# IMPLEMENTATION OF ACTIVE ROLE BASED ACCESS CONTROL IN A COLLABORATIVE ENVIRONMENT

**W. Caelli[1], A. Rhodes[2]**

## *Abstract*

Although dominant literature exists that strongly suggests that RBAC is at the point of maturity and will be become predominant technology, there are limitations that currently restrict RBAC from evolving and reaching its full potential. While current RBAC models successfully allow organizations to model security from an enterprise perspective, the dynamic authorization needs of an organizations are not facilitated. The dynamic nature and continual change of authorizations need to be accommodated by RBAC models before they are widely accepted and applied as the ideal security paradigm for organizations.

All commercial Workflow Management Systems (WFMSs) that currently support role-based authorizations are passive in nature. In particular, they are not capable of modeling authorization constraints on roles. By introducing a constraint specification language we can provide an active role-based security model that provides very tight, just-in-time permissions suitable for use in WFMSs.

The overall aim and intent of the research reported herein has been to investigate current RBAC models and the possible methods that can be applied to transform "passive" models into "active" models such that they can fulfil the current security requirements of collaborative environments.

*Keywords:* Security, Role Based Access Control, Collaborative Systems, Role Language

## *Introduction*

Although dominant literature exists that strongly suggests that RBAC is at the point of maturity and will be become predominant technology, there are limitations that currently restrict RBAC from evolving and reaching its full potential. While current RBAC models successfully allow organizations to model security from an enterprise perspective the dynamic authorization needs of an organizations are not facilitated. The dynamic nature of enterprise security parameters and continual change of authorizations need to be accommodated by RBAC models before they are widely accepted and applied as the ideal security paradigm for organizations.

Specifying authorizations on roles is not only convenient but reduces the complexity of access control because the number of roles in an organization is significantly smaller than that of users. Moreover, the use of roles as authorization subjects, instead of users, avoids having to revoke and re-grant individual authorizations whenever users change their positions and/or duties within the organization.

However, it has been discovered that the currently accepted notion of RBAC is not ideally suited to model the security needs of most organizations [1]. More sophisticated models are required to control access in collaborative environments; situations where sequences of operations need to be governed, such as workflow management systems (WFMSs).

This requirement to control access in situations where sequences of operations need to be governed demands investigation into higher level models that are "active" in nature. Most well known access control models are considered to be passive in nature. These models do not distinguish between permission assignment and activation. Furthermore, passive security models are not capable of representing or considering any levels of context when processing an access operation on an object. Thomas [2] expects "active security concepts to be an important area of research and believes they will influence the evolution of RBAC".

[1] School of Data Communications, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia.
[2] School of Computing Science, Queensland University of Technology, GPO Box 2434, Brisbane 4001, Australia.

Two recent studies [3], [2] investigated the use of active RBAC models in collaborative environments. Both studies present ideas that argue RBAC is a natural method to model the security requirements of collaborative environments. The presented arguments are well supported as many collaborative environments model their security needs in a way that closely resembles RBAC. That is, collaborative environments encompass the concept of roles, users and role authorization. The problem however, is that organizations commonly express their security policies as constraints on the users and roles within the organization. A typical constraint that may be enforced is separation of duty. Unfortunately, current RBAC models are not yet adequate to satisfactorily model the security requirements of collaborative environments.

Both research efforts found that the currently accepted IEEE RBAC model defined by Sandhu et al [1] would require refinements and additional notations if RBAC were to be to effectively applied in a collaborative environment. Thomas [2] states “it remains to be seen whether these requirements should lead to yet another variation of one or more models of RBAC, or whether such requirements should form another access control model layered on top of RBAC”.

In summary, the overall aim and intent of this research effort will be to investigate current RBAC models and the possible methods that can be applied to transform passive models into active models such that they can fulfil the current security requirements of collaborative environments.

