Abstract. We propose a new static approach to RBAC policy enforcement. The static approach we advocate includes a new design methodology, for applications involving RBAC, which integrates the security requirements into the system’s architecture, helping to ensure that policies are correctly defined and enforced. We apply this new approach to policies restricting calls to methods in Java applications. However, our approach is more general and can be applied to other Object-Oriented languages. We present a language to express RBAC policies on calls to methods in Java, a set of design patterns which Java programs must adhere to for the policy to be enforced statically, and a high-level algorithm for static enforcement.

1 Introduction

The objectives of an access control system are often described in terms of protecting system resources against inappropriate or undesired user access. When there is a request for a resource, the system must check who triggered the request (authentication), check if that user has the permission for the request to be fulfilled (authorisation) and as a result allow or deny the request (enforcement). Thus, an implementation of access control requires a specification of the rights associated to users in relation to resources (a policy). For this, several models of access control have been defined, from simple access control lists giving for each user the list of authorised operations, to more abstract models, such as the popular Role-Based Access Control (RBAC) model [14], where each user has one or more roles, and each role has an associated list of permissions on resources.

Our focus is on enforcement, for which there exist two main approaches, static and dynamic, with a recently emerged third approach combining the two: the hybrid approach. The static approach performs all access checks at compile-time, whereas the dynamic approach performs these at run-time. In short, the static approach enables policy violations to be detected earlier, facilitating debugging and reducing the impact on testing, and usually involves a lower run-time cost. However, the kinds of policies enforceable statically are not as expressive nor as flexible as those enforceable by the dynamic approach. We refer to [3] for a more detailed comparison; see also [20] for hybrid analysis of programs, although not directly applicable to our problem.

In order to explain our approach, we introduce a small informal running example of a GP/doctor’s surgery where the data is stored according to the Entity-Relationship Model in Figure 1. In this example, there are roles Admin, NHS Doctor and Private Doctor. The Admin should be allowed to read, write and delete NHS and private patient records, appointment records and staff records. The NHS Doctor should be able to read, write and delete NHS visit records (check-ups for NHS patients) and NHS prescriptions, and read NHS patient records. The Private Doctor should be allowed similar operations to the NHS Doctor but for Private patients instead of NHS patients.

This paper addresses the problem of enforcement of RBAC policies within Object-Oriented (OO) programs, showing examples in the Java programming language [2]. In OO languages, programs are collections of objects that are classified using the notion of a class. An object combines data (fields) and operations (methods). The overall goal of our work is to enforce general access control policies using a hybrid approach, that is, using a combination of compile-time and run-time checks. In this paper, we present the first stage of our work, which is focused solely on static enforcement. Our main result is a mechanism to fully verify RBAC policies statically. More precisely,
we consider implementations of RBAC policies in Java, where policies restrict method invocations, and present a static source-code verifier to enforce the policies. Our static verifier ensures that programs that are validated contain no unauthorised method invocations.

RBAC is a widely used policy specification model. Our static program analysis is applicable to RBAC implementations under certain important conditions. The first of these is that the source code must be available at compile-time. Secondly, the code should not be modified at run-time through mechanisms such as reflection, therefore our system is aimed at non-malicious, "honest", programmers. Thirdly, the policy should not rely on dynamic information (which changes throughout execution). The latter condition holds for the first and second "levels" of the standardised RBAC models: flat-RBAC and hierarchical-RBAC [19]. We therefore provide a policy specification language which supports resource, permission and role definitions, and also role hierarchies. To the best of our knowledge, these kinds of policies are typically enforced dynamically in today’s available RBAC systems (e.g., Java Web Security amongst others [17, 2]). One significant reason for this is that during static analysis, it is difficult to know which regions of code are accessible by users with which roles. This is because the roles are not usually part of the application at the design and source level - they exist only at run-time as part of the (dynamic) security context information. We have solved this problem through the use of a program design methodology, which integrates the RBAC model in the system’s architecture.

Motivation. To highlight the problem, consider the following Java code:

```java
if (securityContext.isUserActiveRole("admin"))
    wipeData();
```

These kinds of code snippets are common in RBAC implementations. In such cases, a programmer would want to be sure that only the authorised role ('admin', in this example) can invoke the security-critical, or protected method ('wipeData', in this example). This would usually be done
using a dynamic check - the \textit{if} statement (which in this case utilises Java Servlet API’s \texttt{isUserInRole()} method [21]), before any such method invocation. The program would then have to be rigorously tested to ensure that each role can reach only those invocations that it is allowed to. It would be reasonable to assume that the number of test cases needed would increase as the number of roles increases and the number of protected invocations in the program increases.

Catching errors at an early stage statically aids debugging, reduces testing time and can reduce impact on run-time resources when compared to catching errors only at run-time. So, since a hierarchical-RBAC (and also flat-RBAC) policy is static, a program implementing this policy should be able to be checked at compile-time for policy compliance.

Having said that, just because the policy is static does not mean it is trivial to statically check the program for policy compliance. Let us start by removing the dynamic check in our example code - the \textit{if} statement - leaving just the invocation of the protected method. We now need to know statically which roles can perform the invocation. The difficulty is that the active role exists only in the security context of the program’s execution. In this paper, we show that it is indeed possible to statically enforce static (i.e. flat or hierarchical) RBAC policies. We can omit the user-role assignments from our static information in order to allow these to change, providing more flexibility.

\textbf{Contributions.} We propose a static solution to RBAC policy enforcement for Java programs through the use of new \textit{RBAC MVC} design patterns, which integrate roles into the program as a set of Model-View-Controller (MVC) [16] components (i.e. classes) for each role. Each role’s associated MVC classes act as a role-specific interface to accessing resources - the invocation of protected methods in resources are made in these role classes only. The flow of the program directs users to the set of role classes associated to their active role. Finally, the protected invocations are checked statically for policy compliance. We present a static verifier, which performs syntactic checks and call graph analysis to ensure the invocations to methods in resource classes are made only in role classes, and role classes do not invoke methods of components belonging to other roles. To the best of our knowledge, this is the first static verifier available for RBAC policies.

\textit{Overview} Section 2 recalls the main notions in RBAC and the Java EE platform, as well as the concept of a design pattern. Section 3 gives an overview of the methodology we propose. Section 4 introduces the policy language, and Section 5 the set of design patterns RBAC MVC. Section 6 illustrates the patterns with some examples. Section 7 presents the static verification algorithm. Section 8 describes a case study. Section 9 evaluates the work, and Section 10 describes the related work. Finally, Section 11 contains the conclusions and discusses future work.

2 Preliminaries

In this section, we describe the basic notions on which our work is based: Role-Based Access Control, the Java EE platform and the concept of a design pattern. For more details we refer to [14, 15, 9].

2.1 Role-Based Access Control

Role-Based Access Control (RBAC) is a popular mechanism to protect resources from unauthorised use in an organisation, where each member has a role, or multiple roles, assigned to them [14]. In an RBAC policy, instead of specifying all the accesses each user is allowed to execute, access authorisations on objects are specified for roles. Each role is given access rights, and each user is given roles so that only authenticated users who have activated the required role can access and use the restricted resources. Roles can be arranged in a hierarchy, where a more senior role ‘subsumes’ another; the senior role inherits the permissions of the subsumed role and can be assigned further permissions.
Java EE is a platform for the creation of distributed applications using the Java language. It provides the means to create the server-side components and business logic of an application. These can then be used in two ways for the client side of the application:

- an Application Client, which is a Java program on the client-side which interacts with server-side components through Remote Method Invocation (RMI) where methods are remotely invoked over a network, or
- a Web Client, where an internet browser on the client-side interacts with the server-side components through dynamic HTML pages.

In either case, JEE Applications usually follow an architecture consisting of three tiers, as shown in Fig 2 (taken from the JEE 7 tutorial). Although four tiers appear, the tiers are distributed over three locations: client machines, the Java EE Server machine, and database or legacy machines. For this reason, the JEE application model is considered to be a three-tiered architecture. We describe the main components of the architecture below.

Three-Tiered Architecture.

- The **Client Tier** consists of either a Java Application Client, or dynamic HTML pages which are generated by the server and accessed via a web client (usually an internet browser) on the client machine. These components run on the client machine.
- The **Server Tier** consists of two types of components:
  - **Web Tier** components of which there are three types:
    - Servlets, which usually control the flow of a web application and are invoked from web pages or JavaServer Pages (see below) via the web client,
* JavaServer Pages (JSP) which are web pages interleaved with Java code, or
* JavaServer Faces (JSF), which builds on Servlets and JSPs by providing a user-
  interface component framework for web applications.

In this work, when discussing the Web Tier we focus on Servlets and JSPs, since they form
the basis of a JEE application and JSF is actually built on top of these.

- **Business Tier** components, called Enterprise Java Beans (EJB). EJBs are classes that
  contain the business logic of the application and are run on the JEE Server.
  - The **Database Server Tier** consists of data sources such as databases, or other components
    such as web services.

