            |ae_consultation                |select          |staff_nurse_d
u0026            |bed                            |select          |staff_nurse_d
u0026            |diagnosis                      |select          |staff_nurse_d
u0026            |patient                        |select          |staff_nurse_d
u0026            |patient                        |update          |staff_nurse_d
u0026            |patient_diagnosis              |select          |staff_nurse_d
u0026            |room                           |select          |staff_nurse_d
u0026            |usr                            |select          |staff_nurse_d
u0026            |ward                           |select          |staff_nurse_d

18 rows selected.
```

Output 39: ***permittable*** *results for* `staff_nurse_d`.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'staff_nurse_n' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0015            |ae_consultation                |select          |staff_nurse_n
u0015            |bed                            |select          |staff_nurse_n
u0015            |diagnosis                      |select          |staff_nurse_n
u0015            |patient                        |select          |staff_nurse_n
u0015            |patient                        |update          |staff_nurse_n
u0015            |patient_diagnosis              |select          |staff_nurse_n
u0015            |room                           |select          |staff_nurse_n
u0015            |usr                            |select          |staff_nurse_n
u0015            |ward                           |select          |staff_nurse_n
u0027            |ae_consultation                |select          |staff_nurse_n
u0027            |bed                            |select          |staff_nurse_n
u0027            |diagnosis                      |select          |staff_nurse_n
u0027            |patient                        |select          |staff_nurse_n
u0027            |patient                        |update          |staff_nurse_n
u0027            |patient_diagnosis              |select          |staff_nurse_n
u0027            |room                           |select          |staff_nurse_n
u0027            |usr                            |select          |staff_nurse_n
u0027            |ward                           |select          |staff_nurse_n

18 rows selected.
```

Output 40: ***permittable*** *results for* `staff_nurse_n`.

As with `student_nurse`, `permittable` for `staff_nurse` returns no rows. However, for `staff_nurse_d` (Output 39), it returns 9×2=18 rows: 2 users (**u0025** and **u0026**) are assigned to this role, which (as shown earlier) has 9 permissions associated with it (remembering that 2 of the 10 rows were the same). Again 2×9=18 rows are returned, for the 2 users (**u0015** and **u0027**) assigned to `student_nurse_n` (Output 40).

```
SQL> select usr "User", object "Object", action "Action", role "Role" from authorizable where role = 'sister_d'
ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0014            |ae_consultation                |select          |sister_d
u0014            |bed                            |select          |sister_d
u0014            |diagnosis                      |select          |sister_d
u0014            |patient                        |select          |sister_d
u0014            |patient                        |update          |sister_d
u0014            |patient_diagnosis              |select          |sister_d
u0014            |patient_diagnosis              |update          |sister_d
u0014            |room                           |select          |sister_d
u0014            |usr                            |select          |sister_d
u0014            |ward                           |select          |sister_d
u0028            |ae_consultation                |select          |sister_d
u0028            |bed                            |select          |sister_d
u0028            |diagnosis                      |select          |sister_d
u0028            |patient                        |select          |sister_d
u0028            |patient                        |update          |sister_d
u0028            |patient_diagnosis              |select          |sister_d
u0028            |patient_diagnosis              |update          |sister_d
u0028            |room                           |select          |sister_d
u0028            |usr                            |select          |sister_d
u0028            |ward                           |select          |sister_d

20 rows selected.
```

*Output 41: **permittable** results for sister_d.*

permittable on *sister_d* (Output 41) and *sister_n* (not shown) each yield 2×10=20 rows (as before, 11 rows were returned in **rpa_full**, but only 10 were unique).

permittable on *specialist_nurse* (not shown) returns 2×13=26 rows.

The expected results were obtained for other roles (see Table 31).

Some Users Activated

The queries on **permittable** and **authorizable** produced the same results as for when no users were activated. This is as expected. However, **permitted** and **authorized** returned some rows, relating to users who had been activated. This is elaborated in further detail in Static User Permissions and Authorizations (permittable, authorizable, permitted and authorized) (page 295), since the principle is the same: **permitted** and **authorized** only return rows for active users.

All Users Activated

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permittable where role = 'student_nurse_n' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016            |bed                            |select          |student_nurse_n
u0016            |patient                        |select          |student_nurse_n
u0016            |room                           |select          |student_nurse_n
u0016            |usr                            |select          |student_nurse_n
u0016            |ward                           |select          |student_nurse_n
```

*Output 42: **permittable** results for student_nurse_n.*

Since all users were active, **permitted** and **authorized** mostly, but not always, produced the same results as **permittable** and **authorizable**. Where the results were different, this was because some users were defined with more than one role in **ura**; however, a user can only be active in one role at a time. Where this is the case, **permittable** and **authorizable** returned all permissions and authorizations that the user has in whatever role

the user is defined as, while **permitted** and **authorized** returned only those relating to the role for which the user is active.

For example, consider the rows returned for the role *student_nurse_n*.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from authorizable where role = 'student_nurse_n' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0016            |bed                            |select          |student_nurse_n
u0016            |patient                        |select          |student_nurse_n
u0016            |room                           |select          |student_nurse_n
u0016            |usr                            |select          |student_nurse_n
u0016            |ward                           |select          |student_nurse_n
```

Output 43: ***authorizable*** *results for* `student_nurse_n`.

```
SQL> select usr "User", object "Object", action "Action", role "Role"
from permitted where role = 'student_nurse_n' ORDER BY usr, object, action;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role"
from authorized where role = 'student_nurse_n' ORDER BY usr, object, action;

no rows selected
```

Output 44: ***permitted*** *and* ***authorized*** *results for* `student_nurse_n`

permittable (Output 42) and **authorizable** (Output 43) return the rows relating to the user, **u0016**, defined as *student_nurse_n*. However, `u0016` is not active here as *student_nurse_n*, but as the other role for which it is defined, *student_nurse_d*. Since no user is active as *student_nurse_n*, **permitted** and **authorized** both return empty sets (Output 44). As before, the equivalent dynamic (**_cc**) queries return no rows, due to the constraint requiring a *sister* or *staff_nurse* to have been logged on for 2 hours not being fulfilled.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from permittable
where role = 'receptionist' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0005            |patient                        |select          |receptionist
u0009            |patient                        |select          |receptionist
u0022            |patient                        |select          |receptionist
```

Output 45: ***permittable*** *results for* `receptionist`.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from authorizable
where role = 'receptionist' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0005            |patient                        |select          |receptionist
u0009            |patient                        |select          |receptionist
u0022            |patient                        |select          |receptionist
```

Output 46: ***authorizable*** *results for* `receptionist`.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from permitted
where role = 'receptionist' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0022            |patient                        |select          |receptionist
```

Output 47: ***permitted*** *results for* `receptionist`.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from authorized
where role = 'receptionist' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0022            |patient                        |select          |receptionist
```

Output 48: ***authorized*** *results for* `receptionist`.