## Background

### Role Based Access Control

RBAC is a security mechanism devised to assist and simplify security administration and review. Within an RBAC framework operations are associated with roles, and users are made members of roles. Each role defines a specific set of operations that the individual acting in that role may perform. The user acquires the permissions of the role in which they are a member. Therefore, the operations that a user is permitted to perform are based on the user's role [4].

Security administration is costly and prone to error because administrators usually specify access control lists for each user on the system individually. RBAC simplifies management of authorization while providing an opportunity for greater flexibility in specifying and enforcing enterprise specific policies. Another advantage is that RBAC supports three well-known security principles: least privilege, separation of duties, and data abstraction [1], [5].

RBAC is ideally suited for use in organizations [6]. The use of roles to control access can be an effective means for developing and enforcing enterprise-specific security policies that map naturally to an organization's structure. It allows and promotes security to be managed at a level that corresponds closely to an organization's structure and simplifies security management.

In such an environment, roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed.

For further details on RBAC refer to papers mentioned in the Bibliography.

### Passive and Active Security Models

A passive security model primarily serves the function of maintaining permission assignments, such as current RBAC models where permissions are assigned to roles. Once a permission is assigned to a subject, the subject always possesses that permission independent of any other considerations such as context or history of events. In these models, security information (such as a specification of which subject can access which object) are stored and access control requests are validated against this information. Unfortunately, this information represents independent and primitive access control information distanced from application logic as well as from any emerging context associated with ongoing tasks, work units and processes.

On the other hand, an active security model distinguishes between task- and context-based permission activations from permission assignments. The permissions that a user possesses may change over time depending on the history of events and context. These models consider the overall context associated with tasks before validating authorizations.

## Problem Formulation

### Limitations of Current RBAC Models

It has been discovered that the currently accepted notion of RBAC is not ideally suited for the security needs of most organizations [1]. More sophisticated models are required to control access in situations where sequences of operations need to be governed, such as those found in collaborative environments.

The ability and ease provided by RBAC to model the static authorization requirements of an organization has been a contributing factor behind the recent interest in RBAC [6], [7], [1]. However, organizations commonly express their security policies as constraints on the users and roles within the organization. A typical constraint that may be enforced is separation of duties. Unfortunately, current RBAC models are not adequate to enforce the constraints modeled by an organization's security policy.

A security requirement that is essential for organizations to provide a high level of system integrity is the principle of least privilege. Such a constraint ensures that authorized users gain access to required objects only during the execution of the associated task and prevents access at all other times (active security model). The granting and revoking of privileges needs to be synchronized with the progression of the tasks within a collaborative environment and/or over time. Most access control models allow the granted access from the time of assignment until the privileges are manually revoked. Therefore, a user possesses privileges to access the objects even though the user has completed or not yet begun the task, potentially compromising security.

The requirement of RBAC models to handle constraints is highlighted by the recent study conducted by Thomas [2]. The research which investigated the security issues for clinical workflows associated with patient care revealed the need:

1. for a hybrid access control model that incorporates the advantages of broad, role-based permissions across object types, yet provides fine-grained, identity-based control on individual users in certain roles and to individual objects;
2. to distinguish the passive concept of permission assignment from the active concept of context-based permission activation.

The inability of current RBAC models to enforce these requirements means that they must be enforced and implemented as application code and embedded into the various tasks. This approach makes the specification and management of authorization constraints difficult, if not impossible.

### Recent Research Addressing RBAC Shortcomings

Several research efforts [2], [3] have identified RBAC as a security model that would be well-suited in collaborative environments. However, the passive and rigid nature of current RBAC models present problems that prevent a natural integration. In particular, current RBAC models do not allow fine-grain control on individual users in certain roles and to individual objects. RBAC also provides no support for the context associated with collaborative tasks. Thomas' research [2] concluded that "additional notations are necessary to effectively apply RBAC in a collaborative setting".