### 2.3 Patterns

A pattern describes a particular recurring design problem that arises in specific design context, and
presents a generic scheme for its solution [9]. Patterns are usually described using the semi-formal
Unified Modelling Language (UML) notation, showing its constituent components, their responsibilities
and relationships, and the way in which they collaborate. They are used by developers, who
are not necessarily familiar with formal methods. The goal of patterns is to provide a mechanism
to guide the implementation of a solution to a specific problem. Patterns originate from known
good development practices, and can be directly used to aid development. A widely used template
for describing patterns is the Pattern-Oriented Software Architecture (POSA) template [9]. From
it, we use the following sections to present the patterns:

- **Name**: Each design pattern has a unique name.
- **Example**: A concrete example of the problem’s existence highlighting the need for the pattern.
- **Context**: The context in which the pattern can be useful.
- **Problem**: A description of the problem that this pattern solves.
- **Solution**: A description of the classes and objects used in the pattern, how they interact with
each other and a graphical representation in UML.
- **Implementation**: A description of guidelines to help in the implementation of the pattern.
- **Consequences**: The results, side effects and trade-offs caused when using the pattern.
- **Known Uses**: Examples where the pattern is used.
- **Related Patterns**: A description of which patterns solve similar problems or are somehow
  related to this pattern.

**Model-View-Controller Pattern.** Our work utilises concepts from the Model-View-Controller
(MVC) pattern. This pattern achieves separation of concern for user interaction [16], separating
data processing from user interaction, allowing both to be modified independently. Data access,
data-processing logic and data presentation are divided into three distinct categories: Model,
Controller and View. The Model components contain the data or data-related logic, views present
the data, and the controller processes events affecting the model or views. Figure 3 illustrates the
relationships between the components of MVC.

### 3 Concept Overview

We propose a design methodology for OO programs, where RBAC is implemented at the 'Tasks'
level (see Figure 4). Our goal is to statically enforce an RBAC policy that restricts the methods
that users call on resources within a program. There are three main stages in this approach:

1. The first stage is specifying a policy following the RBAC model, where roles are assigned
   permissions to invoke methods on resource classes.
2. The second is designing programs such that they implement a specialisation of the general
   model of the flow of a program that implements RBAC (the general model is shown in the
   left-hand side of Figure 4). This is achieved through our design patterns, *RBAC MVC*, which
we describe in more detail in the following Subsection and in Section 5. Put simply, when users log-in, they are presented with an interface for the Session. This Session-specific interface then allows users to interact with the system via their role, performing their role-specific tasks in the system, or without their role thus performing "Other" tasks. Each role is implemented via a set of MVC components, which can be called by the Session interface when the user wishes to interact with the system using their currently active role.

3. The third stage is then running the static verifier, which checks the code in Role MVC classes to ensure that each method call to an access-restricted method respects the policy, and checks the whole program to ensure that calls to access-restricted methods are only made in these components.

Below we give a more detailed overview of the second stage above.

### 3.1 Overall Program Flow and Static Approach

We now describe the general flow of RBAC programs and how this general flow is specialised in our approach to enable static verification.

In general, programs that restrict access to resources from users typically involve an initial user authentication phase, where users log in and retrieve their access rights, then allowing users to undertake user-tasks which may involve accessing resources, and finally logging out of the system. We present a simplified model of the general flow of a program which implements RBAC in the left-hand side of Figure 4. In RBAC, authentication also involves retrieving and activating the role(s) associated to the user, and logging out also involves deactivating the role(s). Controlling access most commonly takes place between 'Tasks' and 'Resources', for example through a reference monitor intercepting all access requests made to resources at run-time, stopping those requests which are invalid according to the policy. The left hand side of Figure 4 shows that each and every access request to resources must be intercepted (at run-time) by the reference monitor, to check if it is allowed. Only the allowed requests can then continue to access resources.

In our approach, we divide user-tasks into three groups: 'Role tasks' that users with certain roles in the system can perform (these may access resources), 'Other tasks' that the policy is not directly concerned with and 'Session tasks' related to the functioning of the Session.

When at the 'Tasks' stage in our approach, there is a 'Session Interface' - which can be thought of as a "master" interface specific for the session. This contains components implementing Session Tasks, which will always be active, e.g. components for logging out which need to be accessible by the user at all times. Components for performing Role tasks and Other tasks are composed into the Session Interface. Role tasks are performed through 'Role Interface', which prevents direct access to the resources, as shown in the right-hand side of Figure 4. These are realised through a set of MVC components for each role: a set of View components, one Controller component
and one Model component. A resource is realised as a Model component. In order to perform a
task which may include accessing a resource, the user interacts with a Role View which will then
communicate with its Role Controller, which will then communicate with its Role Model. Only the
latter, Role Model, implements the business functions of the role - including tasks which may access
resources. The Role Views deal with presentational logic for user-interaction and Role Controllers
handle communication between Role Views and Role Models (and vice-versa). The implementation
of 'Other tasks' is unrestricted in our approach, but will be verified to ensure the policy is not
violated. The Session interface is also made up of MVC components, which handle three aspects:
implementing Session Tasks e.g. log-in and log-out, linking to Role interfaces (to compose them
with the Session Interface) and linking to the classes that implement 'Other Tasks'. In this way,
roles are integrated into the program design, which ensures that it is possible to determine, at
compile-time, which role is making a particular invocation. When verifying an invocation in a role
component, the role which this component belongs to is the role that can reach this invocation.
Our 'RBAC MVC' patterns guide the implementation of the program to achieve this flow.

4 Policy: Context and Language Syntax

As we have already mentioned, the goal of our approach is to statically verify a Java program that
contains some classes containing methods whose invocation needs to be restricted. This restriction
is specified using hierarchical-RBAC. Thus, these classes become resources (in RBAC) and the
restricted methods are the operations (in RBAC) that can be performed on them. In this section
we first define the core concepts of the policy in the context of our approach, and then present a
language that will be used to specify RBAC policies restricting access in Java programs.

4.1 Policy Concepts in our Approach

Definition 1 (Resource). A resource is realised as a Resource class containing some methods
whose invocation needs to be restricted. Invocations are restricted for instances of Resource classes.

Methods in resource classes are categorised as follows.

Definition 2 (Actions and Auxiliary Methods). An action is a method in a resource class
that must only be invoked by those users with the permission to do so.
An auxiliary method is a method in a resource class that is not part of the policy definition. Such methods are usually required for the correct initialisation and operation of a class, and should not be invoked directly by users.

Definition 3 (Permission). A permission is a pair \([\text{res, act}]\) where \(\text{res}\) is the name of a resource and \(\text{act}\) is the name of an action of that resource. The action is allowed to be invoked on any instance of that resource class by the role (see Definition 6) which the permission is assigned to.

Definition 4 (Access). An access to a resource is an invocation or call to an action method of an instance of a Resource class.

Definition 5 (Task). We divide the concept of a user-task into three groups as follows. Firstly, a Role Task is an operation, or business function, to be performed by an authorised user in a specific role, which could involve the invocation of one or more actions on resources. Secondly, a Session Task is an operation required to correctly manage the session e.g. log-in and log-out. Thirdly, an Other Task is an operation or function that is executable by all users, regardless of the notion of role as it does not access resources (in the access control sense).

An example of a Role task is as follows. A user in an Admin role in a GP surgery may need to perform the task \texttt{registerPatient}, which would involve a call to an action e.g. \texttt{addPatient} in the \texttt{Patients} resource.

Definition 6 (Role). At the policy level, a role consists of a name and a list of permissions to access resources.

In the context of our system, a role is implemented by a set of MVC components: a set of Role View components (i.e. classes), a Role Controller class and a Role Model class (as defined below). Together, these provide a Role-specific Interface for the user to perform tasks. We define these components below.

Definition 7 (Role Model). A Role Model provides Role Task methods which should only call those actions that are permitted for its role. Its name must be prefixed with the name of the role, followed by 'Model'.

Definition 8 (Role Controller). A Role Controller acts as an intermediary between the Role Model and View classes. Its name must be prefixed with the name of the role, followed by 'Controller'. Role Controller methods are invoked by Role View classes to communicate with the Controller.

Definition 9 (Role View). A Role View provides (part of) the user-interface for users to execute the tasks of their role. Its name must be prefixed with the name of the role, followed by 'View' (followed by any valid Java identifier).

For any role \(r\), its single associated Role Model class contains the code that performs the tasks \(r\) can do in the system. The role’s multiple associated Role View classes and its single associated Role Controller class, provide the means for users that have activated this role to access these tasks (and perform them).

An example of a set of role components is as follows. Firstly, Role Model class \texttt{AdminModel}, which provides a Role Task \texttt{registerPatient()} that calls on the \texttt{addPatient()} action in the \texttt{Patients} resource. Secondly, a set of role view components \texttt{AdminViewPatients}, \texttt{AdminViewAppointments}, e.t.c. Thirdly, a role controller \texttt{AdminController} acting as an intermediary between the role view and model components.

Definition 10 (Session). A Session is the state of the program in which an authenticated user is able to perform the three kinds of tasks in the system. The Session has a user interface composed of a Session-specific interface, the role-specific interface (made up of Role MVC components discussed above) of the current active role and any interfaces implementing Other tasks. The
Session-specific interface is made up of a set of MVC components: one Session Model, one Session Controller and a set of Session View classes. The Session Model implements the Session Tasks which are: log-in/authentication, role activation, log-out, calling a role-interface and calling classes that implement 'Other Tasks'. The Session Views and Controller provide the means for the user to access these Session tasks. Names of Session classes start with the string 'Session' followed by either 'Model', 'Controller' or 'View'. For the latter, since there can be many Session View classes, any valid Class identifier (in Java) is allowed to follow in the name.

