6-2-2008

Making RBAC Work in Dynamic, Fast-Changing Corporate Environments

Ruslan Y. Dimov
Dartmouth College

Dartmouth College Computer Science Technical Report TR2008-624

Making RBAC Work in Dynamic, Fast-Changing Corporate Environments

Senior Honors Thesis

Ruslan Dimov[1]
Advisor: Sean W. Smith[2]
Research Assistant: Sara Sinclair[3]

June 2, 2008

[1] Undergraduate in Computer Science Department, Dartmouth College
[2] Professor at Computer Science Department, Dartmouth College
[3] Graduate Student in Computer Science Department, Dartmouth College

Abstract

In large organizations with tens of thousands of employees, managing individual people's permissions is tedious and error prone, and thus a possible source of security risks. Role-Based Access Control addresses this problem by grouping users into *roles*, which reflect job functions in the corporation. Permissions are assigned to roles instead of directly to users, which means that all users assigned to a role have the same set of permissions with respect to that role. However, adoption of RBAC in organizations such as investment banks is hindered by two main factors: first, it is costly and time-consuming to define roles. Second, there are certain job functions (such as "consultant") that cannot be expressed as RBAC roles, because their users need to have different permission sets.

The topic of this thesis is to investigate whether roles can be applied to domains that exhibit the peculiarities of the investment bank example. We introduce a new framework for roles that allows us to separately represent what the role *means* as a job function and what permissions its individual users have. That way we maintain the key property of RBAC – that the number of roles is small, while allowing for variations among users. We have also investigated machine learning approaches in order to figure out whether roles are concepts that can be learned or approximated by a function. We present our findings that certain learning schemes, such as Probably Approximately Correct (PAC) earning and Instance-based learning are not applicable to roles, while others – such as decision-tree learning, might be useful.

Acknowledgments

I would like to thank first and foremost my thesis advisor – Sean Smith, and my research assistant – Sara Sinclair, for their skillful guidance, invaluable brainstorming sessions, and the countless hours they spent reading and commenting on my drafts. This thesis would not have been written without your reality checks and sharp minds. I would also like to thank Mark Strembeck from Vienna University of Economics and Business Administration for his interest in my research and the papers that he pointed me to about scenario-driven role engineering. I also owe thanks to Gerhard Schimpf of SMF Team, Germany, Chairman of the German Chapter of the ACM, for his help on finding research materials about bottom-up role discovery approaches. Last but not least, I thank my friends for bearing with me throughout the last couple of terms, and learning probably more than they ever wanted to about role-based access control.

This report results in part from a research program in the Institute for Security Technology Studies at Dartmouth College, supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security.

Contents

1 Introduction 5

1.1 Overview of RBAC 5

1.2 Formal Definition of the RBAC Model 6

1.3 Challenges of Deploying RBAC 7

1.3.1 Role Engineering 7

1.3.2 Finer-Grained Control on Permissions 9

1.3.3 Issues With the Traditional Role Concept 9

2 Related Work 10

2.1 RBAC in the Enterprise 10

2.2 Role Parameterization 11

2.3 Temporal and Other Context Constraints 12

3 Current Challenges 13

3.1 Dynamic Task Reassignment 13

3.2 Dynamic Resource Pool 15

4 Permission Types 16

4.1 Background and Motivation 16

4.1.1 Roles as Functions 16

4.2 Permission Types in Depth 19

4.2.1 Formal Definitions 19

4.2.2 Related Work 20

4.3 Object and Operation Taxonomies 21

4.3.1 Motivation 21

4.3.2 Representation of the Taxonomies 22

4.3.3 Building the Taxonomies 25

4.4 Permission Type Constraints 28

4.4.1 Motivation 28

4.4.2 Formal Definition 29

4.5 Putting it All Together 29

4.5.1 Application to Roles 29

4.5.2 Usefulness and Tradeoffs of the Model 31

5 Application of Learning Approaches 32

5.1 Roles as a Learning Problem 32

5.1.1 The Role as an Access Control Function 32

5.1.2 Learning Role Structure 34

5.1.3 The Challenge 36

5.2 Learning Approaches That Might Not Work 37

5.2.1 Computational Machine Learning Theory 37

5.2.2 Instance-Based Learning 38

5.3 Learning Approaches That Might Work 39

6 Methodology and Experimentation 40

6.1 Sample Data 40

6.1.1 Explanation of the Data 40

6.1.2 Format of the Data 41

6.2 Permission Type Metrics 41

6.3 Learning Role Structure 43

6.4 Conclusions 44

7 Future Research 45

7.1 Role Housekeeping 46

7.1.1 Role Classes 46

7.1.2 Rules and Context Constraints 48

7.2 Role Functions and Learning 50

7.2.1 Time as a Parameter 50

7.2.2 Numerical Representations 51

A Sample Data 55

B Collected Statistics 57

C Role Structure Function Data 60

D See5 Output 61

1 Introduction

1.1 Overview of RBAC

Before we proceed to tell the story behind the cryptic acronym RBAC, we need to take a step back and briefly explain the need for corporate access control. In any given corporation, employees have *permissions* to certain resources, i.e. a set of allowed operations they can perform on these resources. Permissions are needed because the corporation would like to protect its assets (including data) from malicious use, such as leaking sensitive information, embezzling money, modifying crucial data, etc. This, together with separation of duty concerns, and the well-known security principle of least privilege, leads to the conclusion that employees should only have access to the resources they need to perform their duties in the company. However, large corporations can have tens of thousands of employees or more, and manually specifying the permissions for each of them would be extremely tedious and time-taking. That's why permissions are frequently "copied over", e.g. if Bob takes Alice's place in the company (or even a similar job position), he gets Alice's permissions. This, however, poses a security risk, because the permissions Alice had reflect her access needs over the period of time she held the position, and may as well be a superset of what a beginner like Bob needs. This also illustrates another problem, common in the real world — Alice may have permissions she no longer needs, but which were never removed from her permission set. In summary, management of individual employees' permissions is onerous and therefore promotes uncareful practices, which may lead to over- or under-entitlement.

Role-based Access Control, or RBAC, has gained a lot of traction since it was first introduced in the early 90s [4]. It is an access control model which aims to reduce the managerial effort of assigning privileges to users in large organizations. Under RBAC, users are grouped into *roles*, where a role is essentially a set of permissions and reflects the function of its respective users in the organization. Thus, permissions are no longer assigned directly to users, but to roles instead. Under this representation, people having the same role have an identical set of permissions, a simplifying assumption we will later need to revisit. Finally, a user may have (and indeed frequently has) more than one role. Another advantage of RBAC is that it is policy-neutral, i.e. it can be configured to enforce both mandatory (MAC) and discretionary (DAC) access control policies [5, 16]. For a more comprehensive treatment of MAC and DAC policies and how they are enforced via RBAC, the reader is encouraged to read the paper by Sandhu et al. [16].

However, traditional RBAC roles are too "static" for corporations where job assignments or the available human and physical resources are highly dynamic entities, i.e. they frequently change. In the former case, that might be due to the fluid nature of one's duties within the company (i.e. he or she might perform different tasks at different times). In the latter case, it might be due to mergers and acquisitions. Investment banks are a good example of such *dynamic, fast-changing corporate environments (DFCEs)*, and one that we will use throughout the text. The purpose of this thesis is, then, to come up with a framework for enhancing RBAC with an alternative definition of a role, so that it can be more successfully applied to DFCEs. We recognize that our framework does not provide a complete solution to the problem, as such a solution would require a lot of continual research effort, but it is rather a starting point for future research in “expanding” the role concept.

To demonstrate the benefit of using RBAC, I will use the discussion in the recent book on RBAC by Ferraiolo et al. [5, pages 19-20]. The authors note that there is usually a direct relationship between administration costs and the number of associations (e.g. user-permission) that are needed to describe and maintain a certain access control policy. If we let U be the set of users in *one particular* job position and P be the set of permissions required by this job position, then the direct mapping between users and permissions would have $\mid U \mid \cdot \mid P \mid$ associations, one for each user-permission pair. On the other hand, since a role is a set of permissions, we can view P as a role, and in that case we need $\mid U \mid$ user-role associations, and $\mid P \mid$ role-permission associations, or a total of $\mid U \mid + \mid P \mid$ associations. Therefore, RBAC provides an administrative advantage whenever $\mid U \mid + \mid P \mid < \mid U \mid \cdot \mid P \mid$, which is true for $\mid U \mid, \mid P \mid > 2$, or at least two individuals per role and two permissions per role. This, the authors argue, accounts for the majority of roles in large organizations.

There has been significant research attention to RBAC, and efforts to standardize it, as later chapters will show. The first such effort is the seminal paper by Ferraiolo et al. in 1992 [4], which aimed to address the deficiencies of the existing commercial security solutions at the time. In 1996 Sandhu et al. [21] presented an overview of four RBAC varieties — core RBAC [4], RBAC with constraints, RBAC with role hierarchies and finally RBAC with both constraints and role hierarchies. In 2000, Ferraiolo et al. [20, 6] proposed a NIST standard for RBAC, which in 2004 was approved as ANSI/INCITS Standard 359-2004 by the InterNational Committee for Information Technology Standards [5, page 15]. Throughout this work, I will refer to these standards as “traditional” RBAC.

RBAC has successfully been applied to several domains — health care, government and military applications, and banking to name a few. A rich database of case studies is maintained by the NIST RBAC Group and is available on their website [15].

1.2 Formal Definition of the RBAC Model

Another advantage of the RBAC model, and a factor that enables its subsequent extensions, is its formal algebraic definition. The basic building blocks of core RBAC are the sets of users, subjects (i.e. a process or a collection of processes acting on the user's behalf), operations, objects and roles, and the relations between them. I have taken the following definitions of the essential RBAC components directly from Ferraiolo et al. [5, page 66]:

- $USERS$, $ROLES$, OPS, and OBS (users, roles, operations and objects, respectively).

- $UA \subseteq USERS \times ROLES$, a many-to-many mapping between users and roles.
- $assigned\_users : (r : ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users.
- $PRMS = 2^{OPS \times OBS}$, the set of permissions.
- $PA \subseteq PRMS \times ROLES$, a many-to-many mapping between permissions and roles.
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role r onto a set of permissions.
- $SUBJECTS$, the set of subjects.
- $subject\_user(s : SUBJECTS) \rightarrow USERS$, the mapping of a subject s onto the subject's associated user.
- $subject\_roles(s : SUBJECTS) \rightarrow 2^{ROLES}$, the mapping of a subject s onto a set of roles.

1.3 Challenges of Deploying RBAC

Despite its promise and apparent applicability, traditional RBAC faces challenges that make its wide adoption in the corporate world difficult. We already alluded to some of them in Section 1.1. In addition, the fundamental task of defining the roles in an RBAC deployment, a process known as *role engineering*, turns out to be quite tricky; furthermore, many companies realize they need finer-grained control on permissions than what the model provides, and finally — in certain domains, such as highly dynamic organizations (e.g. investment banks), the very concept of a role as a set of permissions seems to be inadequate. I will proceed to elucidate these three problematic areas.

1.3.1 Role Engineering

There are two main approaches to role engineering — a top-down and a bottom-up one. The scenario-driven model, proposed by Mark Strembeck and Gustaf Neumann [14], is a widely used top-down approach. The idea is that a given *work profile* (analogous to a job position in an organization) is decomposed into a set of *tasks*, each of which describes a particular activity within the work profile. Since the same task could be carried out in different ways under different circumstances, tasks are further decomposed into *scenarios.* A *scenario* could be viewed as a sequence of steps in order to accomplish a certain task, such as the example the authors provide about withdrawing cash from an ATM. This representation of scenarios makes it easy to infer the needed permissions for any given scenario by listing the permissions needed at each step. Then, through tasks, permissions for a scenario can be related back to the work profile. The top-down decomposition process outline above is meant to be iterative — the substeps are repeated until a complete and accurate model has been created.

However, the scenario-driven role engineering process can be quite time-consuming and tedious, because it requires the time and effort of a lot of people to identify work profiles, tasks and scenarios, and refine them until the result is perceived as "complete". This would be impractical in some DFCEs such as investment banks. There, it is first of all logistically hard to get managerial staff together to sit and think about what the different work profiles should be and how they are further decomposed into scenarios and tasks. Secondly, it is not even clear whether managers know all the specifics of their employees' duties to come up with a correct and complete decomposition. Thirdly, every second lost to secondary tasks (such as deriving a complete and correct role model for the bank) is viewed as a decline to productivity, and it is hard to convince working specialists to devote time to the process. Apart from its being onerous, the scenario-driven process ignores the fact that the company where RBAC is to be deployed has already been running for a while, and the permissions employees have more or less accurately describe what they actually need.

The two main challenges to top-down approaches identified above have been discussed in the literature, especially where an alternative, bottom-up approach is presented [25, 12, 27]. The motivation behind bottom-up approaches is that, as discussed, existing permissions are a good indicator of what employees need to do their job. Bottom-up approaches use data-mining techniques such as clustering to infer roles (viewed as clusters of permissions) and to decide which users should be grouped in the same role. One advantage of using such techniques is that the need for human intervention is decreased in comparison with top-down methods, since the process is meant to be automated. One major drawback is that the resulting roles are not very meaningful and do not map well to the organizational structure of the company [27, 22]. There exist bottom-up approaches that do not rely on data mining, such as Graph Optimization [27], which uses matrix decomposition of the user-permission assignment matrix to come up with an optimal role hierarchy, where roles are encoded in the form of user-role and role-permission matrices. In fact, there is an interesting theoretical result, which shows that the *Role Mining Problem*, or RMP, is NP-complete if defined as the optimal matrix decomposition of the user-permission matrix into a user-role and a role-permission matrix, where the measure of optimality is the smallest number of resulting roles [24]. Finally, a common problem with all bottom-up approaches is that the resulting roles become out-of-date very fast in DFCEs due to the rapid evolution and changing nature of corporate duties.

In light of the above, many researchers agree that a hybrid approach is needed — one that borrows ideas from top-down and bottom-up approaches [10, 27, 5].