Similar behaviour can be seen from the role `receptionist`, where 3 users are defined as having this role, but only one of these is active in it (Output 45–48).

Some Users Deactivated

```
SQL> select usr "User", object "Object", action "Action", role "Role" from permittable
where role = 'house_officer_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0005            |ae_consultation                |select          |house_officer_d
u0005            |bed                            |select          |house_officer_d
u0005            |diagnosis                      |select          |house_officer_d
u0005            |patient                        |select          |house_officer_d
u0005            |patient_diagnosis              |select          |house_officer_d
u0005            |room                           |select          |house_officer_d
u0005            |usr                            |select          |house_officer_d
u0005            |ward                           |select          |house_officer_d
u0007            |ae_consultation                |select          |house_officer_d
u0007            |bed                            |select          |house_officer_d
u0007            |diagnosis                      |select          |house_officer_d
u0007            |patient                        |select          |house_officer_d
u0007            |patient_diagnosis              |select          |house_officer_d
u0007            |room                           |select          |house_officer_d
u0007            |usr                            |select          |house_officer_d
u0007            |ward                           |select          |house_officer_d

16 rows selected.
```

Output 49: ***permittable*** *results for* `house_officer_d`.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from authorizable
where role = 'house_officer_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0005           |ae_consultation                |select          |house_officer_d
u0005           |bed                            |select          |house_officer_d
u0005           |diagnosis                      |select          |house_officer_d
u0005           |patient                        |select          |house_officer_d
u0005           |patient_diagnosis              |select          |house_officer_d
u0005           |room                           |select          |house_officer_d
u0005           |usr                            |select          |house_officer_d
u0005           |ward                           |select          |house_officer_d
u0007           |ae_consultation                |select          |house_officer_d
u0007           |bed                            |select          |house_officer_d
u0007           |diagnosis                      |select          |house_officer_d
u0007           |patient                        |select          |house_officer_d
u0007           |patient_diagnosis              |select          |house_officer_d
u0007           |room                           |select          |house_officer_d
u0007           |usr                            |select          |house_officer_d
u0007           |ward                           |select          |house_officer_d

16 rows selected.
```

Output 50: ***authorizable*** *results for* *house_officer_d*.

For role *house_officer_d*, **permittable** and **authorizable** output rows relating to both users defined for this role, namely **u0005** and **u0007** (Output 49–50).

```
SQL> select usr "User", object "Object", action "Action", role "Role" from permitted
where role = 'house_officer_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0007           |ae_consultation                |select          |house_officer_d
u0007           |bed                            |select          |house_officer_d
u0007           |diagnosis                      |select          |house_officer_d
u0007           |patient                        |select          |house_officer_d
u0007           |patient_diagnosis              |select          |house_officer_d
u0007           |room                           |select          |house_officer_d
u0007           |usr                            |select          |house_officer_d
u0007           |ward                           |select          |house_officer_d

8 rows selected.
```

Output 51: ***permitted*** *results for* *house_officer_d*.

```
SQL> select usr "User", object "Object", action "Action", role "Role" from authorized
where role = 'house_officer_d' ORDER BY usr, object, action;

User            |Object                         |Action          |Role
----------------|-------------------------------|----------------|-------------------------------
u0007           |ae_consultation                |select          |house_officer_d
u0007           |bed                            |select          |house_officer_d
u0007           |diagnosis                      |select          |house_officer_d
u0007           |patient                        |select          |house_officer_d
u0007           |patient_diagnosis              |select          |house_officer_d
u0007           |room                           |select          |house_officer_d
u0007           |usr                            |select          |house_officer_d
u0007           |ward                           |select          |house_officer_d

8 rows selected.
```

Output 52: ***authorized*** *results for* *house_officer_d*.

However, since **u0005** is no longer active, **permitted** and **authorized** only return rows relating to **u0007** (Output 51–52).

Dynamic User Permissions and Authorizations (permittable_cc, authorizable_cc, permitted_cc and authorized_cc)

No Users Activated

```
SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row" from permittable_cc
where role = 'nurse' ORDER BY usr, object, action, row_id;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row" from authorizable_cc
where role = 'nurse' ORDER BY usr, object, action, row_id;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row" from permitted_cc
where role = 'nurse' ORDER BY usr, object, action, row_id;

no rows selected

SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row" from authorized_cc
where role = 'nurse' ORDER BY usr, object, action, row_id;

no rows selected
```

Output 53: ***permittable_cc****,* ***authorizable_cc****,* ***permitted_cc*** *and* ***authorized_cc*** *results for* `nurse`.

As before, the queries on the container role *`nurse`* retrieved no rows (Output 53). The same is true of the role *`student_nurse`* (not shown). Furthermore, all queries on both *`student_nurse_d`* and *`student_nurse_n`* also return empty sets (not shown), although queries on static permission views for both these roles returned rows. The dynamic constraints that apply to roles contained by *`student_nurse`* mean that users with such roles have no access rights at this time: users with role *`student_nurse`* have no permissions unless a user of a role contained within *`staff_nurse`* has been logged on for at least 2 hours.