The Session-specific interface should always be active so that the Session tasks are always available to the user. We, of course, have minimum expectations such as log-out is only available if logged-in and so forth. The Session-specific interface also allows the user to interact with the system via their role by calling a role-interface, or without their role thus calling Other task implementing classes.

The classes required for the session - Session Classes - constitute part of our Trusted Computing Base (TCB); the other part is the actions (within Resource classes), which we trust behave safely. The Session classes should contain the minimum code necessary to implement Session tasks, so that the TCB is small. We perform few checks, and exercise few constraints, on Session classes and actions, in order for their implementation to be as flexible as possible. Therefore, we do not deal with authentication in this paper. However, an important aspect of an RBAC system is activating a role, which is to be implemented by the Session classes. We give guidelines for role activation in our approach below.

**Definition 11 (Role Activation).** Role activation constitutes storing the chosen role in a field called activeRole in the SessionModel class and invoking the Role Controller of that role. This process is achieved in a method 'activateRole()' in the SessionModel class (See Definition 10). This will result in presenting a Role View of the active role’s Role-specific Interface to the user by composing it with the Session Interface.

### 4.2 Policy Language: JPol

We define a policy specification language for hierarchical-RBAC where resources, together with their associated lists of actions, and roles, together with their associated permissions, can be declared. To simplify, we assume that only the access requests that are allowed are expressed, so all other requests are not allowed. The policy file will be parsed and represented as a set of tables, to be used only at compile-time by the static algorithm in order to perform the access checks.

The policy language does not support user definitions and user-role assignments, since we do not deal with authentication in this paper. Since with our static algorithm, only the roles which have been declared in the policy will be permitted to be assigned to users, the resources will still be protected because each role will have been checked at compile-time to ensure it does not perform any illegal access requests. The proposed approach is flexible: new users can be added to the system and role assignments can change depending on changes within the organisation.

**Syntax and Representation** The policy language adopts an object-oriented, Java-like syntax designed to make the policy implementer’s transition from target program language, Java, to policy language as effortless as possible. However, as we will see later, the static algorithm relies solely on the information generated as a result of parsing the policy file. Thus, the syntax of the policy language can change and be adapted to any environment using hierarchical- (or flat-) RBAC.

The grammar of the policy language is as follows, where the keyword *subsume* indicates role inheritance. The abstract syntax of the policy language is illustrated in Figure 5.

```plaintext
stmts ::= (stmt ';'*)+
stmt ::= decRole | decRoleSubsume | decRes |
       | addActRes | addPermRole

decRole ::= 'Role' ID '=' 'new' 'Role' name

decRoleSubsume ::= 'Role' ID '=' 'new' 'Role' name
```
The Parser for the policy specification language checks that a policy declaration is syntactically correct, producing the Abstract Syntax Tree (AST) shown in Figure 5. It then generates intermediate data structures - tables called 'Resources' and 'Roles' containing the information needed for the static verification algorithm.

Listing 1.1 shows an example specification in JPol for patient-related resources and permissions for roles in an example GP/doctor’s surgery, with the resulting tables 'Resources' and 'Roles' shown in Figure 6.

**Listing 1.1.** Example JPol code declaring Resources with their actions and Roles with their permissions

```java
Resource nhspatient = new Resource('Nhspatient');
nhspatient.addAction('getFirstName');
Resource privatepatient = new Resource('Privatepatient');
privatepatient.addAction('getFirstName');
Role nhsdoctor = new Role('NHSDoctor');
nhsdoctor.addPermission('Nhspatient', 'getFirstName');
Role privatedoctor = new Role('PrivateDoctor');
privatedoctor.addPermission('Privatepatient', 'getFirstName');
Role admin = new Role('Admin');
admin.addPermission('Nhspatient', 'getFirstName');
admin.addPermission('Privatepatient', 'getFirstName');
```

**Fig. 5.** Abstract Syntax of Policy

**Fig. 6.** Example Roles and Resources Tables Representation
Semantics. We can state the semantics of the policy language in a concise manner by mapping the abstract syntax to elements of the RBAC model: there is a one-to-one correspondence between the resources, roles and permissions specified in JPol and in the RBAC model. In particular, an “addPermission” statement in JPol syntax (see the grammar rule for addPermRole above) corresponds directly to a permission in the RBAC sense. Therefore, we can define policy satisfaction as follows.

**Definition 12 (Policy Satisfaction).** A Java program satisfies a JPol policy if, for any invocation \( \text{res.m} \) that exists in the program, where \( \text{res} \) is an instance of a resource class \( \text{Res} \) and \( m \) an action, only authenticated users with active role \( r \), such that the JPol policy specifies the permission \([\text{Res}, m]\) for \( r \), can perform \( \text{res.m} \).

5 Program Design Patterns - RBAC MVC

In order for the target program to be statically checked for policy compliance, it must follow our RBAC MVC Patterns described below. The goal of these patterns is to specify a scheme for implementing access to resources utilising RBAC concepts. For static checking, it must be known statically where in the code action invocations are allowed to be made and then which actions are invoked. This is achieved by providing a model component for each resource and a model, a controller and a set of view components for each role. The role model component provides the Tasks that users of that role can perform, and so invocations of actions are only allowed in these components. Each role’s MVC components, together with the Session MVC pattern, provide a scheme for the system to operate such that users perform all their Tasks, which may involve accessing resources, through their roles which are implemented as sets of role MVC components.

The program can then be statically checked to ensure that action invocations only occur in role model components, and that the actions invoked are valid according to the policy for the role of each role model component.

5.1 RBAC Model Patterns

This pattern describes the design of Resource and Role model components. In this case, permissions on the resource are defined at the class level. This means that a role is assigned the permission to invoke an action on any instance of the resource. Its aim is to represent a resource and a role as classes such that the role classes present a coarse-grained set of methods which correspond to Tasks, each of which may invoke one or more actions. Action invocations should only be made in these Task methods.

![Fig. 7. UML Class Diagram of RBAC Model Pattern](image-url)
Name: RBAC Model

Example: A system in which some classes represent resources, e.g. a class each for tables in a GP Surgery database such as Patients, Appointments, Staff. Accessing a resource is achieved by invoking an action method of a resource class, such as a method `findAll()` in each of the classes representing these tables. Access to each table needs to be restricted using the RBAC model, so different roles have a different set of permissions associated with them. Roles such as 'Admin' and 'Staff' would exist. Each role has a set of Role Tasks that it can do e.g. 'Admin' can do a Role Task 'Register Patient'.

Context: An application in which access to resources needs to be restricted, following the hierarchical-RBAC model, with administrative changes disabled, and where access checking is to be performed at compile-time. In this case, the policy specifies restrictions on method invocations by roles. A resource is a class (e.g. PatientsTable) which contains some methods that can only be invoked by roles that have been assigned the permission to do so (e.g. `read()`, `write()`, `delete()`). We call such methods action methods. Any other methods in resources are called auxiliary methods. So roles are associated with a set of permissions; the right to invoke a particular action of a resource. Assigning a role the permission to invoke an action on this kind of resource means that the invocation can be on any of its instances. At compile-time, the static verifier will decide if all invocations of actions in the program are valid according to the policy.

Problem: The policy specifies permissions for roles to invoke actions on resources, however a user’s interaction with the system will usually consist of performing a set of tasks in the system that are dependent on their role(s). The business logic part of the program needs to implement these tasks. Also, static verifier must ensure that the action methods can only be invoked by users with the required role and that the action invocations are valid according to the policy. In the business logic part of the program, when an action is invoked in the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.

Structure: The structure for the solution is shown in the UML Class diagram in Fig 7. It contains:

- **Resource1**: a Resource Model component, which is a class representing a resource. One class for each kind of resource in the policy (referred to as `Resource1, Resource2, ..., ResourceN`) is needed. Named the same as the represented resource (without suffix 'Model').

- **Role1Model**: a Role Model component, which is a class representing a role. One class for each role in the policy (referred to as `Role1, Role2, ..., RoleN`) is needed. Named the same as the represented role with the suffix 'Model'.

Note that there can be multiple packages containing Resource and/or Role Model classes, as long as their names contain the string 'model'.

Dynamics: The business logic part of the application contains Resource and Role Model classes (any classes not related to the policy can also be included). Resource classes contain a set of action and auxiliary methods. Role Model classes contain a set of task methods which contain the business logic for carrying out tasks based on the role and can contain one or more action method invocations. In the business logic part of the application, invocations of actions should only be made in Role Model classes and Resource classes.

Implementation: All Role Model and Resource classes should be implemented in packages with the string 'model' appearing in the package name. In Java, the package name would contain 'model' as the name of one hierarchical part of the package name such as 'model.tables' for the tables listed in 'Example' section. Resource Model classes must be named the same as the resource they represent. Using the same example, there would be classes `PatientsTable`, `AppointmentsTable` and `StaffTable`. Action methods should have the modifier 'public' and auxiliary methods should have the 'private' modifier, so that they are not accessed by other classes. If each of the classes had an action `findAll()`, it would have the modifier `public`. If each had an auxiliary method `format()`, it would have the `private` modifier. Role Model components should be implemented as classes with the same name as the role followed by the suffix 'Model' e.g. for a role 'Admin', the Role Model class name will be `AdminModel`. The Task
methods should have the 'public' modifier. In JEE, the model components can be implemented on the server-side, as EJBS.