The research by Thomas [2] provides preliminary ideas and concepts on the notion of Team-Based Access Control (TMAC) which is an approach to allow fine-grain control on individual users in certain roles and to individual objects. TMAC is an active model of access control that automatically assigns and removes permissions as teams are collaborating and the workflow progresses. TMAC provides the advantage of being able to offer the administrative and modeling advantages of RBAC and yet provide fine-grained control over permission activation to individual users and objects.

To address the limitations of current RBAC models to model constraints, Bertino et al [3] have proposed a language for specifying the various types of constraints. Bertino et al [3] also proposes to undertake future research including the modeling of temporal and event-based constraints, the support of more complex role models and the integration with advanced authorization models.

Both research efforts found that the currently accepted IEEE RBAC model defined by Sandhu et al [1] would require refinements and additional notations if RBAC were to be to effectively applied in a collaborative environment. Thomas [2] states "it remains to be seen whether these requirements should lead to yet another variation of one or more models of RBAC, or whether such requirements and concepts should form another access control model layered on top of RBAC".

## A Suggested Solution: Role Language

Barkley [8] noted that access conditions (constraints) for roles are located exclusively within the role classes. Therefore, role policy changes do not require modifications to the applications themselves. This leads to the natural idea of constructing a role language [2], [3], [5] that can define constraints on role and user assignments. A role language mechanism may be utilized with an RBAC model to deal with the dynamic nature of roles and real-time changes in authorization. The ability to easily change access conditions associated with roles permits rapid (dynamic) response to organizational policy changes in a transparent fashion to applications.

A role language seems the appropriate software tool to model an organization's security policy. The role language needs to be simple enough that data and security administrators can use it. The role language would provide a model that achieves synchronization between authorization flow and collaborating tasks. The authorization flow needs to be tightly coupled to the collaborating tasks in order to ensure users possess authorizations only when required.

# ***Role Language Design***

## Requirements

Research conducted by Bertino et al [3] has proposed a formal model that includes:

1. A language for defining authorization constraints as clauses in a logical program (role language).
2. Formal notions of constraint consistency.
3. Algorithms to check for the consistency of the constraints and to assign roles and users to the workflow tasks in such a way that no constraints are violated.

This paper outlines activities undertaken to implement and extend this formal model. In particular, the proposed role language will be investigated as a possible method that can be applied to transform passive models into active models such that they can fulfil the current security requirements of collaborative environments.

Bertino et al's proposed role language is only formally expressed as clauses in a logic program and there is no evidence presented of it actually being defined in a high-level language suitable for security administrators. Therefore, this paper outlines research undertaken to implement and test the logic program to provide a proof-of-concept model that will illustrate the strengths and weaknesses of such an approach. Hopefully the developed proof-of-concept model will be used as a foundation for further research into active security models.

The collaborative environment which we will concentrate on, and incorporate the role language into, will be a workflow management system (WFMS). The focus of this project is the security model itself and for this reason many, if not most, of the contemporary issues of WFMSs are ignored. The designed system implements a very simple WFMS in which the active RBAC model is incorporated. However, there is no problem preventing the proposed solution from being incorporated into more sophisticated WFMSs.

Focusing on the research conducted by Bertino et al [3] this paper also addresses the second finding of Thomas' research [2]. Reiterated, findings by Thomas [2] revealed the need:

1. for a hybrid access control model that incorporates the advantages of broad, role-based permissions across object types, yet provides fine-grained, identity-based control on individual users in certain roles and to individual objects;
2. to distinguish the passive concept of permission assignment from the active concept of context-based permission activation.

## Evaluation of Success

Development of a prototype has been the immediate measure for success of this project. A successful prototype provides a proof-of-concept model that will demonstrate and highlight the benefits of a role language. Two case studies are provided to demonstrate the developed prototype. It is also envisaged that this work will provide a solid foundation for continued investigation into higher level models that are active in nature.