```
SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row"
from permittable_cc where role = 'staff_nurse_d' ORDER BY usr, object, action, row_id;

User            |Object                         |Action          |Role                           |Row
----------------|-------------------------------|----------------|-------------------------------|-------
u0025            |ae_consultation                |select          |staff_nurse_d                  |c00001
...
u0025            |patient                        |select          |staff_nurse_d                  |12345
u0025            |patient                        |select          |staff_nurse_d                  |12347
u0025            |patient                        |select          |staff_nurse_d                  |12352
u0025            |patient                        |select          |staff_nurse_d                  |12353
u0025            |patient                        |select          |staff_nurse_d                  |12354
u0025            |patient                        |select          |staff_nurse_d                  |12355
u0025            |patient                        |select          |staff_nurse_d                  |12356
u0025            |patient                        |select          |staff_nurse_d                  |12357
u0025            |patient                        |select          |staff_nurse_d                  |12358
u0025            |patient                        |select          |staff_nurse_d                  |12359
u0025            |patient                        |update          |staff_nurse_d                  |12345
u0025            |patient                        |update          |staff_nurse_d                  |12347
u0025            |patient                        |update          |staff_nurse_d                  |12352
u0025            |patient                        |update          |staff_nurse_d                  |12353
u0025            |patient                        |update          |staff_nurse_d                  |12354
u0025            |patient                        |update          |staff_nurse_d                  |12355
u0025            |patient                        |update          |staff_nurse_d                  |12356
u0025            |patient                        |update          |staff_nurse_d                  |12357
u0025            |patient                        |update          |staff_nurse_d                  |12358
u0025            |patient                        |update          |staff_nurse_d                  |12359
...
u0026            |patient                        |select          |staff_nurse_d                  |12346
u0026            |patient                        |select          |staff_nurse_d                  |12348
u0026            |patient                        |select          |staff_nurse_d                  |12349
u0026            |patient                        |select          |staff_nurse_d                  |12350
u0026            |patient                        |select          |staff_nurse_d                  |12351
u0026            |patient                        |select          |staff_nurse_d                  |12360
u0026            |patient                        |select          |staff_nurse_d                  |12361
u0026            |patient                        |select          |staff_nurse_d                  |12362
u0026            |patient                        |select          |staff_nurse_d                  |12363
u0026            |patient                        |select          |staff_nurse_d                  |12364
u0026            |patient                        |select          |staff_nurse_d                  |12365
u0026            |patient                        |select          |staff_nurse_d                  |12366
u0026            |patient                        |update          |staff_nurse_d                  |12346
u0026            |patient                        |update          |staff_nurse_d                  |12348
u0026            |patient                        |update          |staff_nurse_d                  |12349
u0026            |patient                        |update          |staff_nurse_d                  |12350
u0026            |patient                        |update          |staff_nurse_d                  |12351
u0026            |patient                        |update          |staff_nurse_d                  |12360
u0026            |patient                        |update          |staff_nurse_d                  |12361
u0026            |patient                        |update          |staff_nurse_d                  |12362
u0026            |patient                        |update          |staff_nurse_d                  |12363
u0026            |patient                        |update          |staff_nurse_d                  |12364
u0026            |patient                        |update          |staff_nurse_d                  |12365
u0026            |patient                        |update          |staff_nurse_d                  |12366...
```

Output 54: Partial ***permittable_cc*** *results for* *staff_nurse_d*.

The queries on container role *staff_nurse* also return an empty set, as expected. However, 210 rows are returned by **permittable_cc** for *staff_nurse_d*. This is because a row is returned for every user, action and database tuple for which access is granted. For most tables, a row representing every row in the table is returned, because there are no row-level context constraints. However, **permittable_cc** returns the following rows for the table **patient** (Output 54).The same users, **u0025** and **u0026**, occur as in the static queries on *staff_nurse_d*. However, while the table **patient** contains 21 rows, it is clear from the above query results that specific users only have access to some of these rows in the table. The **Row** column holds the primary keys of rows in a table to which access is granted. **permittable_cc** returns 11 rows each for **select** and **update** (the same rows for each action) for user **u0025**. It also returns 11 rows for each of these two actions for **u0026**. However, the rows are different from those returned for **u0025**, as is shown by the values of **Row**. This is because of a context constraint limiting access by users of role *student_nurse* and *staff_nurse* to data of patients in wards to which they are assigned: these users are assigned to different wards.

permitted_cc and **authorized_cc**, as before, returned no rows anywhere because no users have been activated.

All queries on **staff_nurse_n** returned empty sets. This is because the test was run during the day, outside the hours during which users in roles contained by *night_duty* are permitted to access data.

Some Users Activated

The queries on **permittable_cc** and **authorizable_cc** produced the same results as for when no users were activated. This is as expected. However, **permitted_cc** and **authorized_cc** returned some rows in this test run. For example, **permitted_cc** on *staff_nurse_d* returned 103 rows (compared to 0 rows when no users were activated, and 210 rows returned by **permittable_cc**). **permitted_cc** returns all rows relevant to user **u0025**. This is because in this test run, **u0025** was activated, while **u0026**, the other user assigned to *staff_nurse_d*, was not.

All Users Activated

```
SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row"
from authorized_cc where role = 'receptionist' ORDER BY usr, object, action, row_id;

User            |Object                         |Action          |Role                           |Row
----------------|-------------------------------|----------------|-------------------------------|----------------
u0022           |patient                        |select          |receptionist                   |12345
u0022           |patient                        |select          |receptionist                   |12346
u0022           |patient                        |select          |receptionist                   |12347
u0022           |patient                        |select          |receptionist                   |12348
u0022           |patient                        |select          |receptionist                   |12349
u0022           |patient                        |select          |receptionist                   |12350
u0022           |patient                        |select          |receptionist                   |12351
u0022           |patient                        |select          |receptionist                   |12352
u0022           |patient                        |select          |receptionist                   |12353
u0022           |patient                        |select          |receptionist                   |12354
u0022           |patient                        |select          |receptionist                   |12355
u0022           |patient                        |select          |receptionist                   |12356
u0022           |patient                        |select          |receptionist                   |12357
u0022           |patient                        |select          |receptionist                   |12358
u0022           |patient                        |select          |receptionist                   |12359
u0022           |patient                        |select          |receptionist                   |12360
u0022           |patient                        |select          |receptionist                   |12361
u0022           |patient                        |select          |receptionist                   |12362
u0022           |patient                        |select          |receptionist                   |12363
u0022           |patient                        |select          |receptionist                   |12364
u0022           |patient                        |select          |receptionist                   |12365
u0022           |patient                        |select          |receptionist                   |12366
u0022           |patient                        |select          |receptionist                   |12367

23 rows selected.
```

Output 55: ***authorized_cc*** *results for* `receptionist`.

As with static permissions, Since all users were active, **permitted_cc** and **authorized_cc** (Output 55) produced the same results as **permittable_cc** and **authorizable_cc**, except in the case of users defined with more than one role in **ura**.

Enforcement of RBAC in Meta-Data

As before, the test results of this section are described according to Conditions.

No Users Activated

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table"
FROM ALL_TAB_PRIVS WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|------------------------------
HOSP                           |HOSP1_U0001                    |EXECUTE         |SET_CONTEXT
```

Output 56: Privileges granted to ***HOSP1_U0001***.