- **Consequences**: The business logic part of the program is defined and separated from the rest of the program via the use of the keyword 'model' in the package name. It is also separated to reflect the Roles that have been defined in the policy, using Role Model classes. These act as the interface to performing tasks in the system, which includes accessing the Resource classes, so other parts of the program should not access resources directly. Where an action is invoked in the business logic part of the program, we know statically what role will be making the invocation - the role that is represented by the Role Model class in which the action is invoked. The static verifier can then check if that role has the permission to call that action according to the policy. Note that actions can be invoked in Resource classes also, which is left unrestricted (in terms of both design and static analysis) in order to allow flexibility in implementing resources that depend on one another.

- **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC (i.e. permissions require no dynamic/run-time data). In this paper, we show an implementation in JEE (see Section 8).

- **Related Patterns**: RBAC Controller, RBAC View, RBAC Session.

### 5.2 RBAC Controller Patterns

The following describes the design pattern for the controller component required for each role.

![UML Class Diagram of RBAC Controller Pattern](image)

**Fig. 8.** UML Class Diagram of RBAC Controller Pattern

- **Name**: RBAC Controller
- **Example**: The example is carried over from the previously discussed RBAC Model pattern.
- **Context**: The context is also carried over from the RBAC Model pattern, however the application is assumed to follow the RBAC Model Pattern described above. So, each role has a corresponding model component, a Role Model class, which provides the Task methods which may invoke actions on resources.

- **Problem**: Each role has an associated model component, a Role Model class, which implements the business logic for the tasks which that role can perform in the system. Consequently, each role will also need a controller component to mediate communication between view components and the Role Model classes. The controller has two responsibilities. This first is to process input from the user from view components to decide which task(s) in which Role Model class(es) to invoke. The second is to process output from the Role Model classes to map the output to certain view components. Similar to the RBAC Model pattern, when an action is invoked in the controller parts of the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.

- **Structure**: The structure is shown in Figure 8. As well as the model classes from the RBAC Model pattern, it contains the following elements:
• **Package controller.roles**: A package containing a controller for each role in the policy.

• **Role1Controller, Role2Controller**: These are **Role Controllers**. These are controller components corresponding to each role in the policy, named the same as the matching role suffixed by the string 'Controller'.

• **RoleController**: An empty interface, with this exact name, that each Role Controller must implement.

Note that there can be multiple packages containing Role Controllers, as long as their names contain the string 'controller'.

- **Dynamics**: A Role Controller class provides the controller component that acts as intermediary between View components and the Role Model class of the same Role. It handles input from the views to select one or more Role tasks to invoke, and maps the output of a Role task to a view. Each Role Controller is associated with only one Role Model, the one with the same role name. No other class can invoke a method in a Role Model class. Additionally, since Role Controllers indirectly invoke actions through invoking Tasks, no classes except role-specific View components (discussed in the next pattern) should invoke Role Controllers. Role controller can also call actions, therefore any action invocation must be checked by the static verifier. The empty interface groups Role Controllers so that they can be linked to as a group.

- **Implementation**: In JEE, there are two separate implementation cases. In the case of a Web Client, Role Controllers can be realised as Servlets, which contain the HTTP GET, and HTTP POST methods, implemented as `doGet()`, and `doPost()`. The latter two methods must be ‘protected’, as per the API specification. In the case of an Application Client, Role Controllers can be realised as normal Java classes acting as controller components.

- **Consequences**: Role Model components will be interacted with using role-specific controller components. In order to perform Tasks and access resources, the view components presented to the user must interact with these Role Controllers. Action invocations in controller classes can be checked statically since the role that would invoke that action will be the role that is represented by that Role Controller. Methods in Role Controllers have the 'public', 'protected' (as well as 'private') modifier, meaning they are accessible in other classes therefore their usage needs to be verified at compile-time by the Static Verifier.

- **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and/or 'protected', where the access control policy uses static RBAC. In this paper, we show an implementation in JEE.

- **Related Patterns**: RBAC Model, RBAC View, RBAC Session.

### 5.3 RBAC View Pattern

Now that we have defined role-specific model and controller components, we define role-specific view components.

- **Name**: RBAC View

- **Example**: The example is carried over from the RBAC Model pattern.

- **Context**: The context is also carried over from the RBAC Model pattern, however the application is assumed to follow the RBAC Model and RBAC Controller patterns discussed above. So, each role has a corresponding model component (i.e., a Role Model class), and controller component (i.e., a Role Controller class).

- **Problem**: Each role has an associated controller and model component. The UI presented to the user should allow them to perform only the tasks that are specific to their role(s). The view components used to build and display the UI must interact with the Role Controller(s) associated with the same Role(s). Similar to the RBAC Model pattern, when an action is invoked in the presentation part of the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.
Fig. 9. UML Class Diagram of RBAC View Pattern

– **Structure**: The structure is shown in Figure 9. As well as the controller classes from the RBAC Controller pattern, it contains the following elements:
  - Package *view*: a package for the view components for each role. The name of these packages must have the prefix 'view.'.
  - Role1ViewN, Role2ViewN: The view components that are specific to each role. Names are restricted such that they begin with the name of the role, followed by the string 'View', followed by any valid class identifier.

Note that each role should have a corresponding package where the last hierarchical levels in the package name are the string 'view.' followed by the name of the role e.g. 'view.admin'.

– **Dynamics**: Each role has an associated set of view components. These views present the UI elements needed for each specific role’s tasks. Each Role View interacts only with the same role’s Role Controller. The Role Views can invoke actions, for example to print a 'PatientsTable' object’s data to the screen by calling its 'read()' method.

– **Implementation**: In JEE applications, there are two separate cases. In the case of a web client, the Role View elements could be implemented as JSP pages. Invocations to the same role’s controller component could be achieved via a form or button submit action (or similar). This would invoke the Role Controller Servlet’s HTTP GET or POST method, which could in turn invoke any other method in the Role Controller or a Role Task in the Role Model class belonging to the same Role. In an Application Client, they would be normal Java classes implemented as view components. To interact with the same role’s controller component, Role Views will contain an invocation to the same role’s Role Controller methods. In either case, the package names would have one hierarchical level as the word ‘view’ and the next hierarchical level as the name of the role, for example the view components for a role 'Admin' would be in the package 'view.admin'.

– **Consequences**: The presentation part of the application is separated from the rest of the program, and also segregated into groups reflecting the roles specified in the policy. Each role has a specific set of UIs. The user can only perform those operations in the system that are available in the Role Views for the role(s) currently active for their session. These can then be statically checked to ensure they only invoke the same role’s controller component. Action invocations in Role Views can be statically checked since it can be known which role would invoke that action - the role that is associated with that Role View.

– **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC. In this paper, we show an implementation in JEE.
5.4 RBAC Session Patterns

The previous patterns have defined a scheme for the program so that the user interacts with the application through their role-specific MVC components. Now, we define a pattern that guides the implementation of the Session in our approach, which is a Session-specific interface implementing the Session Tasks made up of MVC components - one Session Model, one Session Controller and a set of Session View components. See Definition 10 for details of the Session Tasks and the behaviour of the Session interface.

- **Name:** RBAC Session
- **Example:** The example is carried over from the RBAC Model pattern. Specifically for this pattern, users have multiple roles assigned to them. For example, a user could have the role 'Admin' and 'Employee' assigned to them.
- **Context:** The program follows the RBAC Model, RBAC View and RBAC Controller patterns described above. Each Role has a specific set of MVC components which are used by the user in order to interact with the application.
- **Problem:** Each role has a specific set of MVC components associated with it. This means that each role becomes an interface to the application. The flow of the program should reach a stage in which the user can perform the three types of Tasks in the system: Role Tasks corresponding to their active Role, Session Tasks or Other Tasks.
- **Structure:** The structure for this pattern is shown in Figure 10. As well as the Role Controller classes from the RBAC Controller pattern, it contains:
  - **SessionModel:** A model component representing the notion of a session. It contains methods that implement the Session Tasks. Specifically, it links to the Role Controller (and also Role Views) of the currently active role of the user, to call that role’s role-interface and display it to the user.
- **SessionView**: One or more view components representing the Session, which displays a user-interface allowing the user to access and perform Session Tasks.
- **SessionController**: A controller component representing the Session. The user-interaction from the SessionView is passed to the SessionController, which processes it and decides what to do with it e.g. invoke a method in SessionModel. It then retrieves the result from the SessionModel and displays it using a SessionView.
- **OtherClass**: One or more classes implementing Other Tasks.

- **Dynamics**: The user’s interaction with the system begins through the Session components. These implement Session Tasks, which the user initiates through the Session Views, which communicate with the Session Controller, which processes the input from the Views and decides which Session Task to invoke from the Session Model class (if any). These Session MVC classes constitute a Session Interface. This interface is active at all times, to enable the user to invoke Session Tasks at any time. The user is initially presented with the Session View(s) that show the log-in/authentication user-interface components. If successful, the roles assigned to the user are retrieved and stored in the Session Model class. Then, the user can choose to interact with the system via their role, by activating one of the retrieved roles. This invokes the Role Controller of that Role, which then calls one or more of that role’s Role Views which are composed with the Session Interface. The user can also interact with the system without their role, thus invoking an Other class, which contains code implementing Other Tasks.