1.3.2 Finer-Grained Control on Permissions

Another challenge to core RBAC is meeting the needs of companies for additional levels of fine-grained access control. For example, it is common in many organizations that employees have a certain function or position only for a specified period of time and at a certain interval, say a part-time worker who can only do his job in certain days and between certain hours [7]. Furthermore, the core model assumes that all users of a role have identical permissions for that role. This is a problem if one wants to, for example [1], define the role *Account_Holder* to represent an account holder in a bank — clearly, we want each user to be able to only see and modify their own account information. Core RBAC can solve this by introduce a lot of account-specific roles, but that undermines the usefulness of RBAC as a tool to simplify management of people and resources. Lastly, certain domains and applications require that arbitrary constraints can be put on permissions based on context attributes [23]. Even though core RBAC supports permission constraints, they are mostly separation-of-duty (SoD) constraints.

This is not at all an exhaustive list, but it comes to show that there has been significant interest in enhancing RBAC with features needed by real-world organizations. In Chapter 2 we will discuss some RBAC extensions that have a bearing on the current research, and show the evolution of RBAC over time.

1.3.3 Issues With the Traditional Role Concept

The previous two sections identified directions for improvement of RBAC, but they did not question the formulation of its central concert — the role. We turn our attention to it now, in view of the concerns raised at the very beginning of the chapter.

Despite the numerous extensions of RBAC, it still does not address certain scenarios in the real world. For example, the hypothetical RBAC role of *Analyst* in an investment bank is in fact a cover term for a range of very different activities, as there are many types of analysts. Moreover, a typical employee in an investment bank may have multiple changing functions within the bank that depend on a series of factors — the current project(s) the employee is working on, the fact that the employee is acting as a backup for someone else, etc. All of this has an influence on the current set of permissions, and respectively roles, that an employee holds. Viewing a role as a "static" set of permissions does not seem to suit such employees too well, because it does not capture the dynamism of their function in the company. Contrast this with the role *Bank_Teller* in a commercial bank – the duties of a bank teller don't change significantly over time, so it can, in fact, be conveniently described in core RBAC terms.

This brings us to the focus of the current research – revisiting the concept of a "role" and making it more flexible and malleable, so it can hopefully account for the unique circumstances of highly dynamic organizations. Chapter 2 will focus on prior work on extending core RBAC, and Chapter 3 will explore in more detail some tricky scenarios that are not addressed by these extensions. Chapter 4 will introduce the concept of a *permission type*, which is an attempt to glean information about the internal structure of a role, and a step toward providing a middle ground between top-down and bottom-up role engineering approaches. Chapter 5 will introduce a new view on roles as learnable concepts, and investigate the applicability of several machine learning algorithms to this domain. In Chapter 6 we present some experiments and measurements of the usefulness of the previously described new approaches, and Chapter 7 concludes the thesis with insights we have gained and ideas for future research.

2 Related Work

In this chapter we are going to look at several extensions of RBAC, which address some of the challenges to its commercial deployment.

2.1 RBAC in the Enterprise

First off, it is important to note that traditional roles contain permissions that are targeted for a particular hardware and/or software platform, such as an operating system (OS), a database management system (DBMS), and a mainframe or a server. However, large organizations such as investment banks usually have a varied mix of IT platforms, and a user's job may require permissions on multiple systems. Therefore, roles may span multiple heterogeneous systems. One way to address this issue is to encapsulate the permissions on the different systems in different roles. This would, however, be tedious, and we would end up with a lot of additional roles, which undermines the purpose of RBAC to simplify and expedite the management of users and permissions within an organization.

An alternative approach would be to make roles more platform-independent so that they could accommodate for permissions across multiple systems. This approach has been realized as an extension of RBAC, which is called Enterprise Role-Based Access Control (ERBAC) [5, 11, 9, 12, 10]. The elements of ERBAC are mostly the same as in traditional RBAC – we have users, roles and permissions, as well as the usual relations between them. However, one difference is that ERBAC does not support sessions, which are described in the NIST RBAC standard [20]. Sessions are used, among other things, to determine which of the roles assigned to a user should be enabled during the current dialog of the user with the system. This is why dynamic separation of duty (SoD), which is possible with traditional RBAC, is not supported directly by ERBAC.

In their treatment of the matter, Ferraiolo et al. [5] introduce the concepts of a *generalized* *data resource* and a *generalized operation*. These concepts make it possible to abstract away the platform-specific assumptions we would be making if we were to specify privileges for a particular type of resource. Thus, we end up with generalized permissions of the form <generalized-op, generalized-res>, which is a theme we will return to when we introduce the concept of *permission types*. Finally, Ferraiolo et al. present a way to express the ERBAC components and access control data using an XML schema language.

In a different work [9], ERBAC is augmented with a type of role parameterization as a result of practical experience with deploying the access control model in large corporations. Both RBAC and ERBAC allow for the specification of role hierarchies, so that roles can better reflect the organizational structure. Child roles "inherit" all the permissions of their parent roles, which removes some redundancy from role permission sets and thus results in increased ease of management. However, role hierarchies can be defined in terms of multiple factors – job position, location, company branch, etc. Taking all these factors into consideration would result in a very large number of roles, one for each possible combination. In contrast, the authors propose a scheme where the role hierarchy is defined based on one factor and the remaining ones are used as *parameters* to the resultant roles. This is achieved by using *attributes* and *rules*. Attributes are used to specify any additional information that may be relevant to the decision whether to grant access, such as branch location, user name and job position, etc. They can be applied to users, roles, user-role assignments, permission-role assignments and role-to-role assignments. Rules determine what should be done when attribute values or assignments change, and can therefore be used to dynamically recompute the roles a particular user holds at any given time. The idea of using rules to overcome the static nature of traditional RBAC roles has led to the introduction of Rule-Based RBAC, or RB-RBAC [11].

2.2 Role Parameterization

The idea of role parameterization has been developed into a formal model by Ali Abdallah and Etienne Khayat [1]. It is not rule-based, and gives a precise definition to parameters and how they are used. As mentioned before, under traditional RBAC two users sharing a role have an identical set of permissions with respect to that role. However, as the authors note, this is not always good, because users exercise their permissions differently based on what their specific duties are. The motivating example used to illustrate this issue is in the online banking domain, where the subjects are account holders and the objects are the bank accounts. If we try to model the interactions between subjects and objects in this scenario with a traditional RBAC role – *Account_Holder* – we run into a problem: since each user should only be able to view and modify their own account, no two users of the *Account_Holder* role should have identical permissions! The solution is to use *parameters*, in this case the account number, to come up with *parameterized roles* (or permissions, objects, etc.). In our banking example, the role *Account_Holder* would be replaced by the instantiations of the role *Account_Holder(m)*, where m is drawn from the set of account numbers. Below is a summary of some important facts about role parameterization:

- it can be done *iteratively*, i.e. we may have an arbitrary-depth nesting of parameters until access control is sufficiently fine-grained for the organization's needs. (However, using many parameters results in a higher number of instances of the parameterized role, so is it suggested that this feature be used sparingly.)
- parameter values can be drawn from any set of abstract labels; however it is advisable that parameters are meaningful with respect to the domain, such as account number, bank branch, etc. for the banking domain.
- as an immediate corollary of parameterization, users assigned to different instantiations of the same parameterized role may have different permissions.

However, parameterized RBAC is not a panacea – even with parameterized roles, we have the same "types" of permissions for each "incarnation" of the role (e.g. all *Account_Holder*s can view their current balance, and withdraw or deposit money to their account). In the *Analyst* example quoted earlier, different analysts may do entirely different things as opposed to constrained varieties of the same set of tasks.

2.3 Temporal and Other Context Constraints

A conceptually similar, but implementationally quite different extension to RBAC is the ability to engineer arbitrary context constraints, presented in a paper by Mark Strembeck and Gustaf Neumann [23]. They start off by defining a categorization for constraints relevant to an RBAC model, identifying the following dimensions:

- *static* vs. *dynamic* constraints, whereby the former are evaluated at administration time (e.g static SoD), and the latter – at runtime (e.g. dynamic SoD).
- *endogenous* vs. *exogenous* constraints, i.e. constraints intrinsic to the RBAC model vs. ones that take external information into consideration (e.g. temporal constraints).
- *authorization* vs. *assignment* constraints – the former place additional restrictions with a bearing on the access control decision (e.g. Chinese Wall policies), while the latter control the way in which users are assigned to roles, or permissions to roles.

Based on the above, a *context constraint* is a *dynamic exogenous authorization constraint*. There are several components to a context constraint – a **context attribute** is an attribute of the environment whose value may dynamically change (e.g. date, time), or whose value is drawn from a set of possible options (e.g. location). We need a **context function** to obtain the current value of the context attribute (e.g. *date*()). Tests on attribute values are performed in **context conditions**, which are predicates with an operator and two or more operands, the first of which is always a context attribute. Finally, a **context constraint** is a clause containing one or more context conditions, and a permission associated with one or more context constraints is called a *conditional permission.* The authors also discuss the `xoRBAC` software component, which exemplifies the administration of RBAC with context constraints.

Temporal constraints, briefly mentioned above, receive their own formal treatment whose result is Temporal RBAC, or TRBAC, due to Elisa Bertino et al. [5, 7]. TRBAC is appropriate for domains where temporal semantics is needed, such as workflow-based systems [7]. Other examples include cases where a certain role should only be enabled at a particular time and for a particular duration, such as a part-time job or a routine maintenance task, or acting as a back-up for someone else. Under TRBAC, for a user to have the permissions of a certain role, it not only has to be enabled (the user is authorized to activate the role), but it also has to be *active* at the time the user request a permission from that role. TRBAC also introduces the notion of *role triggers*, which are a mechanism to enforce temporal relationships between roles, so that one can say "Enable role X when role Y is enabled". One example in [7] is that the role `NightNurse` should be enabled when the role `NightDoctor` is enabled.

Despite the maturity of the RBAC model and the numerous proposed extensions to it (a sample of which has been discussed above), there are still fundamental challenges to its wide-scale adoption. We look more closely at some tricky scenarios in the next chapter.

3 Current Challenges

The following discussion is based on anecdotal evidence collected by Dartmouth College security researchers. In their dialogs with representatives of different companies, they have identified several scenarios that pose problems for the deployment of traditional RBAC. While this is by no means an exhaustive list of tricky scenarios, it should give the reader an idea of the current challenges faced by RBAC.

3.1 Dynamic Task Reassignment

Some employees in, say, an investment bank, are frequently being totally reassigned. For example, someone who works as a consultant might spend a few weeks working on account A, then a few more weeks working on account B, then a week working half-time between both accounts A and C. A conventional RBAC *Consultant* role, therefore, could not be defined because the pattern shown above does not have to be the same for all consultants in the company. This precludes one possible solution to the problem – simply update the permissions associated with the *Consultant* role. However, that could give consultants working on entirely different projects access to resources they should not be able to touch, which creates a risk of an insider attack.

We could try to rephrase the problem and instead of coming up with one *Consultant* role we might have several varieties of it, so that a reassignment of responsibilities is accompanied by a reassignment of roles. The formulation of the above scenario suggests that we could, much in the spirit of the *Account_Holder* example, parameterize the *Consultant* role by account number. Therefore the consultant whose assignment pattern we presented will have the following role assignment pattern: *Consultant(A)*, *Consultant(B)*, and finally *Consultant(A)* and *Consultant(C)*. However, that does not remove the need for frequent manual reassignment of roles (in this case different parameterizations of the same role), which has to be done for *any* consultant.

In this scenario, having multiple instantiations of the *Consultant(m)* role is offset by having one parameterized role in the RBAC model. However, the example is simplistic, because the consultant's reassignment of tasks is accompanied by changes of permissions related to only one type of resource, and the same type of resource throughout the reassignment pattern – e.g. bank accounts. In general, the types of resources an employee needs between reassignments may change, and the number of relevant resource types may be more than one. Parameterized RBAC allows for an arbitrary number of role parameters, so in theory it could address this problem, but we would end up with many parameterized roles, and an even more overwhelming number of actual instantiations of these roles. Moreover, we cannot have all the needed parameterized *Consultant* roles at the time of the RBAC model definition, because that implies we know, indefinitely into the future, what projects a consultant may be given. Thus, even with parameterized RBAC, the dynamic task reassignment problem does not reduce to simple role reassignment.

The same challenge of not knowing *a priori* what set of roles can describe the changing nature of an employee's task assignment exists for TRBAC and RBAC with context constraints. TRBAC would push the complexity to defining temporal roles that capture the property that each reassignment lasts for a particular amount of time, and reassignments form a temporal sequence. However, this sequence is not necessarily periodic – each new assignment can last an arbitrary amount of time, and assignments can recur at irregular intervals (e.g. account A that appears twice in the example above). TRBAC was designed to handle scenarios where periodicity and duration are more uniform. RBAC with arbitrary context constraints would, in turn, push the complexity to defining elaborate context constraints for the different *Consultant* roles. The constraints would check for the values of as many context attributes as needed to describe the reassignment, which means that the role engineering stage would be dominated by defining various context constraints on a possibly large set of attributes[4].

[4] Note that context constraints are defined during the top-down role engineering stage [23]

3.2 Dynamic Resource Pool

Another challenge to traditional RBAC comes from the fact that the set of resources to which access is being controlled might change. Some examples include: purchasing new software to replace the old, getting a new client (and thus new accounts or database entries), and more notably – mergers and acquisitions. The latter is an especially problematic case, because we have a new organizational substructure with a whole new set of resources. This has implications both for the role hierarchy of the RBAC model, since the newly acquired company would have to be integrated into the acquiring company's existing organizational hierarchy, and for the set of supported roles, objects and permissions (we get new offices, physical resources such as servers, other resources such as accounts, etc.).

In terms of an RBAC deployment, a merger or acquisition amounts to a large-scale role re-engineering project. It might be the case that existing corporate roles change, because their associated users have new responsibilities and require access to new resources, so they have to be re-engineered in order to preserve the accuracy of the RBAC model. However, as discussed in Section 1.3.1, the existing role engineering approaches have serious shortcomings – they are either too time-consuming (top-down approaches), or produce roles that may not be meaningful (bottom-up approaches). Since the extended RBAC models discussed in Chapter 2 still rely on one or the other, they cannot address the problem.