All attempts to access or manipulate data failed, as expected. Queries on the meta-data also showed that no permissions were granted. Output 56 shows an example for user **u0001**.

This shows that the database user **HOSP_U0001** (corresponding to user **u0001** in the RBAC data) only has permission on one meta-data table, which is necessary for logging in. The user has no permissions on any data tables. The same is true for all users in this test, since no users are active.

Some Users Activated

No data access or manipulation could be performed when logged in as any inactive users. Active users could perform actions that they were authorized to do by the RBAC rules.

Consider user **u0002**, active as role *specialist_registrar*.

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |WARD
HOSP                           |ROOM
HOSP                           |BED
HOSP                           |PATIENT
HOSP                           |DIAGNOSIS
HOSP                           |AE_CONSULTATION
HOSP                           |PATIENT_DIAGNOSIS
HOSP                           |USR
```

Output 57: Tables visible to user ***HOSP1_U0002***.

The **SELECT** statement in Output 57 retrieves all tables that are visible to the user (the **WHERE** clause prevents tables from other irrelevant schemas, especially meta-data, from being returned). Note that if the user has no access privileges on a table, then it does not appear in the results of this query. Effectively, the table does not exist for the user. For inactive users, the query returns an empty set.

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table"
FROM ALL_TAB_PRIVS WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|------------------------------
HOSP                           |HOSP1_U0002                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |WARD
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |ROOM
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |BED
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |DIAGNOSIS
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |USR
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |AE_CONSULTATION
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT_DIAGNOSIS
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |DIAGNOSIS
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |AE_CONSULTATION
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |PATIENT_DIAGNOSIS
HOSP                           |HOSP1_SPECIALIST_REGISTRAR     |INSERT          |PATIENT_DIAGNOSIS

13 rows selected.
```

Output 58: Privileges granted to ***HOSP1_U0002***.

The query in Output 58 returns the table-level *static* permissions granted to a user. Note the **Grantee** column above, indicating the role through which the user obtains a particular access right. The **Grantor** is always HOSP, the database user under which the database was set up. As before, the **WHERE** clause excludes irrelevant privileges, such as those on meta-data.

```
SQL> SELECT * FROM ward;

WARD_ID   |TYPE       |WARD_CAPAC
----------|-----------|----------
ward1     |Operating  |10
ward2     |Hemotology|12
```

*Output 59: **HOSP1_U0002** reads **ward**.*

Since no dynamic constraints apply to this user, **SELECT** queries on the tables named in the query on **sys.all_tables** return all rows, as shown in Output 59 for user **u0002**.

```
SQL> SELECT * FROM nurse_ward;
SELECT * FROM nurse_ward
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

*Output 60: **HOSP1_U0002** fails to access **nurse_ward**..*

However, when the user attempts to **SELECT** from a table to which he does not have access rights, the system behaves as though the table does not exist, as shown in Output 60.

```
SQL> UPDATE patient_diagnosis SET diagnosis_desc = 'Coronary Heart Disease'
WHERE patient_diagnosis_number = 'pd00008';

1 row updated.
```

*Output 61: **HOSP1_U0002** updates **patient_diagnosis**.*

UPDATE and **INSERT** statements that the user has the right to run are performed normally (Output 61).

```
SQL> UPDATE ward SET ward_capacity = 15 WHERE ward_id
= 'ward1';
UPDATE ward SET ward_capacity = 15 WHERE ward_id =
'ward1'
       *
ERROR at line 1:
ORA-01031: insufficient privileges
SQL> INSERT INTO ward (
  2    ward_id,
  3    type,
  4    ward_capacity
  5  ) VALUES (
  6    'ward3',
  7    'Operating',
  8    12
  9  );
INSERT INTO ward (
            *
ERROR at line 1:
ORA-01031: insufficient privileges
```

*Output 62: **HOSP1_U0002** fails to insert into **ward**.*

```
SQL> DELETE FROM ae_consultation WHERE 0 <> 0;
DELETE FROM ae_consultation WHERE 0 <> 0
            *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> DELETE FROM authorized WHERE 0 <> 0;
DELETE FROM authorized WHERE 0 <> 0
            *
ERROR at line 1:
ORA-00942: table or view does not exist
```

*Output 63: **HOSP1_U0002** fails to delete from **ae_consultation** and **authorized**..*

If an attempt is made to **DELETE**, **INSERT** or **UPDATE** a row in a table to which the user does not have the appropriate privilege, but does have some privileges on it, then an *Insufficient Privileges* error is returned (Output 62).

If the user does not have any privileges at all on the table, then as before the table does not exist for the user, as can be shown by the different behaviour of the two **DELETE** statements in Output 63. Note the **WHERE** clause that always fails. Deleting rows during the test run would make the database unusable for testing, so it is not done.

```
SQL> INSERT INTO patient_diagnosis (
  2    patient_diagnosis_number,
  3    diagnosing_doctor,
  4    diagnosis_desc,
  5    cons_number,
  6
diagnosis_code
  7  ) VALUES (
  8    'pd00023',
  9    'u0010',
 10    'Stomach infection',
 11    'c00023',
 12    'diag002'
 13  );
INSERT INTO patient_diagnosis (
*
ERROR at line 1:
ORA-02291: integrity constraint (HOSP.SYS_C009415)
violated - parent key not found
```

Output 64: ***HOSP1_U0002*** *inserts into* ***patient_diagnosis****.*

Indeed, for this test, what matters when attempting to **INSERT**, **UPDATE** or **DELETE** is not whether the transaction was ultimately successful, but whether it could be performed. For example, consider the **INSERT** on the table **patient_diagnosis** in Output 64.

The **INSERT** fails because of an integrity constraint: the value **'c00023'** is a foreign key that does not exist in the referencing table. However, the fact that the statement was executed shows that (as expected) this user has the right to insert rows into the table **patient_diagnosis**.

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |WARD
HOSP                           |ROOM
HOSP                           |BED
HOSP                           |PATIENT
HOSP                           |DIAGNOSIS
HOSP                           |AE_CONSULTATION
HOSP                           |PATIENT_DIAGNOSIS
HOSP                           |USR

8 rows selected.
```