- **Implementation**: In JEE, the pattern can be implemented in two ways depending on the two cases. The Session Model would be similar in both cases i.e. a Session Bean implementing Session Tasks as its methods. When using an Application Client, the Session Views can be implemented as a Window e.g. JFrame and menu components e.g. JMenuBar, e.t.c which enable Session Tasks to be invoked. Role Views can be composed into the main display area of the window e.g. as JTabbedPane. After authentication, the list of roles assigned to the user can be stored in a data member retrievedRoles in the Session Model. If the user selects the Session View to interact with the system via their role, the Role activation task in the Session Model would be invoked through the Session View passing the selected operation to the Session Controller, which would invoke the Role Controller of that Role. Logging out is left open to the programmer but we assume it clears Session View panel. When using a Web Client, the HttpSession provides the notion of Session and thus no special class is necessary. The Session Views could be implemented as HTML menu items. The retrieved roles can be stored in the HttpSession object as a key-value pair where the key is retrievedRoles and the value is a list e.g. ArrayList of role names. This is written to the HttpSession object at the end of the authentication process. The Session View can be implemented as a fragment of a JSP/HTML page such as a sidebar menu which is included in all Role Views. The Session Controller can be implemented as a Servlet.

- **Consequences**: The user can switch between multiple retrieved roles where the view associated to each role presents a set of UIs unique to each role. Also, the user can initiate Session Tasks via Session MVC components or invoke classes that implement Other tasks.

- **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC. In this paper, we show an implementation in JEE.

- **Related Patterns**: RBAC Model, RBAC Controller, RBAC View.

6 Example Of Patterns

The patterns can be applied to many OO languages and implementation platforms, where there is the concept of visibility modifiers and packages. We show an example of implementing the patterns for the GP surgery running example.
6.1 RBAC Model

Figure 11 illustrates the RBAC Model pattern being used for designing the patient-related operations in the system. There are Resource classes for each of the 'NHSPatients' and 'PrivatePatients', which where each instance will represent one row in the corresponding table. The table itself is representing via the 'Facade' classes, which deal with querying the table to retrieve or update data e.g. retrieving all rows in the table as a list of instances of the corresponding class that represents a row. There are also three Role Model classes for each of the roles 'NHS Doctor' 'Private Doctor' and 'Admin'. The structure reflects the policy, which can be easily gleaned from the diagram - note that both the row classes and table classes/facades are Resources.

![Diagram showing RBAC Model](image-url)
6.2 RBAC Controller

Figure 12 illustrates the RBAC Controller pattern being used for designing the patient-related operations in the system. There are Role Controller classes for each of the roles discussed in the RBAC Model pattern example. The methods inside the controllers are responsible for different types of input from the views.

6.3 RBAC View

Figure 13 illustrates the RBAC View pattern being used for designing the patient-related operations in the system. The view ending with 'ops' is for selecting operations on the type of patient specified in the name, and the view ending with 'details' is used to display that type of patient’s data.

7 Static Verification

Our source-level static verifier takes as input a well-formed program, which is defined as follows:
Definition 13 (Well-formed program). A well-formed program consists of a (syntactically correct) JPol policy file and a Java program that implements the RBAC MVC patterns. Implementing the patterns means, in particular, that the Session classes: correctly authenticate users, activate the correct role(s) allowed for the user, switch roles correctly for the retrieved and selected roles.

A well-formed program might contain unauthorised calls to actions on resources. The static verifier should reject a program if an access violation is found, else accept it. In other words, it should only accept programs that satisfy the policy (see Definition 12). In this section, we describe high-level details of the algorithm for static checking. The first of two phases, described in Section 7.1, generates abstract syntax representations of the policy and program, and populates tables to be used in the second phase. The second phase, described in Section 7.2, uses the abstract syntax tree and tables to check that the program respects the policy.

7.1 Parsing

The policy and program are parsed to produce their ASTs and generate relevant intermediate data structures needed for the static verifier. We have already described in Section 4.2 the AST and tables Resources and Roles generated by parsing the policy. In the rest of this section we describe the process of parsing the program.

Categorising Classes and Generating Data. This first phase identifies the classes and categorises them as Resource classes, Role Model classes, Role Controller classes, Role View classes, Session Classes and Other classes (the latter contains any classes that do not fit into the other categories). This is illustrated in the top-level AST of the program in Figure 14.
The AST shown in Figure 14 is extended so that each class is represented by a tree under the node representing its category. The AST for all classes is the same regardless of category; parsing a class follows the standard parsing rules for Java classes. We show a simplified AST of a class in Figure 15.

![Fig. 14. Top-Level Abstract Syntax of Program](image)

![Fig. 15. Abstract Syntax of a Class in the Program](image)

In order to categorise each class, we use naming restrictions on the package and class names. The restrictions are described at a high level as follows. Note that 'package-id' is used as shorthand to represent any valid package identifier in Java and 'class-id' is used as shorthand to represent any valid class identifier in Java and that 'Roles' and 'Resources' refer to the tables generated from the policy discussed in Section 4.2. Also note that 'Class' refers to the Class node in the AST currently being traversed. We use the notation $s_1 \ll s_2$ to specify that $s_1$ is a substring of
\[ s_2, s_1 + s_2 \] to specify that \( s_1 \) is concatenated with \( s_2 \), \( tbl[n] \) to specify the \( n \)-th element of table \( tbl \) (where \( n \) is an integer).

- If the name of the package contains the substring 'model' (i.e., 'model' \( \ll \) package-id) then the package contains Model classes (Resource classes and/or Role Model classes).
- Within a package containing Model elements, Resource and Role Model classes are identified as follows:
  - If the name of the class is the same as a resource, i.e.,
    \[ \exists n. 0 < n < Resources.size \text{ and } Resources[n].name = Class.name \]
    then it is a Resource Class.
  - If the name of the class is the name of a role followed by the string 'Model', i.e.,
    \[ \exists n. 0 < n < Roles.size \text{ and } Roles[n].name + ' Model' = Class.name \]
    then the class is a Role Model class.
- If the name of a package contains the substring 'controller' (i.e., 'controller' \( \ll \) package-id) then the package contains Role controller classes.
- Within a package containing Role Controller classes:
  - If the name of the class is the name of a role followed by the string 'Controller', i.e.,
    \[ \exists n. 0 < n < Roles.size \text{ and } Roles[n].name + ' Controller' = Class.name \]
    then the class is a Role Controller class.
- If the name of a package contains the substring 'view', (i.e., 'view' \( \ll \) package-id) then the package contains Role View classes.
- Within a package containing Role View classes:
  - if the name of the class is the name of a role followed by the string 'View', followed by any valid Java class identifier, i.e.,
    \[ \exists n. 0 < n < Roles.size \text{ and } Roles[n].name + ' View' + 'class-id' = Class.name \]
    then the class is Role View class.
- If the name of a package contains the substring 'session' (i.e., 'session' \( \ll \) package-id) then the package contains Session classes.
- Within a package containing Session classes:
  - if the name of the class begins with the string 'Session', i.e.,
    \[ 'Session' + 'class-id' = Class.name \]
    then the class is a Session Class

After identifying the category of all the classes, this phase also generates tables containing the names of all classes in each category except the category of 'Other classes'. We call these tables 'ResourceClasses', 'RoleModelClasses', 'RoleControllerClasses', 'RoleViewClasses' and 'SessionClasses'. This is to simplify the process of looking up called classes in the checks made by the verifier (discussed below).

7.2 Static Verifier: Implementing Checks

The second phase performs checks of two kinds: checking the modifier of a declared method and checking the class that a method is invoked on. Checking the modifier of a method requires extracting the value of 'modifier' in each element of the list '(methods)' from the abstract syntax tree (AST) for the class (see Figure 15 for the abstract representation of a class). The modifier's correct value will depend on the details of the check being done. Checking the class that a method is invoked on requires extracting the value of 'called_method' when checking each element of the list '(calls)' when checking each element of the list '(methods)' in a class. The call's validity will be determined based on the kind of check being done. Both kinds of checks are discussed below. They are performed by traversing the AST of the program starting from the root. For each category of class (Resource, Role Model, Role Controller, or Other class), the algorithm performs specific checks as follows.
Resource Class Checks. The checks on Resource classes, performed by a subprogram of the verifier, called ResourceClassChecks, are described below.

1. For each method (i.e., each element of the list ’(methods)’, see Figure 15), we search the actions sub-table (generated when parsing the policy, see Figures 5 and 6) for ’Class.name’ (which is the name of a resource) then:
   (a) If the method name is in this sub-table, then the value of the node ‘modifier’ must be ‘public’.
   (b) Else the value of the node ‘modifier’ must be ‘private’.
2. For each call (each element of the list ’(calls)’, see Figure 15), we check that:
   (a) The called class (the node ’called_class’) is not the name of a Role Model class. This is done by searching the names of classes in the table ’RoleModelClasses’.
   (b) The called class is not the name of a Role Controller class. This is done by searching the names of classes in the table ’RoleControllerClasses’.
   (c) The called class is not the name of a Role View class. This is done by searching the names of classes in the table ’RoleViewClasses’.
   (d) The called class is not the name of a Session class.

Summarising: Resource classes cannot contain invocations to methods in Role Model, Role View, Role Controller, or Session classes.

Role Model Class Checks. The checks on Role Model classes, performed by a subprogram called RoleModelClassChecks, are described below.