The above discussion is important for two reasons – firstly, since the challenge presented is due to fundamental properties of deploying RBAC, i.e. having a complicated role engineering stage, we are left with the question if RBAC is even applicable to corporations which experience frequent acquisitions or mergers, such as an investment bank. Secondly, it precisely identifies the bottleneck – it is *existing role engineering approaches* that limit the usefulness of RBAC for fast-changing domains. This also gives us a clue as to what the solution to the problem would be – coming up with a role engineering procedure that relies less on human intervention and more on automation, without a sacrifice to the meaningfulness of the resultant roles. This is also contingent upon the definition of a role, since both the top-down and the bottom-up approaches are justified, for different reasons, by viewing roles as sets of permissions. This thesis is a step toward coming up with a new definition of what a role should be, with the hope that this would facilitate the role engineering stage and thus make a modified RBAC model applicable to a wider variety of domains.

4 Permission Types

In this chapter we define our concept of *permission types (PTs)*, and discuss its usefulness for role engineering as well as for revisiting the concept of a "role". Section 4.1 presents our motivation for developing permission types, Section 4.2 provides the actual definition of permission types, and Section 4.3 introduces our concepts of *object and operation taxonomy*, which are prerequisites for defining permission types in an organization. However, as we point out in Section 4.4, there needs to be a mapping from the quite general permission types to the actual permissions a user should have in a given role, and we achieve that by introducing our concept of *permission type constraints.* Finally, in Section 4.5 we propose a role model which utilizes permission types and permission type constraints in order to meet the challenges of DFCEs.

4.1 Background and Motivation

In this section we provide the motivation for developing the concept of permission types. We believe that this background information is useful in order for the reader to understand the actual definition or permission types. Furthermore, some of the questions we pose here are key for the discussions in subsequent chapters.

4.1.1 Roles as Functions

We already indicated in previous chapters that the RBAC definition of a role as a set of permissions, despite its usefulness, does not always serve well the purposes of dynamic, fast-changing corporate environments (DFCEs). In Chapter 3 we identified specific scenarios which are hard, if not impossible, to address with conventional roles. Therefore, we set out in this research to develop an alternative definition of a role – one that would allow us to account for the changing nature over time of users' permissions within roles in our domain of interest.

As a starting assumption, we hypothesized that roles had an underlying internal structure as opposed to being "flat" sets of permissions. This structure, we posited, accounted for the fact that despite the difficulties of expressing certain corporate roles such as analyst in RBAC terms, we still have a mental model for what an analyst *is* and *does.* However, it is not easy to map our intuition about roles onto their digital representation. Indeed, as supported by research of the Dartmouth College PKI Laboratory conducted at various companies, possibly no one really knows what this internal structure might be like.

Therefore, we decided to pose role structure discovery as a machine learning problem: if we view roles as *functions*, can they be approximated by an appropriate learning algorithm?

The answer to this question depends on the kind of functions by which we represent roles, and the choice of learning algorithms applicable to such functions. As we are going to see in Chapter 5, representing roles as functions is a difficult problem by itself, and there is more than one way to do it. However, since permission types were developed to help us learn role functions that take access requests and return a yes or no answer, we provide an outline of these functions here before the in-depth discussion in the next chapter[5].

Conceptually, a role function could be thought of as a Boolean function that takes as its input an access request – described by the *operation* that a *subject* wants to perform on an *object* – and returns 1 (or `true`) if the request should be granted and 0 (or `false`) if it should be denied. Therefore, a naïve definition of a role function, using the RBAC entities described in Chapter 1, could be:

$$role : SUBJECTS \times OPS \times OBS \rightarrow \{0, 1\}.$$

However, we still want to use a role as a means of managing the permissions of a specific *set* of users. Therefore, any role function as defined above should be restricted only to subjects that correspond to users having that role.

It is important to note that role functions take arguments drawn from finite sets of discrete values – the sets of subjects, operations and objects. The process of learning such functions is called *classification*, as opposed to learning functions on continuous-valued arguments, which is called *regression* [19]. Therefore, naturally, the approximating function we are going to learn for a role is called a *classifier.*

One important algorithm for learning classifiers is *decision-tree learning* [26, 19, 13]. It is a variety of *inductive learning*, where we provide the algorithm with a *training set* of correctly classified examples, or *instances* of the learning problem, and the algorithm produces an approximation that best fits the training data, called the *hypothesis.* In our case, an instance would consist of specific values for the subject, operation and object, as well as the Boolean output of the hypothetical role function, for example:

$$< jsmith, update, db\_5, 0 > .$$

This should be interpreted as: the user whose username is *jsmith* (or a process acting on their behalf), should be denied access when they try to perform an *update* operation on the database *db_5.* To repeat this in machine learning terms, each instance is specified by the values of its *attributes* (in our case the attributes are subject, operation, object and grant_access).

Now, decision-tree learning outputs its hypothesis in the form of a *decision tree.* Figure 1 is a simple example of a decision tree, which classifies days on whether Steve will study

[5] Importantly, we are going to see that permission types remain a useful and expressive concept even for other kinds of role functions

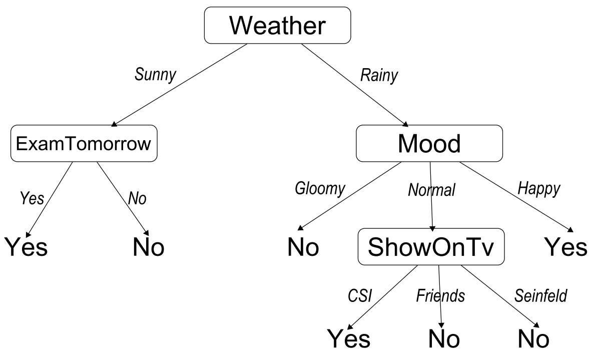

Figure 1: An example decision tree that learns the concept *WillStudy*

or not. Decision trees work as follows: we are given an instance to classify, so we pick an attribute to start with, and based on its value we make a decision of which branch of the tree to follow. At the new node, we test the value of another attribute and branch off again. This step is repeated until we reach a *leaf*, e.g. a classification (in our case a 1 or a 0). Each path along the tree is, then, a sequence of tests on the attribute values in an instance, which produces a classification of that instance. The important question is, therefore, how to pick the attribute to test next. As mentioned above, the learning algorithm is given a training set of classified instances, and it processes them in order. In decision-tree learning, we pick the attribute whose value would best split the remaining instances into sets, such that the members of each set would be predominantly classified as either negative or positive (in the case of a binary classifier). So if, for example, the values of the *subject* attribute split the remaining access requests into sets where most were either granted (grant_access = 1) or denied (grant_access = 0), then it would be a good candidate to be picked next by the learning algorithm.

A problem arises when an attribute has too many values. Consider the problem of classifying employees in a company as efficient or inefficient. If we allow the employee ID number to be an attribute (say *employee_id*), then the decision-tree learning algorithm would pick it first, because it splits the instances into disjoint singleton sets, one for each employee, which can be uniquely classified as either efficient or inefficient. Therefore, the other attributes will not even matter. However, it is very unlikely that employee efficiency depends solely on employee ID – there may be other factors such as the presence of noise, the average number of interruptions per hour, etc. This is an extreme example, but it shows that decision-tree algorithms are biased towards attributes with a high branching factor, and sometimes that may produce a tree that is not meaningful.

That leads us to a major problem the above naïve definition of the role function has – its input arguments draw their values from very large sets: even though, as noted above, we consider only a small subset of the tens of thousands of users in a large corporation (the ones that have the role in question), we still have a huge number of objects (anything that is considered a “resource” in the corporation), and a fairly big variety of operations that can be performed on these objects. This means that the instances of the role learning problem will have attributes with a very high branching factor, which is undesirable for decision-tree learning. *Permission types* address the high branching factor for objects and operations. A permission type is an ordered pair ($optype, obtype$), where:

- *optype* refers to an *operation type*, e.g. a generalization for several semantically similar operations;
- *obtype* refers to an *object type*, e.g. a collective term for a set of resources (objects) that differ only by their particular configuration or other contextual information – for example PCs, Macs and Linux boxes are all representatives of the object type `computer`.

The usefulness of permission types comes from the observation that the sets of operation and object types are smaller than the sets of actual operations and objects, respectively. Thus, under the decision-tree learning view, it would be easier and less error-prone to learn object and operation types. Also, we can view permission types as a layer of abstraction above actual permissions.

4.2 Permission Types in Depth

The previous section presented the notion of a permission type intuitively. In this section, we provide a formal definition of PTs and make a comparison with related concepts in the RBAC literature. We also introduce several auxiliary notions that PTs build upon, and show how PTs relate to roles (or role functions).

4.2.1 Formal Definitions

We mentioned object and operation types as the building blocks of a PT. We provide the following definitions:

- Let *OPTYPES* be the set of operation types.
- Let *OBTYPES* be the set of object types.
- Let *PERMTYPES* be the set of permission types. Then

$$PERMTYPES \subseteq OPTYPES \times OBTYPES.$$

- In addition, since each permission type maps to a non-empty set of actual permissions, we can define this mapping algebraically:

  $$\mathcal{P} : \mathit{PERMTYPES} \rightarrow 2^{\mathit{PRMS}},$$

  where for $\forall pt \in \mathit{PERMTYPES}, \mathcal{P}(pt) \neq \emptyset$.

In addition to these basic definitions, we also want to have a convenient notation for speaking about the type of an object and the type of a permission. We accomplish this by introducing the *type-of()* function, with its two varieties:

$$\text{type-of} : \mathit{OBS} \rightarrow \mathit{OBTYPES}$$

$$\text{and type-of} : \mathit{OPS} \rightarrow \mathit{OPTYPES}.$$

4.2.2 Related Work

As mentioned in Section 2.1 where we discussed Enterprise RBAC (ERBAC), we would like to be able to define permissions in a corporation without making assumptions about the platform on which the resources controlled by these permissions will exist. Therefore, Ferraiolo et al. [5, Chapter 11] define the concepts of a *generalized data resource* and a *generalized operation*, which together yield *generalized permissions*, and the latter are then assigned to ERBAC roles. Both generalized data resources and operations reflect the business perspective on resources and operations, respectively. For example [5, p. 249], *deposit accounts* (e.g. savings and checking accounts) and *loan accounts* in a bank would be generalized data resources, whereas *open*, *credit* (withdraw) and *debit* (deposit) would be the generalized operations on, say, a deposit account.

These two concept seem similar to object and operation types, in that they try to generalize away from the raw, system-specific privileges that employees have in the company. However, object and operation types (and hence permission types) are a further step of abstraction, because they are not concerned with any given business perspective, and instead try to group things that are conceptually the same. For instance, using the above example, both deposit accounts and loan accounts, which are different generalized resources, would belong to the same object type `bank account`, as a way to set them apart from other things that could be "accounts", like a `computer account`. We could, although not necessarily, group the *debit* and *credit* operations into the `change balance` operation type, if for example having the (*debit*, *deposit account*) permission implies having the (*credit*, *deposit account*) permission. To use an analogy, permission types are to actual permissions as primitive data types are to actual data in a computer program – an integer (say in Java) could be odd, even, positive, negative, prime or composite, but it is still an integer.

4.3 Object and Operation Taxonomies

In this section, we describe the concept of *object and operation taxonomies* as a way to uniformly define permission types in a corporation.

4.3.1 Motivation

In our discussion of object and operation types, we have not yet mentioned how they are obtained from raw permissions, and in a similar vein – how the *type-of()* function might work. We provide some background here.

In Chapter 1, we talked about the two main approaches to role engineering – top-down role decomposition (the scenario-driven model) and bottom-up role discovery. It was also noted that researchers agree that a hybrid approach should be sought, one that borrows ideas from both techniques. We share that conviction, and recognized that it would be a good idea to collect information about a role from real-life practitioners of that role. In fact, such information could theoretically be useful to bottom-up approaches, by giving "blind" clustering algorithms hints about the structure of the roles they are trying to discover. We shall henceforth refer to practitioners knowledgeable about a certain corporate function (role) as *domain experts.*

The main reason for using domain experts' knowledge is helping us infer role structure, which is a central problem to this thesis. We envisioned asking domain experts about the kinds of permissions (as opposed to the actual individual permissions) they need to do their job, in order to get a very high-level overview of what having that job entails. Example domain expert input would be "I need to make deposits to certain corporate accounts" and "I need to be able to update certain databases", etc. By eliciting role information in this way, we could, in fact, infer the permission types associated with the role, and then map the actual permissions of the domain expert to their respective type. For example, using the above hypothetical human input, we might conclude that two of the relevant permission types (PTs) for the job are $< deposit, corporate - account >$ and $< update, database >$, and we would proceed to mapping the actual permissions on the actual accounts and databases to these PTs. Note that this is a hybrid approach: the data collection part is top-down, because we ask the domain expert to identify their permission needs, which differ based on the tasks involved in the job. The subsequent mapping of actual permissions is a bottom-up process, because we organize them into sets based on their belonging to a particular PT.

The reader might ask: how is this better than the scenario-driven role engineering process? The answer is that in the latter, we need to identify all tasks associated with the job, and for each task – all scenarios that are associated with it, and for each scenario – all the steps for performing the task in that scenario, and the relevant permissions for each step. However, this is too time- and labor-consuming for a DFCE, and besides, by the time top-down role engineering is done, things in the company will already have changed and some roles will be out of date. In our approach, the domain expert does not have to think at such a level of detail, and he or she is given the freedom to describe his or her permission needs very generally, which is easier and also saves time. For these two reasons, it also has the potential of being more complete while requiring fewer domain specialists from the same field (job), because it is more likely for one person to be able to list all types of permissions they need, as opposed to all individual permissions their corporate function necessitates. The latter may require several iterations and the help of several domain specialists.

One problem of collecting information from domain experts is that they may have very different ways to express their permission needs, and very different mental models of what these needs are. Therefore, we wanted to aid the process by providing all domain experts with "standard catalogs" of operation types and object types, which they can then use to provide their expertise in a consistent format. We call these catalogs the *object taxonomy* and the *operation taxonomy.*

4.3.2 Representation of the Taxonomies

We gained a lot of insights about operation and object taxonomies by trying to formally define the *type-of()* function which, given an object would return the type of that object, and in its other form – given an operation it would return the type of that operation.

Object Taxonomy Our original idea about the object taxonomy is that it would be a tree structure that allows arbitrary depth of each subtree. For example, much in the style of an XML document, we would have `Resource` as the root of the taxonomy, and then a branch for, say, `Computer` and another one for `Printer`, where the `Computer` subtree is of depth, say, n and the `Printer` subtree is of depth k. Figure 2 shows the resulting taxonomy. As the reader can see, along the **Computer** branch we have the following hierarchy of depth $n = 3$: **Computer → Lab Computer → Windows PC → Sudi003-moose**.