Output 65: Tables visible to user ***HOSP1_U0003****.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|-------------------------------
HOSP                           |HOSP1_U0003                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |WARD
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |ROOM
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |BED
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |DIAGNOSIS
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |USR
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |AE_CONSULTATION
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT_DIAGNOSIS
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |DIAGNOSIS
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |AE_CONSULTATION
HOSP                           |HOSP1_SNR_HOUSE_OFFICER        |UPDATE          |PATIENT_DIAGNOSIS

12 rows selected.
```

*Output 66: Privileges granted to **HOSP1_U0003**.*

```
SQL> SELECT diagnosis_code "Diag_Code", illness_name "Illness_Name", usual_symptoms "Usual_Symptoms" FROM
diagnosis;

no rows selected
```

*Output 67: Table **diagnosis** as seen by **HOSP1_U0003**.*

User* u0003 *is activated here in the role *snr_house_officer_n*. Since this a *night_duty* role, and the test was performed during *day_duty* hours, this user should have no access. The user can see tables, and is shown as having privileges granted. This is because the rows returned by the meta-data views **sys.all_tables** (Output 65) and **sys.all_tab_privs** (Output 66) are determined by static privileges.

Therefore, it looks as if the user can **SELECT** and **UPDATE** data from various tables. Yet, an attempt to read any such table produces an empty set (Output 67).

These results are due to the way dynamic RBAC is handled in Oracle VPD. Dynamic context constraints are implemented internally by adding **WHERE** clauses to statements before they are run. For a user in a *night_duty* role running a query during *day_duty* hours (or vice-versa), the **WHERE** clause always evaluates to **FALSE** (it compares the current time with **SYSDATE**), thus causing an empty set to be returned for all **SELECT** queries. The mechanism is called Fine-Grained Access Control, and was originally introduced in Oracle 8.

```
SQL> UPDATE diagnosis SET usual_symptoms = usual_symptoms || ', with foaming at the mouth.'
WHERE diagnosis_code = 'diag003';

0 rows updated.
```

*Output 68: **HOSP1_U0003** updates **diagnosis**.*

When a user is prevented by a context constraint from performing an **UPDATE** or **DELETE** statement, the database cannot find the row to modify, as shown in Output 68.

A row in **diagnosis** does exist for **diagnosis_code = 'diag003'**. However, when logged in as **u0003** at this time the system cannot see the row, as it cannot see any rows in any table for which the temporal constraint is defined. Therefore, no update is run.

```
SQL> SELECT * FROM ward;

WARD_ID   |TYPE      |WARD_CAPAC
----------|----------|----------
ward1     |Operating |10
ward2     |Hemotology|12
```

*Output 69: **HOSP1_U0004** inserts into **ward**..*

```
SQL> SELECT * FROM room;

ROOM_ID   |WARD_ID   |TYPE      |BED_CAPACI
----------|----------|----------|----------
Room1O    |ward1     |Public    |4
Room2O    |ward1     |Public    |4
Room3O    |ward1     |Private   |2
Room1H    |ward2     |Public    |4
Room2H    |ward2     |Public    |4
Room3H    |ward2     |Private   |4

6 rows selected.
```

*Output 70: **HOSP1_U0004** reads **room**.*

User* u0004 *is active as *snr_house_officer_d*. This is a *day_duty* role, and so the user should have access to the data as the test is run during *day_duty* hours. Queries on all_tables and **sys.all_tab_privs** thus produce similar results to those for **u0003**. However, accesses to data tables are successful, as shown in Output 69–70.

```
SQL> SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM patient;

Patient_ID|Last_Name           |First_Name          |Address                                            |DATE_OF_BI
----------|--------------------|--------------------|---------------------------------------------------|----------
12348     |Philips             |Cindy               |10 Brentworth Road                                 |04-MAR-77
12361     |David               |Frances             |177 Calder Pass                                    |02-JUL-66
```

*Output 71: **HOSP1_U0004** reads **patient**.*

The query on **patient** returns only two rows out of the 22 actually in the table, as shown in Output 71.

```
SQL> select usr "User", object "Object", action "Action", role "Role", row_id "Row" from authorized_cc
where role = 'snr_house_officer_d' ORDER BY usr, object, action, row_id;

User            |Object                          |Action          |Role                          |Row
----------------|--------------------------------|----------------|------------------------------|-----
...
u0004           |patient                         |select          |snr_house_officer_d           |12348
u0004           |patient                         |select          |snr_house_officer_d           |12361
...

138 rows selected.
```

*Output 72: **authorized_cc** results for snr_house_officer_d concerning **patient** and **u0004**.*

Compare this to the rows produced by **authorized_cc** for this user and the **patient** table (Output 72).

```
SQL> UPDATE diagnosis SET usual_symptoms = usual_symptoms || ', with foaming at the mouth.' WHERE diagnosis_code =
'diag003';

1 row updated.

SQL> UPDATE ae_consultation SET cons_description = 'Diarrhea and Vomiting' WHERE cons_number = 'c00022';

1 row updated.

SQL> UPDATE patient_diagnosis SET diagnosis_desc = 'Coronary Heart Disease' WHERE patient_diagnosis_number =
'pd00008';

1 row updated.
```

*Output 73: **HOSP1_U0004** updates **diagnosis**, **ae_consultation** and **patient_diagnosis**.*

Updates on tables for which this is permitted by the static rules were also run. Output 73 shows user **u0004** successfully updating tables **diagnosis**, **ae_consultation** and **patient_diagnosis**, as the user should be able to do when active in role *snr_house_officer_d*.

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |URA
HOSP                           |SENIOR_TO
HOSP                           |INCLUDED_IN
HOSP                           |INHERITS_RPA_PATH
HOSP                           |IS_A
HOSP                           |RPA
HOSP                           |D_RPA
HOSP                           |USR_SESSION
HOSP                           |DSD
HOSP                           |SSD
HOSP                           |WARD
HOSP                           |ROOM
HOSP                           |BED
HOSP                           |PATIENT
HOSP                           |DIAGNOSIS
HOSP                           |AE_CONSULTATION
HOSP                           |PATIENT_DIAGNOSIS
HOSP                           |NURSE_WARD
HOSP                           |USR
HOSP                           |PASSWORD
HOSP                           |ROLE
HOSP                           |D_S

22 rows selected.
```

Output 74: Tables visible to user ***HOSP1_U0017****.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|------------------------------
HOSP                           |HOSP1_U0017                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |WARD
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |ROOM
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |BED
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |PATIENT
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |DIAGNOSIS
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |AE_CONSULTATION
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |PATIENT_DIAGNOSIS
HOSP                           |HOSP1_SNR_DATA_MANAGER         |SELECT          |WARD
HOSP                           |HOSP1_SNR_DATA_MANAGER         |UPDATE          |WARD
HOSP                           |HOSP1_SNR_DATA_MANAGER         |DELETE          |WARD
HOSP                           |HOSP1_SNR_DATA_MANAGER         |ALTER           |WARD
...