1. First obtain the name of the associated Role for this class by removing the substring ’Model’ from ’Class.name’.
2. For each call, we check that:
   (a) If the called class is a Resource class, then
      i. If the called method (the node called_method) is an action, which is done by searching the ’actions’ sub-table for that resource in the table ’Resources’ generated when parsing the policy, see Figures 5 and 6, then the pair of values [’called_class’, ’called_method’] must appear in the permissions for the associated Role of the class (done by searching the ’permissions’ sub-table of the matching role in table ’Roles’).
   (b) The called class is not the name of a Role Controller class.
   (c) The called class is not the name of a Role View class.
   (d) The called class is not the name of a different Role Model class. This is done by checking if the called class contains the substring ’Model’, the name of the class must be the same as the value in the node ’Class.name’.
   (e) The called class is not the name of a Session class.

Summarising: Role Model classes cannot contain invocations to methods in Role View, Role Controller or Session classes; in addition, they cannot contain invocations to methods in a Role Model class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

Role Controller Class Checks. The checks on Role Controller classes, performed by a subprogram called RoleControllerClassChecks, are described below.

1. First obtain the name of the associated Role for this class by removing the substring ’Controller’ from ’Class.name’.
2. For each call we check that:
   (a) If the called class is a Resource class, then
      i. If the called method is an action the pair of values [’called_class’, ’called_method’] must appear in the permissions for the associated Role of the class.
(b) The called class is not the name of a different role’s Role Model class. This is done by checking if the value of 'called_class' contains the substring 'Model', then remove this substring and check that it matches the Role of this class.

(c) The called class is not the name of a different role’s Role Controller class. This is done by checking if the value of 'called_class' contains the substring 'Controller', then the called class must be the same as the value in 'Class.name'.

(d) The called class is not the name of a different role’s Role View class. This is by checking if the value of 'called_class' contains the substring 'View', if so then remove this substring and the following characters in the name up to the end, and check that it matches the role name of this class.

(e) The called class is not the name of a Session class.

Summarising: Role Controller classes cannot contain invocations to methods in Session classes; in addition, they cannot contain invocations to methods in a Role Model, Role View or Role Controller class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

**Role View Class Checks.** The checks on Role View classes, performed by a subprogram called RoleViewClassChecks, are described below.

1. First obtain the name of the associated Role for this class by removing the substring 'View' and the following characters up to the end from 'Class.name'.
2. For each call we check that:
   (a) If the called class is a Resource class, then
      i. If the called method is an action the pair of values ['called_class', 'called_method'] must appear in the permissions for the Role of the current class.
   (b) The called class (i.e., the node 'called_class') is not the name of a Role Model class.
   (c) The called class is not the name of a different role’s Role Controller class.
   (d) The called class is not the name of a different role’s Role View class.
   (e) The called class is not the name of a Session class.

Summarising: Role View classes cannot contain invocations to methods in Role Model or Session classes; in addition, they cannot contain invocations to methods in a Role View or Role Controller class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

**Session Class Checks.** The checks performed by the static verifier on Session classes are specified in a subprogram called SessionClassChecks. For each call, we check that:

1. The called class is not a Resource Class.
2. The called class is not a Role Model Class.

**Other Class Checks.** The checks performed by the static verifier on Other classes, using the tables 'ResourceClasses', 'RoleModelClasses', 'RoleControllerClasses' and 'RoleViewClasses', are as follows. They are specified in a subprogram called OtherClassChecks.

1. No invocation of a Resource Class method.
2. No invocation of a Role Model Class method.
3. No invocation of a Role View Class method.
4. No invocation of a Role Controller class method.
5. No invocation of a Session class method.
7.3 Properties

The static verification algorithm described in the previous sections ensures that the programs that pass the checks do not perform invalid access requests. More precisely, programs satisfy the following properties:

**Definition 14 (OK program).** A program $P$ is OK, written $OK(P)$, if:

1. its actions are 'public' and auxiliary methods are 'private',
2. Resource classes do not invoke methods of a Role Model, Role Controller, Role View or Session class,
3. Role Model methods do not invoke methods in Session, Role Controller or Role View classes or an action that is not allowed by the policy for the associated Role,
4. Role Controller classes do not invoke Session Classes or an action that is not allowed by the policy for the associated Role,
5. Role View methods do not invoke Role Model classes or an action that is not allowed by the policy for the role that the Role View class belongs to,
6. Role classes do not call classes belonging to other Roles;
7. Session Classes do not invoke Role Model classes and do not invoke methods in Resource classes.
8. Other Class methods do not invoke a method of a Resource, Role Model, Role Controller, Role View, or Session class.

**Theorem 1.** If a well-formed program $P$ is accepted by the static verifier, then $OK(P)$.

**Proof.** The algorithm traverses the abstract syntax tree of the program, and for each kind of class, the subalgorithms ResourceClassChecks, RoleModelClassChecks, RoleControllerClassChecks, RoleViewClassChecks, SessionClassChecks and OtherClassChecks ensure that the property holds. We consider each part in Definition 14 in turn.

Part 1 is a consequence of the checks performed in the ResourceClassChecks algorithm (see Section 7.2), specifically checks 1(a) and 1(b) where the program is rejected if the modifiers for methods in the Resource classes are not 'public' for actions and 'private' for auxiliary methods; in check 2, all the invocations within resource class methods are checked to make sure that they do not call on a method from a Role Model, Role Controller, Role View or Session class, as required in part 2.

Similarly, parts 3, 4, 5 and 6 are a consequence of the checks performed in the subalgorithm RoleModelClassChecks, RoleControllerClassChecks and RoleViewClassChecks, respectively.

Part 7 is a consequence of SessionClassChecks checks 1 and 2, which reject a program if in a Session class method there is an invocation of a Resource class or Role Model class.

Part 8 is a consequence of OtherClassChecks checks 1, 2, 3, 4 and 5 respectively, which reject a program if an Other class method contains a call to a method in a Resource, Role Model Role Controller, Role View or Session class, respectively.

**Theorem 2.** If a well-formed program $P$ is rejected by our verifier, then not $OK(P)$.

**Proof.** If the program has been successfully parsed but the verifier rejected it, it is because one of the algorithms ResourceClassChecks, RoleModelClassChecks, RoleControllerClassChecks, RoleViewClassChecks and OtherClassChecks detected an error. If the program is rejected by the algorithm ResourceClassChecks, it is because a method in a Resource class has an incorrect modifier as a result of check 1, or because there is an invocation to a forbidden method as a result of check 2. This could be a method in a Role Model, Role Controller, Role View class or a Session class.

If the program is rejected by the algorithm RoleModelClassChecks, it is because a method in a Role Model class contains an invocation of an action which is not permitted by the policy for role that the Role Model belongs to, as a result of check 2(a), or it contains another forbidden invocation. This could be a method in a Role Controller or Role View class, as an outcome of
checks 2(b) and (c) respectively, or a method of a Role Model class belonging to a different role, as a consequence of check 2(d) or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm RoleControllerClassChecks, it is because there is a call to a forbidden method. This could be an action that is not permitted by the policy, due to check 2(a), or a method in a Role Model, Role Controller or Role View class that belongs to a different role, resulting from checks 2(b), (c) or (d) respectively, or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm RoleViewClassChecks, it is because there is a method in a Role View class which contains a call to an action that is not permitted by the policy, resulting from check 2(a), or there is another forbidden method invocation. This could be a method in a Role Model class, due to check 2(b), or a method in a Role Controller or Role View class that belongs to a different role, the consequence of checks 2(c) and (d) respectively, or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm SessionClassChecks, it is because there is a method in a Session class which contains a call to a forbidden method. This could be a call to a method in a Resource or Role Model class, resulting from checks 1 and 2 respectively.

If the program is rejected by the algorithm OtherClassChecks, it is because there is a method in an Other class which contains a call to a forbidden method. This could be a call to a method in a Resource, Role Model, Role Controller or Role View class or a Session class, resulting from checks 1, 2, 3, 4, or 5 respectively.

Theorem 3. A well-formed program $P$ accepted by the verifier satisfies the policy (see Definition 12).

Proof. To prove that $P$ satisfies the policy according to Definition 12 we need to show that only authorised users with active role $r$ having permission $[Res, m]$ can invoke the action $m$ of an instance of $Res$. Let $res.m$ be a call to $m$ in the program $P$, for which the parser has identified the called class to be $Res$ and the called method to be an action $m$. Since $P$ is well-formed, by Definition 13 it implements the RBAC MVC patterns. Then, a user $u$ can only execute $res.m$ if the user has been authenticated and is in a session, where by Definition 10, one of $u$’s roles, say $r$, has been activated.

By Definition 11, this implies that $r$’s Role Controller has been invoked. Moreover, since $P$ has been accepted by the verifier, by Theorem 1, $OK(P)$. Once the Role Controller for $r$ has been invoked, by Definition 14 the Java code executed from the role classes associated to $r$ contains only invocations to actions $res.m$ that are authorised by the policy, and there are no calls to methods in session classes or methods in role classes belonging to a different role. Moreover, any called method in a class which is not one of $r$’s role class will not contain an invocation to an action (unless it is a method in a resource class) or to a role class (by Definition 14). Therefore once the Role Controller for $r$ has been invoked, all the called methods are authorised. Note that the only classes outside role classes which could call a role class are session classes, which, since the program is well-formed, must satisfy the requirements of the RBAC-MVC pattern. In particular, we trust the calls to Role Classes made in Session Classes.