However, what should the return value of *type-of()* be? Should it be **Lab Computer**? Should it be `Resource`? Instinctively we would like it to return `Computer`, but it is not clear how the function would deal with arbitrary nesting depths (the n and k values above). The problem can be ameliorated a bit by postulating that tree depth should be fixed for the subtrees describing separate resource types, such as `Computer` and `Printer`. One way to do this is illustrated by Figure 3. We designate by `Level 2` the level of the hierarchy that corresponds to a type of resource, e.g. `Computer`, by `Level 1` the level that corresponds to any subtypes of that type, e.g. `Lab Computer` or `Mac`, etc., and by `Level 0` the actual resources. Since there can be arbitrary semantic groupings of resources based on some *grouping factor*, then the taxonomy structure from level 2 to level 0 is not a tree, but actually a directed acyclic graph (DAG).

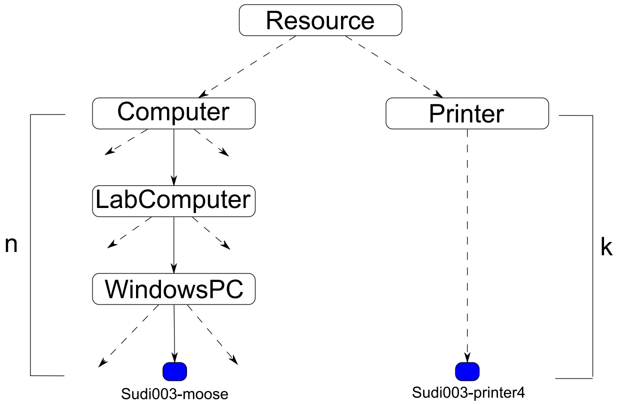

Figure 2: An object hierarchy tree with random-depth subtrees for different object types. In this example, $n \neq k$.

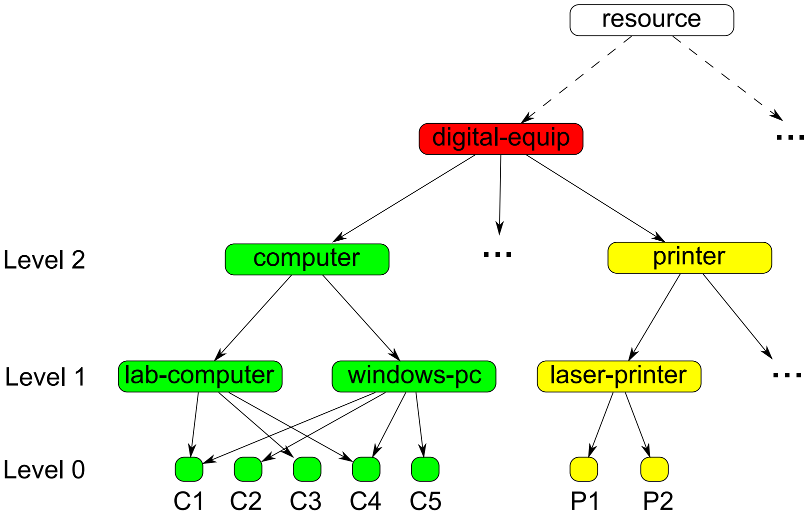

Figure 3: DAG Taxonomy Structure for Levels 2 to 0. Level 2 concepts correspond to object types, Level 1 concepts – to object subtypes, and Level 0 – to actual objects. Note that computers C1 and C4 are classified as both lab computers and Windows PCs, which shows the usefulness of the DAG idea for classifying objects in various meaningful ways.

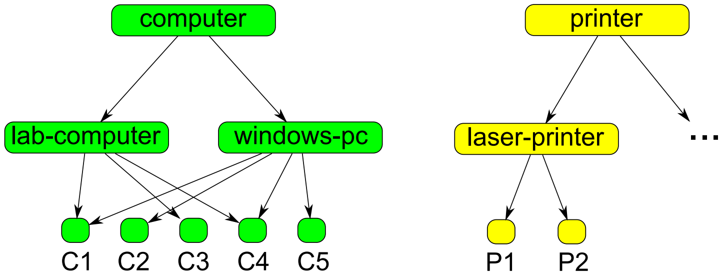

Figure 4: Final Object Taxonomy Representation – "Forest" of DAGs. Each object type has its own DAG.

The problem is not solved though because there is a number of levels between `Resource` and the nodes `Computer` and `Printer`, e.g. the node `Digital Equipment` that branches off into the aforementioned two nodes, so it is not clear how to "instruct" *type-of(Sudi003-moose)* to stop at `Computer` and not return `Digital Equipment` or `Resource`.

These considerations lead us to the final representation scheme for the object taxonomy, illustrated in Figure 4. We have a DAG for every resource type, e.g. `Computer`, which has the three levels discussed above, and no further grouping into super-types. Therefore we end up with a "forest" of DAGs.

A simple algorithm for *type-of()* would then look something like Algorithm 1.

Algorithm 1 *type-of(X: TaxonomyItem)*

```
if is-resource($X$) then
    $Y \leftarrow parents[X].first$
    return $parents[Y].first.description$
else if $parents[X] = \emptyset$ then
    return $X.description$
else
    return $parents[X].first.description$
end if
```

The algorithm is meant to work on each item in the hierarchy which, for any DAG, amounts to the object type, the actual objects, and all intermediate groupings (subtypes) that are of interest. Therefore, by definition, if *type-of()* is evoked on an actual object, it should return a pointer to the object's "grandparent" in the DAG – that is, the object type. If it is evoked on a subtype, it should return a pointer to the subtype's parent, because all subtypes have the object type as their only parent. Finally, if *type-of()* is evoked on an object type, it just needs to return the input. Note that we specifically include subtypes both in the algorithm and in the definition of the taxonomy structure, even though ultimately they are not needed by permission types. This is done to facilitate the task of the domain expert – if they think of their permission needs in terms of access to a particular variety of a resource, we should provide them with varieties to choose from, but still keep track of what overarching type these varieties belong to.

Operation Taxonomy The intuition behind having an operation taxonomy and operation types, respectively, is that the same kind of abstract operation might apply to multiple types of resources. For example, the *Open* operation could be applied to bank accounts, with the semantics of creating a bank account; in Unix, to open a file for writing for the first time is the same as creating the file. Therefore, an operation type indeed maps to an abstract mental model of an operation, such as the human concept of "opening" as a way to create a new instance of something. Another motivation for operation types would be the presence of semantically very similar, but different in spelling, operations. For example we may have the permission $< create, bank\_account\_id >$ on one system and $< open, bank\_account\_id >$ on another, or even *create-account* and *open-account* as alternate names of the above operations. Since they all try to achieve the same thing, i.e. open a new account with the specified account number (ID), it makes sense to group them under the same type of operation.

For the operation taxonomy, we chose a forest-of-trees representation, i.e. each object type is the root of a separate tree, and the leaves of that tree are the operations that belong to the type. Just as with the operation taxonomy, we have a many-to-one mapping between operations and operation types. The choice of depth-one tree versus DAG for the object taxonomy was dictated by two reasons – first of all, since grouping operations semantically is a cognitive task that requires human intervention, it is by default time-consuming and error-prone, and we do not want the extra level of complexity that subgrouping of operations would add. Second, there is no apparent need for such subgrouping, as all we want to say is that a set of operations are synonyms of each other and can be referred to by a generic name.

4.3.3 Building the Taxonomies

In the previous sections we explained how object and operation taxonomies make it possible to use domain expert knowledge to define company-wide permission types. Having all domain experts refer to the same taxonomies makes it more likely that the information they provide about their permission needs has a consistent representation. However, we still need to answer the question of how these taxonomies are built. While this thesis does not provide a definitive answer and a complete procedure for building a taxonomy, we identify two possible approaches to solving the problem.

Distributed Approach One way to view taxonomy creation is as a distributed task, where many people (domain experts) build different parts of the taxonomy. Thus, it is a joint effort where everyone updates a centralized resource (i.e. the taxonomies) and can make use of the portions of it that have already been created. This is good, because certain basic tasks, such as withdrawing from or depositing to bank accounts, are common across many roles, and once represented in the taxonomy they would then be available to all other contributors and would obviate the need for them to redo the representation work. A good idea for the distributed approach, therefore, would be to have an easy and fast way to browse existing object and operation types (or object subtypes) in order to save domain experts time and avoid duplicated taxonomy items.

Below is a sample procedure for building a resource taxonomy. Even though it implies specific implementation techniques, it is included here more as a guide to our intuition about the taxonomy building process, rather than as a complete and correct algorithm. For any uncatalogued resource (object) X:

- Have an alphabetical list of known resource types. Provide user with an expressive search facility so that the user can quickly identify the resource type that X should belong to. If, indeed, a suitable resource type has been found, say RT:
  - Have the same search facility described above, but for subtypes. While a suitable subtype is found, say ST_i , add ST_i to $parents[X]$.
  - If no appropriate subtype is found, prompt user to create a new subtype NT. Add NT to $parents[X]$ and RT to $parents[NT]$.
- If no appropriate resource type has been found, prompt the user to create a new resource type and subtype and update the parent pointers accordingly.

Note that this procedure allows the user to classify a new resource under as many subtypes of a type as semantically possible under the current representation. This makes sense since subtypes can refer to overlapping sets of resources, e.g. `Lab Computer` and `Mac`.

A problem with this procedure is that is has to be carried out for *every* resource in the corporation. The problem is party offset by the fact that different people would be working on different parts of the taxonomy, and can be further offset by implementation decisions such as using what we referred to as an *expressive search facility*, such as having an auto-complete search feature for types and subtypes, which gives the user a list of the possible options as they are typing, or allowing the simultaneous addition of multiple similar resources (e.g. adding all bank accounts with account numbers in a certain range R or set S

by just specifying that range/set, and then the type `bank-account` and the subtype, say, `checking-account`), etc.

The previous paragraph is a motivating example for a more general discussion on the drawbacks of the distributed approach:

- It can potentially be quite time-consuming. This is an issue, because time was also the main argument against the prevailing top-down role engineering methods, so it seems as if we have just moved time complexity from role creation to taxonomy creation.
- It reintroduces the problem of different mental models. The motivation for using operation and object taxonomies was that we wanted permission types to be defined regardless of the differing mental pictures domain specialists may have about their permission needs. However, it seems possible that, especially since certain tasks are shared by many people, we might end up with duplicating taxonomy data because people view these tasks differently. This problem would exist even with a search facility, since the exact way a user formulates things in their mind may quite possibly not be represented yet by the taxonomies, so the search will not return what they are looking for.

However, the approach has some important benefits:

- The fact that many people are contributing to the taxonomies at the same time, ignoring for a moment the problem of representational differences, means that the amount of work per contributor is decreased, and that the number of resources to catalog per contributor is also smaller. In addition, intuitively, the taxonomy building task should become less time-consuming as the taxonomies become more complete by virtue of the combined input from domain experts, so a time bottleneck would exist only at the earliest stages of the process.
- The fact that humans are engaged in the taxonomy building process is a strong indicator that the eventual taxonomies would be semantically sound. That is, even with differing mental models, we end up with human-generated classifications, which are more likely to reflect the intended meaning of the taxonomy than if they were generated by a purely automated method.

Standardization Approach Now we turn our attention to a different taxonomy creation approach. Instead of building taxonomies in a concurrent distributed way, which introduces the problem of dealing with different conceptual model, we may assign most of the taxonomy definition work to a select group of people. This "work group" would define, organization-wide, the resource types and subtypes and the operation types, and then each employee could use the resulting "standard" to classify the resources they individually have. This approach has two main benefits – first, we have conceptual integrity of object and operation types, because they are the product of collaboration between members of a small group. Second, the drudgery of mapping actual resources to these types is distributed among a huge number of people, and they have a unified reference to inform their choice, so the amount of time spent per employee should be quite acceptable.

This approach is akin to role name standardization – the problem of assigning names to RBAC roles consistently across an organization. Efforts have been made to come up with standard catalogs of role names, notably in the healthcare domain [3]. Our standardization approach is an extension to that idea, but applied to object and operation type names.

Although the standardization approach seems promising and can use the insights of related research (role naming), it still requires a group of people to find the time to sit together and do all the type definition work. It is not necessarily clear who these people should be, how much time the process would take, and most importantly – whether they would be available to do it.

4.4 Permission Type Constraints

Section 4.3 described two important preliminaries to permission types (PTs) – namely, the object and operation taxonomy. Here we introduce the concept of *permission type constraints (PTCs)* which are, in a way, the complementary inverse of PTs.

4.4.1 Motivation

While PTs are useful for gleaning information about role structure, they are very general. We need a way to make the translation between a PT in a role and the specific permissions of that PT that are relevant to the role. Furthermore, as we noted in Chapter 3, DFCEs are characterized by dynamic reassignments of duties within a role, and on a more general level – different users in the same role may exercise their permissions differently. We would like PTCs to offer some of this flexibility. Below is a summary for what PTCs need to take into consideration, and it motivates the formal definitions that follow:

- A permission type (PT) can map to permissions across different roles. For example, `(deposit, bank-account)` is a PT that stands for all permissions to deposit money to any type of bank account. Therefore, consultants, payroll officers, accountants, and possibly other roles will utilize different subsets of these permissions. Consequently, PTCs need to be defined for a particular role.
- Trivially, the PTC needs to consider the PT for which it is defined, and finally:

- A PTC needs to consider the individual users in a role, in order to account for the dynamic nature of DFCE roles as summarized above.
- To summarize, if a user U is assigned to role R, and U's permissions for R contain a subset S that belongs to permission type T, then the corresponding PTC will be a mapping from U, T and R to S.

4.4.2 Formal Definition

Let us first define *PTCONSTRAINTS* to be the set of all permission type constraints. Therefore, for $\forall ptc \in PTCONSTRAINTS$, we have the following:

$$ptc : USERS \times PERMTYPES \times ROLES \rightarrow 2^{PRMS}.$$

Using the $\mathcal{P}$-notation we introduced in Section 4.2.1, if $u \in USERS$, $pt \in PERMTYPES$ and $r \in ROLES$, then:

1. $ptc(u, pt, r) \subseteq \mathcal{P}(p)$, or the PTC maps to a subset of the permissions in p;
2. $ptc(u, pt, r) \subseteq assigned\_permissions(r)$, or the permissions to which the PTC maps are assigned to the role r, and
3. $u \in assigned\_users(r)$, or the user u is assigned to role r.