## Proposed Solution

Workflow management systems (WFMSs) have gained popularity both in research as well as in commercial sectors in recent years. WFMSs are used to coordinate and streamline the business processes of an organization. To simplify the complexity of security administration, it is common practice in many business organizations to allocate a role to perform each activity in the process and then assign one or more users to each role, thus granting an authorization to roles rather than to users. Typically the security policies of the organization are expressed as constraints (or rules) on users and roles e.g., separation of duty. Unfortunately, current RBAC models are not adequate to model such constraints.

A number of commercial WFMSs such as Lotus Notes and Action Workflow support role-based authorizations. However, a common drawback of role-based authorization models used in current WFMSs (or even DBMSs) is that they are not capable of modeling authorization constraints on roles and users.

If no proper support is provided, constraints like “separation-of-duties” must be implemented as application code and embedded into the various tasks. Such an approach makes it difficult, if not impossible to control the specification and management of authorization constraints, given the large number of tasks that typically occur in a workflow.

To address this issue, Bertino et al [3] propose a formal model that includes:

1. A language to express both static and dynamic authorization constraints as clauses in a logic program.
2. Formal notions of constraint consistency.
3. Algorithms to check for the consistency of the constraints and to assign users and roles to tasks that constitute the workflow in such a way that no constraints are violated.

This formal model will be explored, used and, in many cases extended, to provide the foundation of the prototype system. The prototype system will produce:

1. A high level language for encoding constraints on user and role assignments.
2. Algorithms to ensure constraint consistency.
3. Algorithms to assign roles and users to the workflow task in such a way that no constraints are violated.

### Note on the Contribution by Bertino et al

Bertino et al [3] present a language for defining constraints on role and user assignment to tasks in a workflow. Such a constraint language supports, among other functions, both static and dynamic separation of duties. Because the number of tasks and constraints can be quite large, a relevant issue is provide some formal notions of constraint consistency and devise algorithms for consistency checking. Bertino et al [3] show how such constraints can be formally expressed as clauses in a logic program, so that all results can be exploited in the logic programming and deductive database area. A further contribution by Bertino et al [3] is the development of an algorithm for planning role and user assignments to the various tasks. The goal of the planner is to generate a set of possible assignments, so that all constraints stated as part of the authorization specification are satisfied. The planner is activated before the workflow execution starts to perform an initial plan. This plan can be, however, dynamically modified during the workflow execution, to take into account specific situations, such as the abort of a task. Bertino et al [3] believe that “this is the first approach proposed to systematically address the problem of assigning role and users to tasks in a workflow”.

# ***Role Language Implementation***

## Analysis

This section documents the findings of the analysis, providing an understanding of the prototype to be developed.

### System Overview

The prototype system provides:

1. A high level language for encoding constraints on user and role assignments.
2. Algorithms to ensure constraint consistency.
3. Algorithms to assign roles and users to the workflow task in such a way that no constraints are violated.

The prototype system relies on the system's current RBAC setup. An RBAC prototype system, *RBACManager,* has been developed and described elsewhere, [9]. This RBAC security prototype system has been modified to write the RBAC setup to a file that this experimental system can read. Although this approach is adequate for the needs of this project, a system incorporating this functionality would be the desired outcome.

## System Architecture

As illustrated in Diagram 1, the system architecture is comprised of the following components, which are briefly discussed below:

- WFMS
- Constraint Analysis and Enforcement Module
- Constraint Base
- User-Role Assignment
- Role Hierarchy
- Task Processing Entities
- Authorization Constraints
- User Requests