154 rows selected.
```

Output 75: Privileges granted to ***HOSP1_U0017****.*

User **u0017** ***is active*** as role *snr_data_manager*, and so has all privileges across all tables (Output 74).

The last row listed in Output 75 is for an **ALTER** privilege, allowing the user to modify the table structure (not tested).

All statements in the test run were executed, and ran on all rows. This includes **DELETE** statements, which in this model can only be run by users active as role *snr_data_manager*. However, the always-**FALSE** **WHERE** clause prevented the **DELETE** statements from having any effect (not shown).

```
SQL> DELETE FROM currently_active WHERE 0 <> 0;
DELETE FROM currently_active WHERE 0 <> 0
            *
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

Output 76: Attempting to delete a view.

Some **DELETE** statements could not be run because they were operating on views, not tables (Output 76).

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|-----------------------------
HOSP                           |HOSP1_U0018                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |WARD
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |ROOM
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |BED
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |PATIENT
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |DIAGNOSIS
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |AE_CONSULTATION
HOSP                           |HOSP1_JNR_DATA_MANAGER         |INSERT          |PATIENT_DIAGNOSIS
```

*Output 77: Tables visible to user **HOSP1_U0018**.*

User* u0018 *is activated in the role *jnr_data_manager*. Users in this role have only **INSERT** privileges on specific tables (think of this role as being a data entry clerk) (Output 77).

This user has no **SELECT** privileges. However, attempts to **SELECT** the tables listed in **sys.all_tables** (not shown) and **sys.all_tab_privs** produce different results from those found previously.

```
SQL> SELECT * FROM ward;
SELECT * FROM ward
              *
ERROR at line 1:
ORA-01031: insufficient privileges
```

*Output 78: **HOSP1_U0018** fails to read **ward**.*

An *insufficient privileges* error is returned instead of the previous *ORA-00942: table or view does not exist*. The table is known to the user (because of an **INSERT** privilege on it), but its contents cannot be viewed (Output 78).

All Users Activated

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |WARD
HOSP                           |ROOM
HOSP                           |BED
HOSP                           |PATIENT
HOSP                           |DIAGNOSIS
HOSP                           |AE_CONSULTATION
HOSP                           |PATIENT_DIAGNOSIS
HOSP                           |USR

8 rows selected.
```

*Output 79: Tables visible to user **HOSP1_U0005**.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|-------------------------------
HOSP                           |HOSP1_U0005                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |WARD
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |ROOM
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |BED
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |DIAGNOSIS
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |USR
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |AE_CONSULTATION
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT_DIAGNOSIS
```

Output 80: Privileges granted to ***HOSP1_U0005****.*

User **u0005** ***is defined as a*** `receptionist`, but is active here as `house_officer_d`. As such, the only privileges given to this user are those of a `house_officer` (Output 79–80).

```
SQL> SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM patient;

Patient_ID|Last_Name           |First_Name          |Address                                            |DATE_OF_BI
----------|--------------------|--------------------|---------------------------------------------------|----------
12349     |Jones               |Julia               |12 Oakley Road                                     |12/12/1979
```

Output 81: ***HOSP1_U0005*** *reads* **patient**.

This can be shown by the query on table **patient** (Output 81) (the output is as it is without formatting commands).

If **u0005** were active as a `receptionist`, then this query should return all 22 rows of the patient table. However, since **u0005** is active as `house_officer_d`, the query returns the one row that satisfies the context constraint attached to the role `house_officer`. User **u0009**, the other inactive `receptionist`, is active as a `consultant`, and so can see all rows of the **patient** table anyway.

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |PATIENT
```

Output 82: Tables visible to user ***HOSP1_U0022****.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|-------------------------------
HOSP                           |HOSP1_U0022                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_RECEPTIONIST             |SELECT          |PATIENT
```

Output 83: Privileges granted to ***HOSP1_U0022****.*

User **u0022,** ***being active as a*** `receptionist`, has only a **SELECT** privilege on the **patient** table (Output 82–83), but with no row-level constraints. [There is, however, a temporal constraint, restricting receptionist users to accessing data during the hours 0900−1700, Monday to Friday; this is defined in the `office_hours` role.]

```
SQL> SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM patient;

Patient_ID|Last_Name           |First_Name          |Address                                            |DATE_OF_BI
----------|--------------------|--------------------|---------------------------------------------------|----------
12345     |Smith               |John                |33 Oak  Street                                     |12/12/1979
12354     |Davies              |Kenneth             |405 Kingston Road                                  |12/12/1979
12353     |Williams            |Louise              |15 Wellstone Street                                |12/12/1979
12352     |McDonald            |Ronald              |23 Portobello Road                                 |12/12/1979
12355     |Wilkinson           |Matthew             |15 Touchwood Lane                                  |12/12/1979
12356     |Matthewman          |Wendy               |23a Tisbury Road                                   |12/12/1979
12357     |Kenwood             |Robert              |14 Minster Lane                                    |12/12/1979
12358     |Constantine         |Frederick           |1 The Avenue                                       |12/12/1979
12347     |Fowler              |Robert              |443 Sidney Gardens                                 |12/12/1979
12359     |Kelly               |Yasmin              |14 Crusader Road                                   |12/12/1979
12349     |Jones               |Julia               |12 Oakley Road                                     |12/12/1979
12346     |King                |Steve               |44 Fulham Broadway                                 |12/12/1979
12350     |Cole                |Katherine           |22 Bridge  Road                                    |12/12/1979
12351     |Robinson            |Tim                 |11 Horsenden Lane                                  |12/12/1979
12360     |James               |Timothy             |16 Bender Lane                                     |12/12/1979
12361     |David               |Frances             |177 Calder Pass                                    |12/12/1979
12362     |Treville            |Marcus              |103 Stanford Drive                                 |12/12/1979
12363     |Mckenzie            |Angus               |100 Creswood Road                                  |12/12/1979
12348     |Philips             |Cindy               |10 Brentworth Road                                 |12/12/1979
12364     |Churchill           |Winston             |88 Kenwood Drive                                   |12/12/1979
12365     |Bhatti              |Salima              |10 Firewood Lane                                   |12/12/1979
12366     |Dijkstra            |Ravi                |17 Strongwood Close                                |12/12/1979
12367     |Christ              |Jesus H.            |The Stables, The Inn, Bethlehem                    |0000-12-25

23 rows selected.
```

*Output 84: **HOSP1_U0022** reads **patient**.*

Therefore, **u0022** can see the whole **patient** table (Output 84). [The final row, **Patient_ID**=**12367**, was added during the test by user **u0017**.]