4.5 Putting it All Together

So far, we have introduced the concepts of permission types (along with the prerequisites to their definition) and permission type constraints. Now we are going to show how they can be used together to produce a new definition of a "role". This definition strives to account for complex role structure and the dynamic nature of users' permissions. Finally, we discuss the usefulness of our role model in terms of its constituent entities – PTs and PTCs.

4.5.1 Application to Roles

Figure 5 shows our view of role structure, complete with PTCs. We say that a role r *contains* a permission type p if for each user $u \in assigned\_users(r)$, $\exists ptc(u, p, r) \in PTCONSTRAINTS$. In other words, each *user* assigned to the role has some subset of the permissions of p. Note that we are not saying that the *role* has a subset of the permissions of p – precisely because we are trying to avoid the role-as-set semantics, and because we want to give roles the flexibility needed by DFCEs. Essentially, we are allowing users to

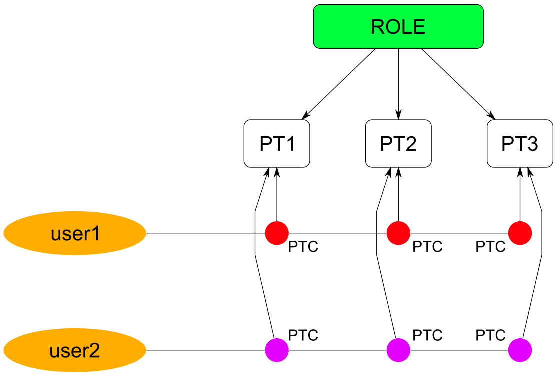

Figure 5: A simple example of a role that contains three permission types and has two users assigned to it. Each user has a PTC for each PT of the role, which allows users to differ in permissions within the same role, and it also allows for certain reassignments of tasks without switching roles or changing the role.

vary along any given permission type that a role contains, while still making the claim that the role *does* contain that permission type. In the *Consultant* example from Chapter 3, the fact that consultants work on bank accounts means that the consultant *role* would contain, in our model, relevant permission types for dealing with bank accounts, such as `(deposit, bank-account)` and `(view-balance, bank-account)`. When a consultant C is reassigned to work on a new account, we only need to change his/her permission type constraints on the relevant permission types (e.g. $ptc(C, (deposit, bank - account), Consultant$ now maps to the newly assigned bank account(s)). We do *not* have to change anything about the consultant *role* in order to reflect the reassignment. Thus, in a way, the PTs contained in a role are a kind of a very high-level overview of that role, and they reflect our intuition, stated earlier, that there is some internal structure to roles.

Also, we can now distinguish between the *user view* and the *role view* of permission assignment. As Figure 5 shows, for any given role that a user is assigned to, the user has a PTC for each PT contained in the role. Thus, two users assigned to the same role no longer have to have the exact same permissions as in traditional RBAC, because they may have different PTCs for the PTs of the role.

From the role's perspective, there is a fixed set of PTs that it contains, and they are applicable to all users assigned to the role. Therefore, a role is no longer a set of permissions, but could be viewed as a set of permission types. Permission type constraints then provide the actual mappings to permissions, in a way that gives these mappings flexibility and makes them more user-specific, while keeping the relation to a concrete role. From now on, we shall refer to the set of PTs contained in a role as the *core* of that role.

The last definition is necessary and useful, because it could be the case that an employee in a certain role, say a consultant or an analyst, needs access to a resource or set of resources that are not represented via PTs in the core of the role. In other words, the needed permissions are not of a permission type that is common to all consultants. There are many ways this situation could arise – one that we have already discussed in Chapter 3 is task reassignment within a role. After a task reassignment, an employee may need access to a type of a resource that is specific to their project only, and not needed by other employees having the same role. From the role's perspective, nothing changes – the core has not changed, and should not change, because we do not want to overprivilege users. Here, having a separate user perspective helps, because we are already keeping track of the PTCs for each user for each role. Therefore, we could add a PTC (or several PTCs) to the particular user's PTC set to encapsulate the new permissions they have gained, and that are specific only to them.

4.5.2 Usefulness and Tradeoffs of the Model

Here, we provide a discussion of some of the tradeoffs inherent in using PTs and PTCs, and analyze their usefulness as role expression concepts.

The conceptual model of roles presented in the previous section is subject to a robustness concern. We are dealing with DFCEs where a single corporate role may be a cover term for many different activities, and we are allowing users to have permissions that are not part of the role core. Won't this model then degrade to each user essentially having their "own role," as defined by their unique mix of core and non-core permission types? The answer to this question depends on which and now many PTs form the core of the role, and how expressive each PT is. A PT is as expressive as the breadth of permissions that it maps to, and so the more general a PT is the more variation it can account for on the role level.

Now we define a metric for the generality of PTs, by focusing on their constituent parts – the object and operation types. Let the *compression factor* of the taxonomy system be defined by the ordered pair (κ_1, κ_2), where

$$\kappa_1 = \frac{\mid OBS \mid}{\mid OBTYPES \mid} \; and \; \kappa_2 = \frac{\mid OPS \mid}{\mid OPTYPES \mid}.$$

Obviously, κ_1 and κ_2 are useful only when they are reasonably big: a small κ_1 would be an indicator that most resources in the organization are extremely heterogeneous and do not fall well into groups, and a small κ_2 would mean that operations on objects do not fall into semantic groups too well, maybe because permissions are highly dependent on applications or objects.

In the context of an investment bank, κ_1 should not be too small, because a lot of the company assets fall into groups such as computers, databases, bank accounts, etc., each of which contains a lot of individual objects. However, κ_2 may be small because it is a known problem that permissions vary greatly across applications [22].

However, we shall use the simplifying assumption, based on emerging corporate practices (as represented in field studies by Dartmouth College researchers), that in the future the definition of application permissions will follow certain guidelines, so that it is easier to group the operation part of the operation-object permission pair. Thus, in theory, the compression factor for an investment bank should be reasonably good. In the Methodology chapter we shall make some empirical measurements to see if actual data corroborate our expectations.

Finally, we need to consider the fact that PTCs provide a lot of flexibility but they also pose some challenges – after all, we have several of them per user, which may be problematic in a big corporation. The next chapter will present our research on using machine learning techniques to automate the management of PTs and PTCs.

5 Application of Learning Approaches

In this chapter we are going to explore the applicability of several machine learning algorithms to the problem of understanding role structure. Section 5.1 discusses some challenges of defining the problem in a rigorous way, and identifies several related learning problems. Section 5.2 presents our thoughts on why certain learning paradigms might not be applicable to the problem space, and Section 5.3 focuses on some promising learning algorithms.

5.1 Roles as a Learning Problem

Here we discuss the difference between learning a role when viewed as an access control function, and learning the structure of a role.

5.1.1 The Role as an Access Control Function

In Section 4.1 we presented the following naïve definition of a role function:

$$role : SUBJECTS \times OPSOBS \rightarrow \{0, 1\},$$

and discussed that the cardinality of the sets of operations and objects, respectively, makes it hard for certain learning algorithms, such as decision-tree induction, to learn such a function. However, there are more substantial problems with this definition. Remember the *Consultant* example from Chapter 3. It was meant to show that there are real-life situations when employees in the same role may need different permissions, and moreover that their permission needs may change quickly over time. In our example, a consultant may work on account A for a few weeks, then on account B for a few more weeks, and so on. Let's assume that our accountant is called John Smith, and that a subject acting on his behalf when accessing accounts is called $jsmith1$ (remember the definition of subject vs. user in Chapter 1). Then, for a few weeks, it will be the case that

$$Consultant(jsmith1, deposit, bank\_account\_A) = 1,$$

but in the next few weeks we will have the situation

$$Consultant(jsmith1, deposit, bank\_account\_B) = 1 \quad (1)$$
$$Consultant(jsmith1, deposit, bank\_account\_A) = 0 \quad (2)$$

The real problem here is the value of the function in (2): it changed between task reassignments for the user John Smith, so in fact we cannot treat it as a function in the strict mathematical sense. If we were to salvage this form of the role function, we would need to add a time argument, but it is not clear how such a function can be even approximated. One reason is that task reassignments don't have to follow a regular, cyclical pattern over time – i.e. the time intervals $t_1, t_2, \cdots, t_n$ that characterize the duration of John Smith's first n assignments could well be, from the point of view of a mathematical series, totally random[6]. Additionally, old permissions can reappear in subsequent assignments, e.g. 5 months from now John Smith may need to work on account A again, so there is hardly a way to predict the permission structure of future assignments. In light of this, the "enhanced" role function looks more like an arbitrary mapping that is updated over time, and not so much like a learnable concept.

In Section 4.1 we talked about object and operation types (and consequently permission types) as a way to make the hypothetical role function amenable to decision-tree (or other) learning algorithms. Looking at the above example, we notice that John Smith's permissions revolve around operations on bank accounts, and of them we focused on the *deposit* operation. The set of permissions of the form $(deposit, bank\_account\_X)$ can be captured by the permission type $(deposit, bank\_account)$. If we recast our initial definition of the role function in terms of object and operation types, we get

$$role : SUBJECTS \times OPTYPES \times OBTYPES \rightarrow \{0, 1\}.$$

Note that using this definition, it is true that across reassignments for John Smith,

$$Consultant(jsmith1, deposit, bank\_account) = 1.$$

Conceptually, this maps to our intended meaning of "John Smith *works on bank accounts*", and it goes back to the idea that permission types capture the essence of a role; they are the "scaffolding" of a role, on which different flavors of the role can be constructed for its different users. However, despite the fact that this definition of the role function behaves like an actual function, it no longer answers the question whether access should be granted or not – it simply confirms that the role contains a certain permission type.

[6] However, one could argue that this sequence is not random, but a product of the dynamics of the organization

Another candidate function that relates to access control is a permission type constraint. Recall that we defined it as:

$$ptc : \mathit{USERS} \times \mathit{PERMTYPES} \times \mathit{ROLES} \rightarrow 2^{\mathit{PRMS}}.$$

That is, if a user U of a role R has a set of permissions S that belong to the permission type T, then $ptc(U, T, R) = S$. Upon closer scrutiny, however, we realize that this function suffers from the same predicament as the naïve role function. Using once again the *Consultant* example above, we notice that for the first few weeks when John Smith is working on account A, we have:

$$ptc(jsmith1, (deposit, bank\_account), Consultant) = \{(deposit, bank\_account\_A)\},$$

but in the next few weeks Mr. Smith is working on his new assignment, where

$$ptc(jsmith1, (deposit, bank\_account), Consultant) = \{(deposit, bank\_account\_B)\}.$$

Thus, in this form, a PTC is in reality not a function, and adding a time parameter would lead to the same concerns that we raised for the role function.

5.1.2 Learning Role Structure

Putting aside for a moment the problem of defining role semantics in terms of an access control function, we turn our attention instead to role *structure*. As we have already pointed out, a role contains a set of permission types, and this containment is “static” in the sense that it does not change for different users and at different times. The only time when we would change a permission type in a role is when the role *itself* has changed, i.e. the functional meaning of that role in the corporation has drifted.

On that note, so far we have touched upon, but not covered in detail how permission types get assigned to roles. Perhaps this assignment is a good area for applying learning techniques. Recall that we talked about using domain experts’ input to infer the permission types for a role. However, as in the consultant or analyst examples, one domain expert may not necessarily represent the permission needs of the others. This might not seem like a problem, because roles contain permission types, and permission types are fairly general – even though different consultants may be working on different bank accounts at any given time, the “work on bank accounts” idea can be expressed by a few PTs, that are then applicable to all consultants. However, remember that we made the distinction between the *core* of a role, e.g. the permission types that apply to everyone, and PTCs that may apply to individual users because of the specifics of their task assignments. For example, all consultants work on bank accounts, but Julia needs access to some web server because of a special project she was assigned by her boss. Thus, if no other consultants need access to web servers, we are not justified in putting the relevant permission types in the *Consultant* role.

Therefore, we may envision using a learning algorithm to determine which permission types form the *core* of a role, so that we can separate that from "special" permission types individual users might need. More precisely, we can train our algorithm on some subset of users from a given role (or job position), whose permissions indicate the permission types they need, and use that to predict, for the remaining users, which of their permissions are "core" and which are "special". With decision-tree learning, we could do that for many roles at the same time. Here are some steps we might need to take:

- Observe that in our discussion of roles in DFCEs we use examples such as *Analyst* and *Consultant*, which are closer to job titles than to traditional RBAC roles. As we discussed in Chapters 2 and 3, these DFCE roles would have to be expressed by a large number of traditional RBAC roles in order to capture the diversity within the job position. In contrast, our approach strives to minimize the number of roles, which is consistent with RBAC pioneers' intention to use the model as a tool for simplifying user and permission management.

- In view of the above, we can use information in the HR database (and possibly other databases), and information from ACLs or other permission management schemes, to come up with training data of the format:

  `job_title, user, operation, object`

  This is, in fact, the format of data we will be using to make some empirical measurements of the efficiency and expressiveness of the permission type model in Chapter 6. For example, we might have:

  `consultant, jsmith1, deposit, bank_account_A`.

- Using the object and operation hierarchies, and treating job titles as roles, each data sample of the above form can be converted into:

  `role, permission_type, 1`,

  where a 1 indicates that this permission type is relevant to the role, by virtue of a user in that role having a permission (object-operation pair) that belongs to the permission type. For extra clarity, note that we have done the substitutions:

$$\begin{aligned} \textit{role} &= \textit{job\_position} \\ \textit{permission\_type} &= (\textit{type-of}(\text{operation}),\ \textit{type-of}(\text{object})) \end{aligned}$$

- Thus, a natural choice for the *role structure function* is the following:

$$\textit{struct} : \textit{ROLES} \times \textit{PERMTYPES} \rightarrow \{0, 1\}.$$

Algorithm 2 Assign-PTCs

```
for $r \in ROLES$ do
    for $u \in assigned\_users(r)$ do
        for $p \in assigned\_perms(u, r)$ do
            $pt = type\text{-}of(p)$
            Add $ptc(u, pt, r)$ to user $u$'s PTC set
        end for
    end for
end for
```

If we can learn this function, then we can also automatically obtain, for each user, the permission type constraints for core (or other) PTs in any role the user might have (Algorithm 2).