To provide flexibility to programmers, we have allowed actions to be invoked within Resource classes. However, this means that there may be an indirect violation of the policy, e.g. role $n$ invokes action $a$ which is allowed, which in turn invokes action $b$, which is not allowed, for $n$ in the policy. Currently, we assume that indirect calls are not a policy violation (i.e., the policy specifies the actions that a role is allowed to call, and it does not restrict the invocations within those actions). The Session classes are the critical part of the program in our approach, in which Role class invocations are trusted and not verified. The minimal Trusted Computing Base in our approach is therefore the action methods and the Session classes.

In future work, we will extend the verifier to include checks within actions, to alert programmers if there is an indirect call to an action not allowed by the policy. Our verifier could alert programmers by giving warning messages if this happens.
7.4 Implementing the Static Verifier

Our implementation consists of a JPol policy parser, produced using the ANTLRWorks tool [22], and a static analysis program which are both part of a plug-in we have produced for the Eclipse Integrated Development Environment (IDE) [23]. Eclipse plugins are able to use the Java Development Tools (JDT) plugins provided by Eclipse, which provide an API to simplify static code analysis. In Java, there are three ways to invoke a method; either invoking a (‘static’) method on a class e.g. ‘ClassName.methodOne()’, invoking a method on a variable e.g. ‘x.methodOne()’ or invoking a method on the object returned by another method call e.g. ‘x.methodOne().methodTwo()’. Using JDT we can get the type binding for variables and method invocation expressions, and so we can check if a resource’s actions are being called or if one role’s components invoke another role’s components. This is sufficient to implement all the static checks discussed in Section 7. We have tested our plug-in on a simple doctor’s surgery web database application implemented in Java Enterprise Edition (JEE) (refer to [21] for an overview of JEE). The tool outputs helpful error messages in Eclipse’s editor window, consisting of the class name and line number where the error occurs, the kind of error that has occurred (e.g. ‘Invocation not permitted’) and a description of why that error could have occurred. We show our case study below.

8 Case Study

We have implemented the GP surgery example as a web database application in Java. The data is stored in a database using Apache Derby and the application is built in JEE using Eclipse Integrated Development Environment (IDE) using the Oracle Glassfish application server. The structure of the application follows from the UML diagrams in Figures 11, 12 and 13. We explain the design and implementation, further, below.

8.1 Server-Side Components

The structure of the components to be held on the server follows the RBAC Model pattern. Therefore, the server contains resource classes, which in the case of the GP surgery are Java Persistence API (JPA) entity classes that represent a row in one table in the underlying database, and Session EJB which perform the business logic of the application. The structure of the application is shown in the UML diagrams in Section 6. Session EJBs suffixed with the string ‘Facade’ are responsible for performing database queries for the table that they share their name with. The methods in these Session EJB Facades may use or return instances of the JPA entity class for that table. JPA entities and Session EJB Facades have been stored in the packages ‘model.jpa’ and ‘model.ejb’ respectively, following the RBAC Model pattern’s guideline that packages for resources or role model components should be prefixed with ‘model.’ The server also contains the Role Model classes, which are stored in the package ‘model.roles’ (although there is no requirement that Role Model classes must be stored in their own package or separately from resource classes).

There are two extra Session EJB classes, which both fall outside of the scope of the specification and enforcement of the RBAC policy in our approach. The first of these is ‘LoginBean’, which is used to check the user’s details to authenticate and ‘log in’ the user. We do not restrict the method for authentication and such an EJB would not be expressed as a resource in the policy, so we do not control its implementation or usage. The second is ‘DrugListBean’, which is a simple class used to store data retrieved from the user when they enter a list of entries to put in the table ‘Nhsprescriptiondrugs’ or ‘Privateprescriptiondrugs’. Each entry in the list will contain the ID of the Drug to be prescribed, the quantity to be prescribed and the dosage. The user will enter multiple sets of these values into the user interface (UI), and in order to store them, each set will be an instance of the DrugListBean and the set as a whole is stores as a List in Java. The data inside this EJB is not confidential, since no information on who the prescription is intended for or created by is stored, therefore it does not need to be expressed as a resource.

In the server-side components, some task methods in the Role Model classes have just one line, to call a method from a Resource class. This means that to program the application according
to the design pattern proposed, in some cases extra lines of code are needed to ‘wrap’ a call. However, in our experience few methods needed this, meaning that the overall code additions were very small.

8.2 Client-Side Components: Web Client

In this case, we have chosen to implement Role Controllers as Servlets and Role Views as JSP Pages. Role Controllers are stored in package ‘controller.servlets’ following the package name guideline in the RBAC Controller pattern. Role Views are stored in ‘WEB-INF\view\rolename’. Both of these follow the package/folder naming guideline given in the Role Controller and Role View patterns respectively. The Role Views are effectively stored in folder ‘view\rolename’, but under the folder ‘WEB-INF\’ which is a folder needed by the Web Application Server. Any JSP pages that are placed in this folder will have their URLs hidden from the user by not appearing in the browser’s address bar. This prevents user’s from access JSP pages directly, which provides extra security but is not a restriction that we impose.

The Role Views interact with Role Controllers by passing String parameters to the controller, which represent inputs such as chosen operations to execute, for example clicking a button to ‘viewAllNHSPatients’, or values from text boxes, for example new patient details. The Role Controller checks the received parameters from the Views, and decides what action to take, which could be to invoke a task method contained in the Role Model class. The Role Controller will usually then pass some attribute to one of the Role Views and present it to the user. The Role View would then read any received attribute and present its information to the user along with any other more static behaviour that it executes. An example of this flow is the following. The user may click on a button which passes the String parameter ‘viewAllNHSPatients’ to the Admin-Controller Servlet. This then checks the value of the parameter and will invoke the task method findAllNHSPatients() on an instance of the AdminModel class. This method has the return type ‘List<Nhspatient>‘ which is a Java List of instances of the JPA entity class ‘Nhspatient’ that represents a row from the ‘Nhspatients’ table. The Role Controller passes this List to the ‘Admin-ViewNHSPatientList.jsp’ Role View, which reads each ‘Nhspatient’ object in the list to write its details in a table by calling its attributes e.g. ‘firstname’, ‘lastname’, etc.

8.3 Policy

In a client-server scenario in Java, resources exist as server-side EJB classes whose functionality is shared by several clients - called by Remote Method Invocation (RMI). The policy must do the following:

1. declare each resource,
2. add to each resource each of its actions,
3. declare each role and
4. add to each role the list of permissions it has to invoke actions on resources.

Listings 1.2 and 1.3 show an extract of a sample policy for the GP surgery example for Resources and Roles, respectively.

**Listing 1.2.** Extract of Sample JPol Policy declaring Resources with their actions

```java
// (1) Declare Resource "Nhspatient"
Resource nhspatient = new Resource("Nhspatient");

// (2) Add each of its Actions
nhspatient.addAction("Nhspatient");
nhspatient.addAction("setFirstname");
nhspatient.addAction("getFirstname");
nhspatient.addAction("setLastname");
nhspatient.addAction("getLastName");
nhspatient.addAction("setDob");
```

From the extracts, we can see some key actions of a resource "Nhspatient" that have been declared. These correspond to the the design of the class "Nhspatient" in the UML diagram in fig 11. The Resource classes and their associated actions must all be declared, because the verifier will check that these classes have implemented the actions as per the policy. We can also see two Roles have been declared. Firstly, "NHSDoctor" which we show has permissions on (all of the actions of) the Resource "Nhspatient". Secondly, "PrivateDoctor" which we show has no permissions on the Resource "Nhspatient".

### 8.4 Applying Verification

Let us first discuss the case where the program passes the checks. If there are no errors found by our verifier, then, according to the policy extracts, classes belonging to PrivateDoctor will not have any calls to methods in Nhspatient class, but NHSDoctor may. Now, we can modify the program to show, from these small extracts of the policy, examples of the kinds of errors that can be caught at compile-time by our static verifier.

**Undefined Action.** In this type of error, a public method has been implemented in a Resource class in the program but not defined as an action in the policy. For example, in Listing 1.2 if we remove line 5, the action "setFirstName" would be undefined. When our verifier checks the program, it would read a public method named "setFirstName" in the Nhspatient class. Since it is a method of a Resource, it must either be a private method - auxiliary - or must exist as an action in the policy for the resource Nhspatient, else the program will be rejected. This prevents public methods of Resources from being ignored from the policy and program validation process; the main objective of our access control approach is to restrict invocation of visible methods of resource classes. Figure 16 shows this error being caught and the resulting error message.

**Invocation Not Permitted.** In this type of error, an action of a Resource is called in a class belonging to a Role which is not permitted for that role according to the policy. For example, we restore line 5 in Listing 1.2, then in the class "PrivateDoctorModel", we invoke the action "setFirstName" on the Resource "NhsPatient". We did not assign any permissions on "Nhspatient"
Fig. 16. Example of Undefined Action error

to Role "PrivateDoctor", and our verifier reads the invalid invocation and rejects the program. The error and resulting error message is shown in Figure 17.

**Invocation Between Roles.** In this type of error, a class belonging to Role $x$ calls a method in a class belonging to a Role $y$, where $x \neq y$. For example, Role Model class "PrivateDoctorModel" containing an invocation to a method “createNHSPrescription” of the "NHSDoctor" Role Model class. Figure 18 shows this error and the resulting error message when the verifier detects it.