```
SQL> INSERT INTO bed (
  2    bed_id,
  3    room_id,
  4    type
  5  ) VALUES (
  6    'Bed023',
  7    'Room1G',
  8    'Electric'
  9  );
INSERT INTO bed (
            *
ERROR at line 1:
ORA-00942: table or view does not exist
```

*Output 85: **HOSP1_U0022** fails to insert into **bed** when logged in as receptionist.*

```
SQL> INSERT INTO patient (
  2    patient_id,
  3    last_name,
  4    first_name,
  5    address,
  6    date_of_birth,
  7    bed_id
  8  ) VALUES (
  9    12367,
 10    'Christ',
 11    'Jesus H.',
 12    'The Stables, The Inn, Bethlehem',
 13    '0000-12-25',
 14    'Bed023'
 15  );
INSERT INTO patient (
            *
ERROR at line 1:
ORA-01031: insufficient privileges
```

*Output 86: **HOSP1_U0022** fails to insert into **patient** when logged in as receptionist.*

User **u0022** is also defined in the role *jnr_data_manager*. However, since the user is not active in this role, he has none of the privileges associated with it (Output 85–86).

If **u0022** were active as *jnr_data_manager*, these **INSERT** statements would be run. Note, again, the difference in behaviour between the two statements. This user has no access at all to table **bed**, so the session behaves as if this table does not exist at all. However, the session does know about the table **patient**, due to the user's **SELECT** privilege. Therefore, the *insufficient privileges* error results.

Conversely, **u0022** would not be able to **SELECT** the patient table if active as *jnr_data_manager*.

The results for **u0021** show a shortcoming in Oracle implementation of this DRBAC: this user (active as manager) can do everything through permissions inherited from the *snr_data_manager*. This is not supposed to happen due to **inherits_rpa_path** definitions.

Some Users Deactivated

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

no rows selected
```

Output 87: No tables visible to user ***HOSP1_U0005*** *after deactivation.*

```
SQL> SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM patient;
SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address", date_of_birth
FROM patient

*
ERROR at line 1:
ORA-00942: table or view does not exist
```

Output 88: User ***HOSP1_U0005*** *has no access to table* **patient** *after deactivation.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|------------------------------
HOSP                           |HOSP1_U0005                    |EXECUTE         |SET_CONTEXT
```

Output 89: No privileges granted to ***HOSP1_U0005*** *after deactivation.*

```
SQL> SELECT * FROM bed;
SELECT * FROM bed
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Output 90: User ***HOSP1_U0005*** *has no access to table* **bed** *after deactivation.*

User **u0005** is one of the users that is deactivated. Therefore, this user now has no access privileges, and cannot see any tables (Output 87–90).

```
SQL> SELECT owner, table_name FROM sys.all_tables WHERE owner = 'HOSP';

OWNER                          |TABLE_NAME
-------------------------------|-------------------------------
HOSP                           |WARD
HOSP                           |ROOM
HOSP                           |BED
HOSP                           |PATIENT
HOSP                           |DIAGNOSIS
HOSP                           |AE_CONSULTATION
HOSP                           |PATIENT_DIAGNOSIS
HOSP                           |USR

8 rows selected.
```

*Output 91: Tables visible to user **HOSP1_U0007**.*

```
SQL> SELECT grantor "Grantor", grantee "Grantee", privilege "Privilege", table_name "Table" FROM ALL_TAB_PRIVS
WHERE TABLE_SCHEMA = 'HOSP';

Grantor                        |Grantee                        |Privilege       |Table
-------------------------------|-------------------------------|----------------|-------------------------------
HOSP                           |HOSP1_U0007                    |EXECUTE         |SET_CONTEXT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |WARD
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |ROOM
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |BED
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |DIAGNOSIS
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |USR
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |AE_CONSULTATION
HOSP                           |HOSP1_HOUSE_OFFICER            |SELECT          |PATIENT_DIAGNOSIS

9 rows selected.
```

*Output 92: Privileges granted to **HOSP1_U0007**.*

```
SQL> SELECT patient_id "Patient_ID", last_name "Last_Name", first_name "First_Name", address "Address",
date_of_birth FROM patient;

Patient_ID|Last_Name           |First_Name          |Address                                              |DATE_OF_BI
----------|--------------------|--------------------|-----------------------------------------------------|----------
12352     |McDonald            |Ronald              |23 Portobello Road                                   |12/12/1979
12364     |Churchill           |Winston             |88 Kenwood Drive                                     |12/12/1979
```

*Output 93: **HOSP1_U0007** reads **patient**.*

```
SQL> SELECT * FROM bed;

BED_ID    |ROOM_ID   |TYPE
----------|----------|----------
Bed001    |Room1O    |Bunk
Bed002    |Room1O    |Bunk
Bed003    |Room1O    |Bunk
Bed004    |Room1O    |Bunk
Bed005    |Room2O    |Bunk
Bed006    |Room2O    |Bunk
Bed007    |Room2O    |Bunk
Bed008    |Room2O    |Bunk
Bed009    |Room3O    |Bunk
Bed010    |Room3O    |Bunk
Bed011    |Room1H    |Bunk
Bed012    |Room1H    |Bunk
Bed013    |Room1H    |Bunk
Bed014    |Room1H    |Bunk
Bed015    |Room2H    |Bunk
Bed016    |Room2H    |Bunk
Bed017    |Room2H    |Bunk
Bed018    |Room2H    |Bunk
Bed019    |Room3H    |Bunk
Bed020    |Room3H    |Bunk
Bed021    |Room3H    |Bunk
Bed022    |Room3H    |Bunk
Bed023    |Room1G    |Electric

23 rows selected.
```

*Output 94: **HOSP1_U0007** reads **bed**.*

In contrast, user **u0007** is still active, in the same role as **u0005** (*house_officer_d*) (Output 91–94).