Note that the learning algorithm takes time proportional to $\mid PRMS \mid$, because it is run for every permission that every user in every role has. Note also these notational details:

- We use $assigned\_perms(u, r)$ to refer to the set of permissions that the user u has within the role r. This is different from the traditional RBAC relation $assigned\_permissions(r)$, which returns the set of permissions that *all* users in the role share. As we already pointed out, it is not always the case that people in the same role have the same permissions, so we need to take individual users into account when figuring out how the role is reflected in each user's permission needs.
- We use the shortcut $type\text{-}of(p : PRMS)$ to refer to the permission type pt for a permission p. This is equivalent to calling $type\text{-}of()$ on the permission's operation and object, and using the resulting operation and object type to create a permission type.

5.1.3 The Challenge

The role model that we described in Chapter 4 turned out to make it hard to express roles as access control functions. However, a more serious criticism is that it seems to reintroduce the problem of per-user permission management – each user has several PTCs per role that need to be created and updated over time. Therefore, we need to investigate ways to automate the assignment of PTCs to users, regardless of whether that involves using learning techniques. Chapter 7, Future Research, discusses several promising ideas about how to achieve this automation.

From a more philosophical point of view, note that it may be fundamentally impossible to have dynamic roles without using some mechanism to map the role "template" to the permissions individual users should have. PTCs are one such mechanism, and the number of

PTCs we need to maintain is proportional to the number of users, or $\Theta(n)$ for $n =\mid USERS \mid$. Maybe there are more economical ways to achieve the functionality of PTCs, which would require $O(\sigma(n))$ elements for $\sigma(n) << n$. If we define the *complexity* of a dynamic role to be the number of concepts that we need to have in order to map the role to the different permissions of different users, it would be very useful to come up with a rigorous mathematical model for this complexity, and find its lower bounds (if they exist).

5.2 Learning Approaches That Might Not Work

Here we present our reflections on certain machine learning models and techniques, which we found difficult to apply to our problem space.

5.2.1 Computational Machine Learning Theory

As the field of machine learning was growing, researchers started asking themselves fundamental questions about learning algorithms – what sort of concepts can we hope to be able to learn? What claims can we make about the accuracy of learning algorithms (error rate) and the probability that they perform well on test data? All these questions lead to the development of the *computational machine learning theory* (COLT) subfield, which tried to come up with a formal model that would allow us to answer the above questions.

A prominent learning paradigm within COLT is Probably Approximately Correct Learning (PAC learning) [8, 13]. In PAC learning, we are trying to inductively learn an unknown *target function*, and after seeing a number of training examples we come up with a hypothesis drawn from a specific *hypothesis space* H. Without going into too much detail about the technical particulars of the PAC model, we point out several of its characteristic traits below:

- We are trying to learn a concept c, drawn from a concept class C, which is defined over a set of instances X. For example, the set instances could be all people, and a concept could be "people of medium build" [8].
- The learning algorithm L has to produce a hypothesis h which, with high probability, has a small error of classifying new examples, i.e. is *approximately correct*. The notation ϵ is used for the error rate and δ – for the probability of failure to produce an approximately correct hypothesis, where $0 < \epsilon, \delta < 1/2$.
- The algorithm L has to output a hypothesis with the properties described above in time that is polynomial in $1/\epsilon$, $1/\delta$ and a couple of other parameters, including the size of the target concept c (defined by the representation of the concept class). These quantities can also be used to determine the number of training examples that we

> would need to be sure that L will output a hypothesis which, with probability $(1 - \delta)$, will have an error rate of ϵ.

Despite the nice theoretical results that are obtained by the PAC learning model, such as an estimate of the number of training examples for PAC-learnable concepts, and proofs that certain classes of functions are PAC-learnable, it has two major drawbacks with respect to our problem space. First, it implies that we know the concept class from which our target function is drawn. In the case of roles, we don't even know for sure if they are functions, let alone learnable ones, so it would be very hard to *a priori* make any assumptions about them. Second, the definition above assumes that there exists a hypothesis in H that can be made arbitrarily close to the target function (or else that function is not PAC-learnable). In our case, there is no intuition or formal evidence to support that belief.

Moreover, PAC-learning is usually illustrated in the context of Boolean and numeric concepts, such as learning rectangles of certain dimensions [8], Boolean conjunctions [8, 13], etc. An extension of the rectangle example is used to learn the concept of "people of medium build" mentioned above, where the height and weight of a person are numeric quantities and can be used akin to the X and Y axes for the original example. PAC-learning, however, does not seem applicable to complex symbolic data. For instance, objects, operations and their associated types do not have an obvious numeric representation, so there is no clear way to define the "axes" of our instance space. Chapter 7 presents our thoughts on how this problem might be overcome.

5.2.2 Instance-Based Learning

An alternative to inductive learning techniques (such as decision-tree learning, PAC-learning, etc.) is *instance-based learning* [13, 19, 26]. It is conceptually simple – you just store the training examples, and then a test example is evaluated for similarity with the stored examples. The benefit of that approach is that we do not commit to a particular approximation to the target function until we have to classify a new instance. This allows for learning possibly very complex functions, which makes this learning approach appealing for our problem.

One family of instance-based learning algorithms is *nearest-neighbor algorithms*, which represent instances as points in an n-dimensional Euclidean space [13]. In k-nearest-neighbor learning, we classify a new instance based on the majority classification of the k stored instances closest to it (by taking the Euclidean distance). A major problem for this algorithm in our domain of interest is that, as we discussed in the context of PAC-learning, it is not clear how we can numerically represent the instance attributes relevant to the role function (be it operations and objects or their types, or some extra information). Therefore, we cannot represent instances as points in n-dimensional space. Another method within instance-based learning that relies on the same representation for instances and would therefore not be applicable to our problem space is *locally weighted regression* [13].

Finally, *case-based reasoning* [13] is an instance-based method that shares the useful properties of the other two, but it lifts the requirement that instances need to be represented as points in a plane. Instead, it allows us to use complex symbolic concepts, which is closer to the nature of roles and role functions. Case-based reasoning has been applied to areas such as designing mechanical devices based on stored previous designs, adjudicating new legal cases based on previous rulings, and complex scheduling problems [13]. The distance metric here is more complicated than mere Euclidean distance, and highly dependent on domain-specific knowledge – for example, if we represent stored problems as graphs of subproblems, our similarity measure could be graph isomorphism. However, case-based reasoning and PAC-learning share a similar problem in relation to role functions. A running theme in the above examples where case-based reasoning has been applied is that we know something about the *structure* of the stored instances, and that is why we look for structural or sub-structural similarities when classifying new instances. But as we pointed out in the discussion of PAC-learning, we don't know anything about the target role function, so we cannot provide its "structure" as part of the training data. In essence, the problem with both approaches is that they assume we know certain properties of the target concept, when in reality we do not.

5.3 Learning Approaches That Might Work

In the current research, we have not investigated the applicability of all popular learning methods – for example, we have not looked into using artificial neural networks or genetic algorithms. We have, however, researched decision-tree learning, and we believe it might be applicable to our problem space. Some key advantages of decision-tree learning are as follows:

- It makes no assumptions that the target function is drawn from a particular class. This is good, because as we saw, requiring us to make such assumptions when we really don't have a solid justification for it resulted in several learning algorithms being unsuitable for our needs.
- It handles noise in the data well, because the resulting trees use the most relevant instance attributes, and noisy attribute values do not have a significant weight. We haven't addressed the issue of noise much in the context of roles and permissions, but we mentioned at the very beginning of the thesis that companies often copy over permissions when a new employee is hired to take the place of an old one, and other practices exist which may result in over-privileging some users. It is exactly these superfluous permissions that constitute "noise" in the data, as they should, ideally, not be present in the user's permission set. Traditional PAC-learning, in contrast, expects that there is no noise in the data [13, 8].
- Most of the computation takes place when the decision tree is being constructed, and

classifying new instances is fast, because it consists of a small number of attribute tests. In contrast, most of the computation in instance-based learning occurs at classification time, which is also the reason for its nice features. However, since we would ideally want to use role functions for access control, and that means classifying access requests, this would be a severe bottleneck, as access requests are many and frequent in a DFCE.

6 Methodology and Experimentation

In this chapter we present the measurements we made using sample data about the permissions of users in several job positions. The data was generated by Sara Sinclair, a researcher in the Dartmouth College PKI Lab, based on her experience with actual permissions in investment banks.

6.1 Sample Data

Here we are going to discuss what our sample data means, and how it is represented in the data file that appears in Appendix A.

6.1.1 Explanation of the Data

Our data is taken from the investment banking domain. We have five basic job positions – *Firm Vice President of Funds*, *Strategic Positioner (Investment Manager)*, *Operations Manager (Tech + HR)*, *Compliance/Legal Division Manager*, and each of these has an *Administrative Assistant*. The data file entries consist of permissions belonging to users assigned to one of these five job positions. The permissions are related to one of the following activities – reading/sending email, logging in to computers, or modifying electronic calendars. The following rules apply:

- All people can send and read their own email, reflected by the permissions `sendEmail` and `readEmail` on their personal accounts (or, for short, we say that a user has `readEmail` for their own email account).
- Administrative assistants can read their charge's email, so they have `readEmail` for their charge's account.
- All people have `adminLogin` for their own machines.
- Administrative assistants have `userLogin` for their charges' desktop machine.

- All people have `modifyCalendar` for their own calendar.
- Administrative assistants have `modifyCalendar` for their charges.

6.1.2 Format of the Data

The data entries follow the format:

```
jobTitle, user, operation, object,
```

where `jobTitle` can be one of `vp` for vice president, `asst` for administrative assistant, `compleg` for a compliance/legal division manager, `sp[-domestic, -mixed, -foreign]` for a strategic positioner, and `oper` for operations manager. Object names are identical to user names when the operation is `sendEmail`, `readEmail` or `modifyCalendar` – in the former two cases we interpret the object name as the email account name, and in the latter case: as the calendar account name. When the object name is of the form `desk-username` it refers to the desktop machine of the user, and when it is of the form `lap-username`, it refers to the user's laptop. Finally, there are 70 data samples in the data file.

6.2 Permission Type Metrics

We created and used a simple software component[7] to process the data. The software component allowed for quickly building the object and operation taxonomies, and once taxonomy creation was completed, statistics were generated about the number and distribution of permission types, as well as the compression factors for the taxonomies. Table 1 summarizes the input data before any processing:

Job Positions	5
Users	12
Operations	5
Objects	48

Table 1: Summary of Input Data

The new information here is that we have 12 users and 48 objects. Note that the 70 permissions are defined on only 48 unique objects, and that the number of actual permissions is only about 29% of all operation-object pairs (240).

[7] Available as a RAR archive at http://www.cs.dartmouth.edu/ ruslan/FancyRBAC.rar

Table 2 presents the statistics we obtained from the software component, which can also be found in Appendix B.

Object Taxonomy Compression Factor	16.0
Operation Taxonomy Compression Factor	1.25
Created Object Types	3
Created Operation Types	4
Created Permission Types	4
Permission Type/Taxonomy Size Ratio	0.33
Permission Type/Role Ratio	0.8
Users per Role	2.4

Table 2: Collected Statistics from Sample Data

It should be noted that the Permission Type/Taxonomy Size Ratio refers to

$$\frac{| PERMTYPES |}{| OPTYPES | \cdot | OBTYPES |},$$

and that the Permission Type/Role Ratio refers to $\frac{|PERMTYPES|}{|ROLES|}$. Having this data, we can make the following observations:

- As we were hoping, the compression factor for the object taxonomy is high – we have managed to reduce 48 objects to three generic types. Appendix B shows the object taxonomy we constructed: it recognizes the types computer (`computer`), email account (`email-acct`) and calendar account (`cal-acct`). The two account types are broken down into subtypes according to who can use them, and the computer type is broken down into the subtypes `laptop` and `desktop`. Here, we see the usefulness of the DAG structure for the object taxonomy – a single email account, for example, can be used both by a manager and his/her administrative assistant, so it would get classified under both the `vp` subtype and the `asst` subtype.

- On the other hand, the compression factor for the operation taxonomy is low – there are four operation types for only 6 operations. This is mostly due to the structure of our sample data – we consider a very small set of operations, and they act on heterogeneous resources. However, it might indicate a trend in a larger data sample – we were still able to obtain a high compression factor for objects, and this makes sense because the same operation (say `sendMail`) can be applied to many objects (all the email accounts).

- The Permission Type/Role Ratio as defined above confirms our observation that permission types may span role – the value of 0.8 indicates that, overall, there are fewer permission types than roles. In fact, our sample data is an extreme example in that *all* permission types are shared by *all* roles. As a consequence, the average number of permission types that a role contains is 4.

- The Permission Type/Taxonomy Size Ratio as defined above is similar to the ratio between actual permissions and possible operation-object pairs: 33% and 29%, respectively. One way to interpret this is that, despite being more general than actual permissions, permission types do not introduce meaningless combinations – i.e., we would not get a permission type for a "bogus" permission like (modifyCalendar, deskvpino01) which makes no sense. On the other hand, permission types may imply permissions that don't *yet* exist, but *would* make sense – if the vice president purchases a new laptop, his/her login permissions will have been covered by the `(login, computer)` permission type.

Finally, because in our data each user's permission set is very small, we see that each permission type constraint would map to one or two permissions, and this is most probably not indicative of what a larger sample might reveal. However, the usefulness of our role structure model is still apparent here – even if all administrative assistants were reassigned to a different manager out of the ones we have seen, the assistant *role* will remain unchanged, because the permission types are still relevant to the new reassignment – only the PTCs would change (e.g. now, Alice needs the permission to read Bob's email and not Charlie's email). Traditional RBAC would have handled that by having several administrative assistant roles (or one parameterized administrative assistant role with several possible instantiations), and reassigning users to a different administrative assistant role (or instantiation thereof). Even though some role reassignments can be automated by integrating traditional RBAC with rules [11], the problem is the need to define several very similar roles for the same job function – something that would decrease the advantage of RBAC as a management simplification tool on the large scale. Chapter 7 will look at some possible ways to automate PTC reassignment, which would have the advantage of achieving the results of automatic role reassignment (i.e. you get the new permissions), but without actually performing a role reassignment or creating several related roles.