9 Evaluation

The static approach to access control enforcement has limitations. Firstly, the policy cannot change after compilation which prevents administrative changes in policies such as changing role permissions. Secondly, permissions cannot be based on any information that changes at run-time. For flat or hierarchical RBAC policies, this is sufficient. For more general versions of RBAC, these two restriction can be relaxed by combining the static approach with a dynamic one, producing a
hybrid approach which will be the subject of future work. User-to-role assignments can change in our approach and these tend to change more often than permission-to-role assignments. Thirdly, the program code must be available at compile-time in order to analyse it statically.

The static enforcement that is a result of our approach mitigates several of the disadvantages of using a reference monitor. One of these is that the run-time costs associated with dynamic enforcement are avoided since all access-checks are done at compile-time. Having said that, it is difficult to compare the performance of a program designed using any given pattern and the same program designed without using it. Performance is not usually taken into consideration when designing a pattern, especially in the case where performance gains are not the main goal of a pattern - as in our approach. We can be sure that in our approach, policy enforcement will have no impact on run-time resources, since no access-checks will be made at run-time. Another advantage of our approach over a dynamic one is that the testing phase in the application life-cycle can be simplified and reduced with regards to checking a program for policy compliance; all dynamic tests related to checking if the dynamic enforcement of the policy works correctly can be avoided because all checks are done at compile-time. Furthermore, some mistakes in the Access Control policy can be caught earlier, at compile-time, which is preferable to discovering mistakes at run-time.
The limitations of the static approach do not mean that it is not useful. A policy commonly contains some static parts and some dynamic parts (even though these may not be clearly separated in the policy). For example, just after log-in, it is highly likely that some static part of the policy will be in effect at that time. Therefore, the static approach can be used in combination with dynamic checks within a hybrid checker.

The use of patterns to guide the implementation towards a program that can be statically verified for policy violations is a strength of our approach. Generally, patterns provide a solution to a particular recurring design problem. As such, adding access control into patterns for the design of the target program yields a uniform methodology to design the application integrating access control into the business logic. Access control is usually added on-top of an application at the last stage, enforced using a reference monitor, whereas using these patterns can guide the integration of the access control into the code. As a result, the program can be verified statically.

The drawbacks of using patterns include that the approach is less flexible since the program must follow the rules of the pattern. This can, however, be said of any approach that relies on the use of specific patterns. Using our design patterns, it takes an initial effort for an architect/programmer to design/implement an initial set of Resource classes and a set of Role MVC classes for one role. However, after this initial stage, designing/implementing the program becomes easier than without using the patterns. Our patterns help to relate the functionality of the program...
with the roles that can access that functionality. Adding this related functionality becomes easier - achieved just by adding more sets of Role MVC classes. Our pattern also helps in the design of resources because it helps to clearly separate the resources from the rest of the program. Current limitations of our patterns are that roles that have many similar operations will require completely separate Role MVC classes, possibly duplicating code. Moreover, role hierarchies are declared in the policy but not reflected in the design of the program; the permissions of a subordinate role are copied to the senior one and this data is used in the static analysis only. Reflecting role hierarchies in the program would reduce code duplication, which we intend to address in future work.

The approach proposed in this paper for the development of web-based applications with RBAC policies, based on the use of RBAC MVC patterns and a static enforcement algorithm, provides an easy-to-use methodology for programmers to integrate RBAC access control policies into their applications and enforce them statically. It can be applied to a variety of languages, not just JEE, by small modifications to the static verifier that are dependent on the details of the chosen language. For example, the naming restrictions in Section 7.1 would have to be altered depending on the naming conventions of the chosen language. Also, the concrete syntax of the policy language can be modified depending on the needs of the user/policy author/programmer. The version presented in this paper is similar to Java, but it could be altered to follow any OO language syntax, or even natural language which would help a policy author that is uncomfortable with programming syntax. Additionally, the RBAC MVC patterns are already independent of implementation language. The only requirement is that the chosen language must have access modifiers for methods.

10 Related Work

10.1 Java Web Security

JEE web security contains a means for RBAC to be implemented using dynamic access control enforcement. There are two types of access control using roles: declarative and programmatic security. Declarative security is a means for securing Servlets or EJBs using the Server. Simply put, the programmer specifies the roles that are allowed to access specific Servlets/EJBs or methods within them, and when a user invokes these, the server performs authentication dynamically to check if the user has the required role. In programmatic security, the programmer specifies dynamic access checks within the code. They secure parts of the code by using provided methods such as `isUserInRole()` to specify that only certain roles can run that code. These are both dynamic enforcement mechanisms, whereas our approach is static.

10.2 Design-Level Security

There has been significant developments in the area of security at the design level. The general approach in this area is to specify security restrictions at the design stage of a program’s lifecycle. One such approach is described by Basin et al in [17], where they propose Model Driven Security as an approach to building secure systems. In their approach, security policies are specified at the design level using UML notation on a Model of the system, which is then transformed into an application with security enforcement code generated for a specific implementation platform. They give an example of a static RBAC policy which is transformed together with the target program into an EJB JEE program. The security code generated utilises Java Web security mechanisms. These mechanisms are dynamic (as discussed above), therefore the key difference between approaches like these and our approach is that we enforce policies using only static checks.

10.3 Security Patterns

There exists a body of work on security patterns. In [13], the authors describe access control, specifically RBAC and Metadata-based Access Control, using patterns and run time checks. However, the approach of our patterns is towards implementing RBAC via only static checks in JEE.
applications. Steel, Nagappan and Lai [11] propose several security patterns that are specifically targeted towards securing JEE applications. Our goal is to enforce RBAC in JEE applications using only static checks, avoiding all the performance costs associated with dynamic checks. However, their work takes a dynamic approach to enforcement. Many of their patterns can and should be used in conjunction with our patterns to secure the overall application aside from enforcing RBAC statically, such as the Intercepting Validator to properly check parameters before executing transactions and Secure Pipe to properly secure the connection between Client and Server or Server and Server.

10.4 Annotation-Based Approach

Zarnett et. al. [12] enforce RBAC in Java using an annotation-based approach on proxy objects in RMI. Their work has the effect of removing the need for run-time access control checks, however their approach relies on annotations, which has several weaknesses. Understanding where a specific annotation should go can be a difficult task, especially in large programs. Moreover, specifying the policy via annotations leaves the policy fragmented throughout the program. In our approach, we check that the policy has been implemented correctly, e.g. that all the resources and roles have been implemented, however they have no such verification techniques since there exists no central policy specification, which means that errors are discovered later (at run time). Also, recalling the model in Figure 4, their approach enforces access restrictions at the level of the 'Resources', by creating proxy objects containing only those methods which are authorised for the currently active role. Our approach enforces access control at the level of 'Tasks', where instead of creating proxy objects of each resource for each role all authorised methods for each role are provided by a user-interface in a role object. Another key difference is that we assume a system where server and client code are available and can be checked at compile time.

10.5 Hybrid Enforcement Approach

Bodden et. al. [20] enforce security properties in programs using a hybrid approach. They generate code for run-time checks, then perform a series of compile-time analyses to determine which dynamic checks can be removed. In their approach, the access control enforcement of (static) roles would not be possible at compile-time. This is because they cannot determine, at compile-time, the access requests that each role can make. Our design pattern solves this. Therefore, in their approach, a static RBAC policy would be enforced using only run-time checks.

11 Conclusion and Future Work

We have described a new system to statically check that a target program respects its RBAC policy. Our system firstly consists of a policy language, JPol, in which a static RBAC policy can be expressed restricting the invocation of protected methods (actions) by roles. Secondly, the target program must implement a set of patterns, which we name RBAC MVC, in order for it to be verified statically. These utilise the well known Model-View-Controller pattern but incorporate RBAC notions. They specify that there must be a class for each resource (which we called the Resource class), and a model, controller and a set of views for each Role (which we called the Role Model class, Role Controller class and Role View classes). There is also a pattern for implementing the concept of users having multiple roles per session. These patterns help to incorporate RBAC security requirement into the design of the program and aid the process of statically verifying that the program respects the policy. The last part of our system is a static verification algorithm which we described at a high-level. This analyses the source code of the program to ensure it contains no unauthorised invocations. Put simply, it checks that classes belonging to a role can only invoke methods in each other (not classes belonging to other roles), or those actions in Resource classes that are permitted for that role by the policy. Any classes that are not Resource classes or do not belong to a role cannot invoke methods in these classes.
If the program contains an unauthorised invocation to a Resource class action or a method in a class belonging to a Role, then it will be rejected by the static verification algorithm. If the program successfully passes the static verifier’s checks, then when using the program, the logged in user can only call those methods that have been authorised for the role currently activated for them, through the Task methods in that role’s Role Model class. Therefore, no run-time access checks are needed. The static verifier catches policy violations early, at compile-time, aiding programmers in implementing policies correctly and nullifying run-time overheads attributed to dynamic enforcement mechanisms such as reference monitors.

In future work, we will extend the static verifier with more checks within actions, to provide warnings to the programmer when there are indirect calls to actions within actions. We intend to write the static verifier in the form of a type checker. Our system will also be adapted to other platforms in addition to JEE, since the underlying concepts are independent of implementation platform (other than the requirements for an OO language with access modifiers). We will also explore adding dynamic conditions in the policy, and generating code in the program to check these at run-time. The result would allow static parts of the policy to be enforced statically, whilst still allowing dynamic policies to be expressed and then enforced dynamically. We aim for our policy language to be able to support more dynamic access control models, particularly Category-Based Access Control [18], whose policies are then enforced using a combination of static and dynamic enforcement mechanisms.

References

10. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley-Longman, 1995.