Separation of Duties

Tests were performed to determine whether the records in **ssd** and **dsd** correctly enforced separation of duties.

```
SQL> start ssd_dsd.sql
SQL> CONNECT hosp/hosp
Connected.
SQL>
SQL> SET ECHO ON;
SQL>
SQL> SELECT * FROM ura WHERE usr = 'u0010';

USR             |ROLE
----------------|------------------------------
u0010           |house_officer_n
```

*Output 95: User **u0010** is defined in the role house_officer_n.*

Output 95 confirms that user **u0010** is defined in the role *house_officer_n*.

```
SQL> INSERT INTO ura(usr, role) VALUES ( 'u0010', 'specialist_nurse' );
INSERT INTO ura(usr, role) VALUES ( 'u0010', 'specialist_nurse' )
            *
ERROR at line 1:
ORA-20000: Conflicting roles: cannot assign u0010 to specialist_nurse.
ORA-06512: at "HOSP.URA_BEFORE_INSERT", line 23
ORA-04088: error during execution of trigger 'HOSP.URA_BEFORE_INSERT'
```

*Output 96: Role conflict error when attempting to define user **u0010** as a specialist_nurse.*

```
SQL> SELECT * FROM ura WHERE usr = 'u0010';

USR             |ROLE
----------------|------------------------------
u0010           |house_officer_n
```

*Output 97: User **u0010** is still defined only as house_officer_n.*

```
INSERT INTO ssd( role1, role2 ) VALUES ( 'doctor', 'nurse' );
```

*Code 65: **ssd** definition preventing the same user from being both doctor and nurse.*

As expected, **u0010** could not be additionally defined as *specialist_nurse:* the attempt returned a programmer-defined error *ORA-20000: Conflicting roles* (Output 96), and the role assignment failed (Output 97). This because of a static **ssd** preventing the same user from being defined in both a *doctor* role and a *nurse* role (Code 65).

Repeating the previous **SELECT** query confirms that the attempted **INSERT** was unsuccessful.

```
SQL> INSERT INTO usr_session(usr, role) VALUES ('u0010', 'painter' );
INSERT INTO usr_session(usr, role) VALUES ('u0010', 'painter' )
            *
ERROR at line 1:
ORA-20000: Not assigned to role: cannot activate u0010 as painter.
ORA-06512: at "HOSP.USR_SESSION_BEFORE_INSERT", line 56
ORA-04088: error during execution of trigger 'HOSP.USR_SESSION_BEFORE_INSERT'
```

*Output 98: Attempt to activate user **u0010** in (non-existent) role painter.*

Here, **u0010** could not be activated in the role *painter*, because the user is not assigned to this role in **ura** (Output 98). Indeed, the role does not exist.

```
SQL> INSERT INTO usr_session(usr, role) VALUES ('u0010', 'manager' );
INSERT INTO usr_session(usr, role) VALUES ('u0010', 'manager' )
            *
ERROR at line 1:
ORA-20000: Conflicting roles: cannot activate u0010 as manager.
ORA-06512: at "HOSP.USR_SESSION_BEFORE_INSERT", line 52
ORA-04088: error during execution of trigger 'HOSP.USR_SESSION_BEFORE_INSERT'
```

*Output 99: Role conflict error when attempting to define user **u0010** as a manager.*

```
INSERT INTO ssd( role1, role2 ) VALUES ( 'manager', 'consultant' );
```

*Code 66: **ssd** definition preventing the same user from being both manager and consultant.*

The attempt to activate **u0010** as a *manager* also fails (Output 99), because of another **ssd** definition (Code 66). In any case, u0010 is not defined as a *manager*, so this attempt would fail anyway.

```
SQL> INSERT INTO usr_session(usr, role) VALUES ('u0010', 'consultant' );
INSERT INTO usr_session(usr, role) VALUES ('u0010', 'consultant' )
            *
ERROR at line 1:
ORA-20000: Not assigned to role: cannot activate u0010 as consultant.
ORA-06512: at "HOSP.USR_SESSION_BEFORE_INSERT", line 56
ORA-04088: error during execution of trigger 'HOSP.USR_SESSION_BEFORE_INSERT'
```

*Output 100: Attempt to activate user **u0010** in role consultant to which he is not assigned.*

Again, an attempt to activate **u0010** as a role to which he is not assigned is unsuccessful. This time, the role is *consultant*, which does exist (Output 100).

```
SQL> INSERT INTO ura(usr, role) VALUES ('u0010', 'consultant' );

1 row created.
```

*Output 101: Assigning user **u0010** in role consultant.*

The **ura** assignment in Output 101 is successful, since it is not prevented by any **ssd** entries.

```
SQL> INSERT INTO usr_session(usr, role)
VALUES ('u0010', 'consultant' );

1 row created.
```

*Output 102: Activating user **u0010** in role consultant.*

```
SQL> SELECT * FROM ura WHERE usr = 'u0010';

USR              |ROLE
-----------------|------------------------------
u0010            |consultant
u0010            |house_officer_n
```

*Output 103: User **u0010** now assigned to both consultant and house_officer_n.*

The user can now be activated as a *consultant* (Output 102). Output 103 confirms that **u0010** is now assigned to two roles.

```
SQL> INSERT INTO usr_session( usr, role ) VALUES ( 'u0016', 'student_nurse_n' );
INSERT INTO usr_session( usr, role ) VALUES ( 'u0016', 'student_nurse_n' )
            *
ERROR at line 1:
ORA-20000: Conflicting roles: cannot activate u0016 as student_nurse_n.
ORA-06512: at "HOSP.USR_SESSION_BEFORE_INSERT", line 52
ORA-04088: error during execution of trigger 'HOSP.USR_SESSION_BEFORE_INSERT'
```

*Output 104: Attempt to activate user **u0016** in role student_nurse_n causing a **dsd** conflict.*

```
INSERT INTO dsd( role1, role2 ) VALUES ( 'day_duty', 'night_duty' );
```

Code 67: dsd constraint preventing simultaneous activation of day_duty and night_duty roles.

Output 104 shows that an attempt to activate **u0016** in the role *student_nurse_n* is unsuccessful because **u0016** is already active as *student_nurse_d*. A user cannot be simultaneously active in both a *day_duty* and a *night_duty* role. This is due to a **dsd** constraint (Code 67).