6.3 Learning Role Structure

Chapter 5 introduced the *role structure function*, which returned true if a role contains a permission type, and false otherwise. As mentioned in that chapter, we could convert our sample data into a format suitable for learning this function:

```
jobTitle, permission_type, 1 (or 0).
```

We used our software to do exactly this, and a part of the result is provided in Appendix C. Unfortunately, we cannot use this data for obtaining a decision tree, because in fact all roles contain the exact same set of permission types, and we have no negative examples (e.g. a permission that a user should *not* have, and respectively its permission type which might not be appropriate for inclusion in the role). However, we still wanted to see whether decision tree learning would cope with noise in the data. To that end, we did the following experiment:

- We split the 70 converted samples into a training set of 36 samples and a test set of 34 samples. The split was done approximately half-in-half, i.e. we added, randomly, half of the converted samples for each role to the test set and to the training set.
- We decided to use the See5 decision-tree classifier for Windows, based on the C4.5 algorithm by Ross Quinlan, the original inventor of decision-tree learning algorithms [17, 18]. We then formatted the training and test data for use by See5.
- We introduced approximately 10% of noise in both the training and test examples, by flipping the value of the `hasPT` attribute from 1 (the default) to 0. We were essentially trying to trick the algorithm that a certain role should not "have" (e.g. contain) a particular PT.
- The result, shown in Appendix D, confirms that the decision tree algorithm does well on the test data, misclassifying only 8.8% of the test samples. The example is a little artificial since the decision tree essentially consists of a test on the `hasPT attribute`, and the error rate corresponds to the noise we introduced. However, it is still a valid classifier, and it still allows for noise in the data.

6.4 Conclusions

Obviously, there is much more experimentation to be done before we can say conclusively whether the role structure model we introduced in Chapter 4 is indeed viable. However, we have seen some early signs of its usefulness, such as a high compression factor for objects, a small set of permission types that serves the purposes of many roles, and the possibility to reassign users without changing their roles. This is an indicator that our approach of separating the role "template" from the mechanisms that instantiate it for each user has practical value, and it brings us closer to a definition of roles that would allow RBAC to be deployed in DFCEs.

Furthermore, the permission type concept made it possible to experiment with decision-tree learning – without it we would have had an "object" instance attribute with a branching factor of 48, which is much higher than either then number of users or operations (the other relevant attributes). We also showed that decision-tree learning deals well with certain levels of noise in the training data, which makes it a promising candidate for future research into learning roles.

What we really need to do in the future is obtain metrics for datasets with multiple users, which have different responsibilities that involve permissions for a diverse set of objects. That would shed light on several important questions, such as: how useful are permission type constraints? How similar do roles become as a consequence of being expressed via permission types and not actual permissions? As we mentioned, in our sample data all roles looked essentially the same, because they all contained the same permission types. Is that the case in general?

7 Future Research

In this concluding chapter of the thesis we talk about some promising areas of research, which might provide answers to fundamental questions we have touched on. For example, can roles be learned and easily maintained in the real world? Do roles have an underlying structure and if so, can we discover it? How can we account for the dynamism of roles in DFCEs without incurring the overhead of managing individual users' permissions? What are some tradeoffs between role expressiveness and ease of management? The answers to these questions are important for several reasons:

- They would show whether it is fundamentally possible to use RBAC, and roles in particular, in the context of the challenges presented by DFCEs. This would be a very satisfactory result, because RBAC has been successful in significantly simplifying the management of permissions in the domains to which it has been already applied [15]. Besides, it has a sizable body of researchers, which would ensure a continuous flow of innovation and improvements.
- However, it may also turn out that the role concept is not expressive enough to meet the demands of DFCEs. This realization would possibly lead to the research and development of new access control models.

The chapter is organized as follows: Section 7.1 will address some issues with using PTs and PTCs as defined in Chapter 4, and Section 7.2 will look at some alternative definitions of the role function (and other auxiliary ones) and give some suggestions about how to adapt role functions for PAC-learning and instance-based learning.

7.1 Role Housekeeping

The role organization model we proposed in Chapter 4 is an attempt to come up with a framework for roles, which would enable us to use the very concept of a role in DFCEs. Thus, we focus on the *role-based* dimension of RBAC, and Section 7.1.1 will elaborate on the role idea. In Sections 7.1.2 and 7.1.3, we turn our attention to the *access control* dimension of RBAC, in an attempt to reconcile the role model we developed with real-world access control concerns.

7.1.1 Role Classes

In Chapter 4 we talked about the *user perspective* and the *role perspective* of permissions. The user perspective was defined as a set of permission type constraints, which partitioned the user's permission set (within a role) into subsets based on their belonging to a permission type. So far the user perspective seems very mechanistic – it is merely a set of mappings from abstract constructs (PTs) to actual entities (permissions). However, we suspect that there is more meaning behind these mappings. As supported by evidence collected by researchers at the Dartmouth College PKI Lab, employees may need certain permissions only in specific circumstances. For instance, an employee may be acting as a back-up for someone else for some duration of time (e.g. two weeks). As a further example , a doctor in a hospital may need access to a patient's medical record in case of an emergency, even if the patient is not theirs, but a critical intervention is needed to save the patient's life.

The first example illustrates an important point: some permissions may be only rarely needed, and it would be nice to have a way of incorporating that information in our role model. The model currently does not give us the ability to say things like "user X needs permission p, but only occasionally," because permission type constraints do not make any distinctions between their constituent permissions. In the second example, we see another real-life concern – even if an employee does not have a certain permission, there are extreme circumstances when the employee might have to use that permission. A common example is doctors in hospitals who might need to act fast in order to save someone's life, or even employees in an investment bank who might need to temporarily assume someone else's duties in their absence, so that money or other assets are not lost. The difference between the two examples is that in the first one we have rarely used permissions, and in the second one we have permissions that have not been assigned to the user (or user's role), but might be needed sometimes. But both cases suggest we might need to define different *classes* of roles, in order to capture different modes of using permissions. Some classes of roles that we have considered are:

- **Basic Roles** – roles that capture the basic duties of employees in an organization.
- **Temporary Roles** – roles that an employee may be assigned to only sometimes, such

as back-ups.

- **Break-Glass Roles** – roles that an employee should be able to have in case of an emergency, for a suitable definition of an emergency. This relates to the doctor example.

The reader might recall from Chapter 2 that the purpose of Temporal RBAC is exactly to capture the *time* dimension of users' assignments to roles, i.e. the idea that a user should be assigned to a role only at certain times. However, TRBAC achieves that by using *periodic expressions*, i.e. things like "every Friday from 9am to 3pm". In the back-up example, unless the need for a back-up is known to be periodic, we cannot directly use this feature of TRBAC. Another concept in TRBAC is that of *role triggers*, for example:

$$\text{enable, } R_0 \rightarrow \text{disable, } R_1,$$

which means that whenever role R_0 is enabled, role R_1 is disabled, i.e. even users assigned to that role cannot activate it in their current session. Thus, we might try to recast the back-up problem as having two roles: `X` and `backup-X`, for which the following role triggers exist:

$$\begin{array}{l}\text{disable, X} \rightarrow \text{enable, backup-X}\\ \text{enable, X} \rightarrow \text{disable, backup-X}\end{array}$$

However, this assumes that in addition to defining the traditional RBAC roles (which we know is time-consuming), we need to define back-up roles too, because back-ups may need only a subset of the permissions of the regular role. Moreover, we need to manually enable and disable the regular role each time a back-up is needed, which might be quite tedious.

Notice, however, that TRBAC may be relevant to the doctor example – TRBAC supports *run-time requests*, i.e. directives of the kind `enable, R` or `disable, R` regardless of the current role triggers and the state of the system. The seminal paper on TRBAC by Bertino et al. [2] provides an example with the `emergency-doctor` role, which is activated via a run-time request, which matches our intuition that emergencies have to be acted upon immediately. It would be worthwhile to investigate the overhead of defining emergency (or, in our nomenclature, break-glass) roles versus their usefulness in the domain of DFCEs. On one hand, we would not want uncontrollable use of break-glass roles, on the other hand, we want to ensure than an employee needing a break-glass role does not have to wait for the help desk officers for too long. This implies the existence of a sound auditing system. We believe that it would be a good idea to have devoted personnel for assigning and de-assigning break-glass roles, because that way we can offload the effort from the help desk, while providing tracking capabilities for who requested what break-glass role.

Going back to our discussion of the user perspective of permissions and the *core* of a role, it would be interesting to find out whether certain permission types for a role map to one

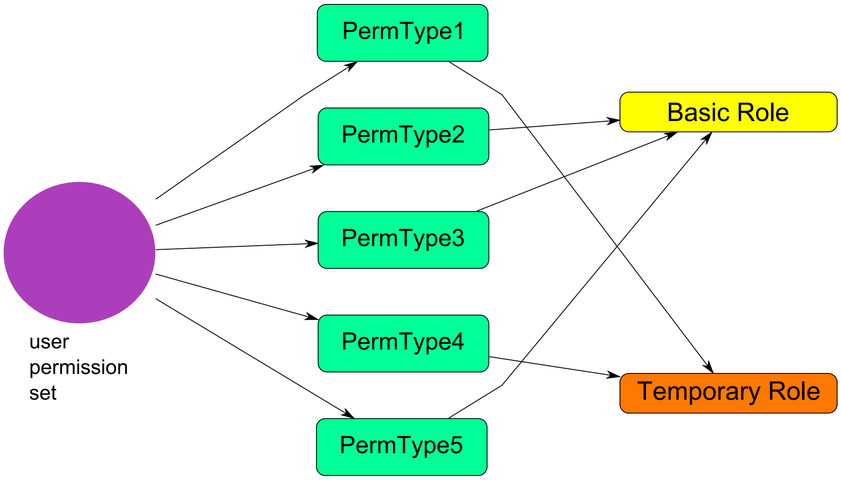

Figure 6: Hybrid Role Discovery – the object and operation taxonomies make it possible to map the user's permissions to permission types, and then the knowledge we have obtained about role structure from the role structure function can be used to map permission types to roles. Finally, knowing the relationship between PTs and role classes can tell us *what* kind of roles the PTs form.

of the role classes we talked about (treating permission types as something of a "sub-role"). For example, the *role structure function* introduced in Chapter 5 tries to discover which PTs constitute the core of a role. The learned PTs may well correspond to the basic duties of the users assigned to that role, so they may be referred to collectively as a *basic role*. Then we are left with permission types specific for individual users, which may correspond to *temporary roles* – either a special assignment, or an activity that the user does not really need to perform often. Discovering mappings between permission types and role classes may make it possible to infer the roles of a user just by looking at the user's permission set (assuming object and operation taxonomies in place), which traditionally is the domain of bottom-up role-clustering approaches. However, unlike blind clustering, we would be using important information about what the permission groupings mean. Schematically, this is represented in Figure 6.

7.1.2 Rules and Context Constraints

Chapter 5 left the question open of how we manage efficiently the numerous permission type constraints per user. Answering this question is crucial, because otherwise it is not clear how our model is *role-based* any more, since permission management is now done per user. One place to start research on this issue is Rule-Based RBAC (RB-RBAC) [11]. RB-RBAC automates the assignment of roles per users, by utilizing the HR and other relevant databases. Therefore, when a user's entry in the database changes, the user's roles are automatically recomputed with the help of *rules*. Rules have a *left-hand side*, which specify conditions on HR database attribute values (for example `If Job-Title = Accountant`), and a *right-hand side* which specifies which role the user should be assigned to.

It would be very useful to see if this rule-based approach can be extended to permission type constraints. In particular, we are curious whether we could devise rules that, instead of taking attribute values from the HR database, use the values of *context attributes* as defined in Chapter 2 where we discussed context constraints. This makes sense, because the times when we need to change the PTCs for a user are the times when this user is reassigned to a new task, for example, and that is accompanied by a change of the "context" in which the user exercises their permissions. Therefore, the left-hand side of our rules would test certain context attributes, and the right-hand side would assign an appropriate PTC to the user. Rules should also take into account the user for whom the PTC is to be calculated. Below are several observations we can make about these rules:

- Rules seem to only make sense for the core PTs of a role, since we cannot predict what special assignments individual users might be given within the role. Therefore, the number of rules for a role is proportional to the number of users and the number of core PTs. Note that even though rules are defined for users and not roles, which raises the concerns we pointed out at the beginning of this section, once defined they subsequently should take care of updating the user's PTCs. Thus, the initial effort would pay in the long run.
- Rules are different from context constraints, because the latter are used to limit the use of one permission, while the former are used to assign permissions. In fact, rules would be used in conjunction with context constraints: rules will produce the relevant permissions per user, and context constraints will control the access to these permissions (as is their intended use).

If rules like the above can indeed be defined, then a generic algorithm for a role function would be Algorithm 3.

The check_context_constraints() function is taken from [23]. Note that this algorithm assumes that at the time the function is called, the rules have already computed the PTCs for the user. By definition, a user has a permission for a certain role if that permission appears in a PTC the user has for the role, and this is exactly what the algorithm is looking for. Once the PTC for the permission has been found, we know that the user should have this permission, so the next step is to check whether any context constraints apply. If they do, and are satisfied, then the access request should be granted. In all other cases it should be rejected.

Algorithm 3 $role(s : SUBJECTS, op : OPS, ob : OBS)$ returns true/false

```
user = subject_user(s)
PT = MAKE_PT(op, ob)
PTC = MAKE_PTC(user, PT, role)
if PTC ∈ ASSIGNED_PTCS(user, PT, role) then
    if check_context_constraints(s, op, ob) then
        return true
    end if
end if
return false
```

7.2 Role Functions and Learning

In Chapter 5 we tried to define a learnable role function. However, as we saw, this is a challenging task. Section 7.2.1 elaborates on the issue of adding time as a parameter to the role function, and Section 7.2.2 returns to the topic of expressing instances of the role learning problem as numeric quantities.

7.2.1 Time as a Parameter

One major difference between traditional RBAC roles and DFCE roles is the concept of change over time. All the definitions of RBAC entities assume that change over time is gradual. That is why the set-of-permissions view of roles works – the underlying permission set does not change often. In fact, it is recommended that the whole role engineering process for RBAC be completely redone once the amount of change has passed a certain threshold [5, Ch. 10]. This is a big reason for not adopting RBAC in DFCEs – mergers and acquisitions are frequent, and with them come changes in tasks and the overall resource pool (Chapter 3), so all the role engineering work would have to be repeated every time. The general problem with DFCEs, as pointed out throughout this thesis, is that change, either through task reassignments or other processes, is very rapid. Thus, time becomes an essential argument of any function we would like to define over roles (except the mapping from roles to PTs). Let us consider, for example, how the *assigned_permissions* relation of traditional RBAC needs to "evolve" to reflect the realities we are dealing with:

1. In RBAC, this is defined as $assigned\_permissions : ROLES \rightarrow 2^{PRMS}$. Since users are assigned to roles, by virtue of this relation we require all users of a given role to have the same exact permissions. We have already discussed why that does not work, so we need to include users into consideration. This yields:
2. $assigned\_permissions : USERS \times ROLES \rightarrow 2^{PRMS}$, where the first argument is restricted to $assigned\_users(r : ROLES)$, and r is the second argument. Now users

are allowed to have different permissions within the same role, which works for roles like *Consultant* as we have seen. But we also know that employees are frequently reassigned to new tasks, so the output of this function would only be valid for the duration of one such reassignment. Therefore, we need to take into account *when* we are asking the question, or:

3. $assigned\_permissions : USERS \times ROLES \times TIME \rightarrow 2^{PRMS}$. Now we are closer to reality, but we have ended up with a pretty complicated function.

A possible line of research using time-aware role-related functions would be extracting usage patterns for individual users. We cannot say for sure that task reassignments are random – they depend on the corporation's dynamics, and could be affected by many factors such as season (and therefore volume of work), stagnation in the financial markets, etc. It could be even simpler – a history of previous permission usage for a user may prove to predict future permission needs well. By seeing which permission types are encountered consistently over time in a user's access history, we may glean information about the structure of a particular role the user has, and compare that with, say, the output of the role structure function.

7.2.2 Numerical Representations

As we saw in Chapter 5, PAC-learning and instance-based learning techniques are hard to apply to roles and role functions, because the latter are complex symbolic data, and the learning algorithms work on numeric or Boolean data. It would be worthwhile to explore whether we can translate between the symbolic representation of our data and a numeric representation.

One idea is to use *utility functions* [19] for, say, object and operation types, in order to obtain a numerical score for them. This would allow us to represent the space of permission types as 2-dimensional Euclidean space, which would make learning problems defined on permission types tractable for nearest-neighbor learning algorithms, as well as PAC-learning. Utility functions can take into account many attributes of the concepts they are applied to, which allows for complicated representation schemes for the utility value of object and operation types.

We have not had an opportunity to conduct research in this area, but one general idea we had is that the utility of both object and operation types is inversely proportional to the risk of misusing the underlying objects and operations. What this means is that, say for object types, the utility of the object type decreases as the risk of using an object from that type increases. Thus, the type `Printer` would have greater utility than `Bank-Account`. For operations, a read-like operation would have higher utility than a write-like operation, because the latter can be used to modify or destroy important data.

References

[1] Ali E. Abdallah and Etienne J. Khayat. A Formal Model for Parameterized Role-Based Access Control. In *Formal Aspects in Security and Trust*, pages 233–246, 2004.

[2] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A Temporal Role-Based Access Control Model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001.

[3] Edward J. Coyne and John M. Davis. *Role Engineering for Enterprise Security Management*. Artech House, 2008.

[4] David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Control. In *Proceedings of the 15th NIST-NSA National (USA) Computer Security Conference*, pages 554–563, 1992.

[5] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Information Security and Privacy Series. Artech House, 2 edition, 2007.

[6] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[7] J.B.D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A Generalized Temporal Role-Based Access Control Model. *Knowledge and Data Engineering, IEEE Transactions on*, 17(1):4–23, Jan. 2005.

[8] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.

[9] A. Kern. Advanced Features for Enterprise-Wide Role-Based Access Control. *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 333–342, 2002.

[10] Axel Kern, Martin Kuhlmann, Andreas Schaad, and Jonathan Moffett. Observations on the Role Life-Cycle in the Context of Enterprise Security Management. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 43–51, New York, NY, USA, 2002. ACM.

[11] Axel Kern and Claudia Walhorn. Rule Support For Role-Based Access Control. In *SACMAT '05: Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, pages 130–138, New York, NY, USA, 2005. ACM.

[12] Martin Kuhlmann, Dalia Shohat, and Gerhard Schimpf. Role Mining - Revealing Business Roles for Security Administration Using Data Mining Technology. In *SACMAT '03: Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, pages 179–186, New York, NY, USA, 2003. ACM.

[13] Tom M. Mitchell. *Machine Learing*. McGraw-Hill, 1997.

[14] Gustaf Neumann and Mark Strembeck. A Scenario-Driven Role Engineering Process for Functional RBAC Roles. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 33–42, New York, NY, USA, 2002. ACM.

[15] National Institute of Standards and Technology. RBAC Case Studies website: http://csrc.nist.gov/groups/sns/rbac/case_studies.html.

[16] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[17] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[18] RuleQuest Research and See5 Information Website. http://www.rulequest.com/see5-info.html.

[19] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[20] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *RBAC '00: Proceedings of the Fifth ACM Workshop on Role-Based Access Control*, pages 47–63, New York, NY, USA, 2000. ACM.

[21] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, 1996.

[22] Sara Sinclair, Sean W. Smith, Stephanie Trudeau, M. Eric Johnson, and Anthony Portera. Information Risk in Financial Institutions: Field Study and Research Roadmap. In *Enterprise Applications and Services in the Finance Industry*, pages 165–180. Springer Berlin Heidelberg, 2008.

[23] Mark Strembeck and Gustaf Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security*, 7(3):392–427, 2004.

[24] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The Role Mining Problem: Finding a Minimal Descriptive Set of Roles. In *SACMAT '07: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 175–184, New York, NY, USA, 2007. ACM.

[25] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. Roleminer: Mining Roles Using Subset Enumeration. In *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 144–153, New York, NY, USA, 2006. ACM.

[26] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[27] Dana Zhang, Kotagiri Ramamohanarao, and Tim Ebringer. Role Engineering Using Graph Optimisation. In *SACMAT '07: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 139–144, New York, NY, USA, 2007. ACM.

A Sample Data

```
vp,vpino01,sendEmail,vpino01
vp,vpino01,readEmail,vpino01
vp,vpino01,adminLogin,desk-vpino01
vp,vpino01,adminLogin,lap-vpino01
vp,vpino01,modifyCalendar,vpino01
sp-domestic,sdoe003,sendEmail,sdoe003
sp-domestic,sdoe003,readEmail,sdoe003
sp-domestic,sdoe003,adminLogin,desk-sdoe003
sp-domestic,sdoe003,adminLogin,lap-sdoe003
sp-domestic,sdoe003,modifyCalendar,sdoe003
sp-mixed,smonroe005,sendEmail,smonroe005
sp-mixed,smonroe005,readEmail,smonroe005
sp-mixed,smonroe005,adminLogin,desk-smonroe005
sp-mixed,smonroe005,adminLogin,lap-smonroe005
sp-mixed,smonroe005,modifyCalendar,smonroe005
sp-foreign,sfolk007,sendEmail,sfolk007
sp-foreign,sfolk007,readEmail,sfolk007
sp-foreign,sfolk007,adminLogin,desk-sfolk007
sp-foreign,sfolk007,adminLogin,lap-sfolk007
sp-foreign,sfolk007,modifyCalendar,sfolk007
sp-foreign,sfolk007,sendEmail,sfolk007
sp-foreign,sfolk007,readEmail,sfolk007
sp-foreign,sfolk007,adminLogin,desk-sfolk007
sp-foreign,sfolk007,adminLogin,lap-sfolk007
sp-foreign,sfolk007,modifyCalendar,sfolk007
compleg,clego009,sendEmail,clego009
compleg,clego009,readEmail,clego009
compleg,clego009,adminLogin,desk-clego009
compleg,clego009,adminLogin,lap-clego009
compleg,clego009,modifyCalendar,clego009
oper,oopenhew011,sendEmail,oopenhew011
oper,oopenhew011,readEmail,oopenhew011
oper,oopenhew011,adminLogin,desk-oopenhew011
oper,oopenhew011,adminLogin,lap-oopenhew011
oper,oopenhew011,modifyCalendar,oopenhew011
asst,aardo02,modifyCalendar,aardo02
asst,aardo02,readEmail,aardo02
asst,aardo02,readEmail,vpino01
asst,aardo02,userLogin,desk-vpino01
asst,aardo02,modifyCalendar,vpino01
```

```
asst,aada004,modifyCalendar,aada004
asst,aada004,sendEmail,aada004
asst,aada004,readEmail,aada004
asst,aada004,readEmail,sdoe003
asst,aada004,userLogin,desk-sdoe003
asst,aada004,modifyCalendar,sdoe003
asst,aarnold006,modifyCalendar,aarnold006
asst,aarnold006,sendEmail,aarnold006
asst,aarnold006,readEmail,aarnold006
asst,aarnold006,readEmail,smonroe005
asst,aarnold006,userLogin,desk-smonroe005
asst,aarnold006,modifyCalendar,smonroe005
asst,aark008,modifyCalendar,aark008
asst,aark008,sendEmail,aark008
asst,aark008,readEmail,aark008
asst,aark008,readEmail,sfolk007
asst,aark008,userLogin,desk-sfolk007
asst,aark008,modifyCalendar,sfolk007
asst,aaquis010,modifyCalendar,aaquis010
asst,aaquis010,sendEmail,aaquis010
asst,aaquis010,readEmail,aaquis010
asst,aaquis010,readEmail,clego009
asst,aaquis010,userLogin,desk-clego009
asst,aaquis010,modifyCalendar,clego009
asst,aargent012,modifyCalendar,aargent012
asst,aargent012,sendEmail,aargent012
asst,aargent012,readEmail,aargent012
asst,aargent012,readEmail,oopenhew011
asst,aargent012,userLogin,desk-oopenhew011
asst,aargent012,modifyCalendar,oopenhew011
```

B Collected Statistics

```
-------------------------------------
Collected statistics on: 1212130742475
-------------------------------------

Analysed:
        Job positions: 5
        Users        : 12
        Operations   : 5
        Objects      : 48

Stats:
        Object taxonomy compression factor: 16.0
        Operation taxonomy compression factor: 1.25
        Object types : 3
        Operation Types : 4
        Permission Types : 4
        Permission type/taxonomy ratio: 0.3333333333333333
        Permission types per role: 0.8
        Users per role: 2.4

Operation taxonomy
-----------------------

send:
        sendEmail
modify:
        modifyCalendar
login:
        adminLogin
        userLogin
read:
        readEmail

Object taxonomy
-----------------------

cal-acct(2)
  asst(1)
    aada004(0)
    aaquis010(0)
    aardo02(0)
    aargent012(0)
    aark008(0)
    aarnold006(0)
    clego009(0)
    oopenhew011(0)
    sdoe003(0)
    sfolk007(0)
    smonroe005(0)
    vpino01(0)
  compleg(1)
    clego009(0)
  oper(1)
    oopenhew011(0)
  sp(1)
    sdoe003(0)
    sfolk007(0)
    smonroe005(0)
  vp(1)
    vpino01(0)

computer(2)
  desktop(1)
    desk-clego009(0)
    desk-oopenhew011(0)
    desk-sdoe003(0)
    desk-sfolk007(0)
    desk-smonroe005(0)
    desk-vpino01(0)
  laptop(1)
    lap-clego009(0)
    lap-oopenhew011(0)
    lap-sdoe003(0)
    lap-sfolk007(0)
    lap-smonroe005(0)
    lap-vpino01(0)

email-acct(2)
  asst(1)
    aada004(0)
    aaquis010(0)
    aardo02(0)
    aargent012(0)
    aark008(0)
    aarnold006(0)
    clego009(0)
    oopenhew011(0)
    sdoe003(0)
    sfolk007(0)
    smonroe005(0)
    vpino01(0)
  compleg(1)
    clego009(0)
  oper(1)
    oopenhew011(0)
  sp(1)
    sdoe003(0)
    sfolk007(0)
    smonroe005(0)
  vp(1)
    vpino01(0)


Permission types
-----------------------

[<send : email-acct>,
 <read : email-acct>,
 <login : computer>,
 <modify : cal-acct>]
```

C Role Structure Function Data

```
vp,send:email-acct,1
vp,read:email-acct,1
vp,login:computer,1
vp,login:computer,1
vp,modify:cal-acct,1
sp-domestic,send:email-acct,1
sp-domestic,read:email-acct,1
sp-domestic,login:computer,1
sp-domestic,login:computer,1
sp-domestic,modify:cal-acct,1
sp-mixed,send:email-acct,1
sp-mixed,read:email-acct,1
sp-mixed,login:computer,1
sp-mixed,login:computer,1
sp-mixed,modify:cal-acct,1
sp-foreign,send:email-acct,1
sp-foreign,read:email-acct,1
sp-foreign,login:computer,1
sp-foreign,login:computer,1
sp-foreign,modify:cal-acct,1
sp-foreign,send:email-acct,1
sp-foreign,read:email-acct,1
sp-foreign,login:computer,1
sp-foreign,login:computer,1
sp-foreign,modify:cal-acct,1
compleg,send:email-acct,1
compleg,read:email-acct,1
compleg,login:computer,1
compleg,login:computer,1
compleg,modify:cal-acct,1
oper,send:email-acct,1
oper,read:email-acct,1
oper,login:computer,1
oper,login:computer,1
oper,modify:cal-acct,1
asst,modify:cal-acct,1
asst,read:email-acct,1
asst,read:email-acct,1
asst,login:computer,1
asst,modify:cal-acct,1
```

D See5 Output

```
See5 [Release 2.05]     Fri May 30 10:26:43 2008
-------------------

Class specified by attribute ‘hasPT’

Read 36 cases (3 attributes) from rolestruct.data

Decision tree:
 1 (36/4)


Evaluation on training data (36 cases):

            Decision Tree
          ----------------
          Size      Errors

             1    4(11.1%)   <<


           (a)   (b)    <-classified as
          ----  ----
                   4    (a): class 0
                  32    (b): class 1


Evaluation on test data (34 cases):

            Decision Tree
          ----------------
          Size      Errors

             1    3( 8.8%)   <<


           (a)   (b)    <-classified as
          ----  ----
                   3    (a): class 0
                  31    (b): class 1
```