**Diagram 1: System Architecture**

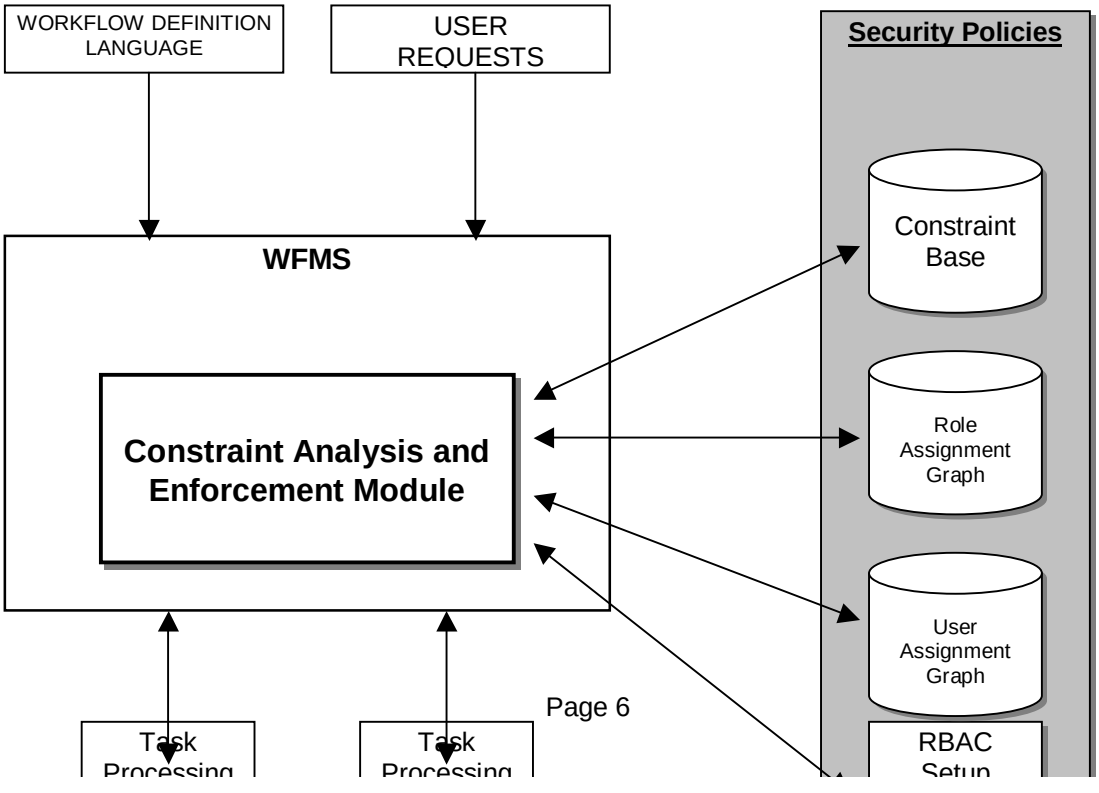
### WFMS

A workflow separates the various activities of a given organizational process into a set of well-defined tasks. Typically a workflow is specified as a set of tasks and a set of dependencies among the tasks. The task dependencies can be specified based on the task primitives such as begin, abort and commit of a task, based on the outcome/result of a task, or based on certain external parameters such as time.

The Workflow Management System (WFMS) is therefore responsible for the scheduling and synchronization of the various tasks within the workflow in accordance with the specified task dependencies, and for sending each task to the respective processing entity. The information concerning the role hierarchy and the user/role assignments are maintained by the WFMS.

### Constraint Analysis and Enforcement Module

The Constraint Analysis and Enforcement Module provides the functionality required to incorporate an active security module into WFMSs. The Constraint Analysis and Enforcement Module determines the role and user assignments for each task either in advance and as the workflow execution progresses. When a user submits a request to execute a task to the WFMS, the WFMS verifies whether the user can be authorized to execute the task according to the user/role assignments computed by the constraint analysis and enforcement module.



The Constraint Analysis and Enforcement Module executes a number of activities. The activities, which are explained in the section detailing the Constraint Analysis and Enforcement Module, are:

- Static Analysis Phase.
- Pruning Phase.
- Planning Phase.
- Runtime Phase.

## Constraint Base

The Constraint Base (CB) contains the set of rules that encode the constraints defined on role and user assignments in the workflow. The rules in the CB are generated by the Workflow Definition Language provided by the security administrator. Rules are also generated by the WFMS and inserted in the CB as the workflow executes.

## Role Assignment Graph

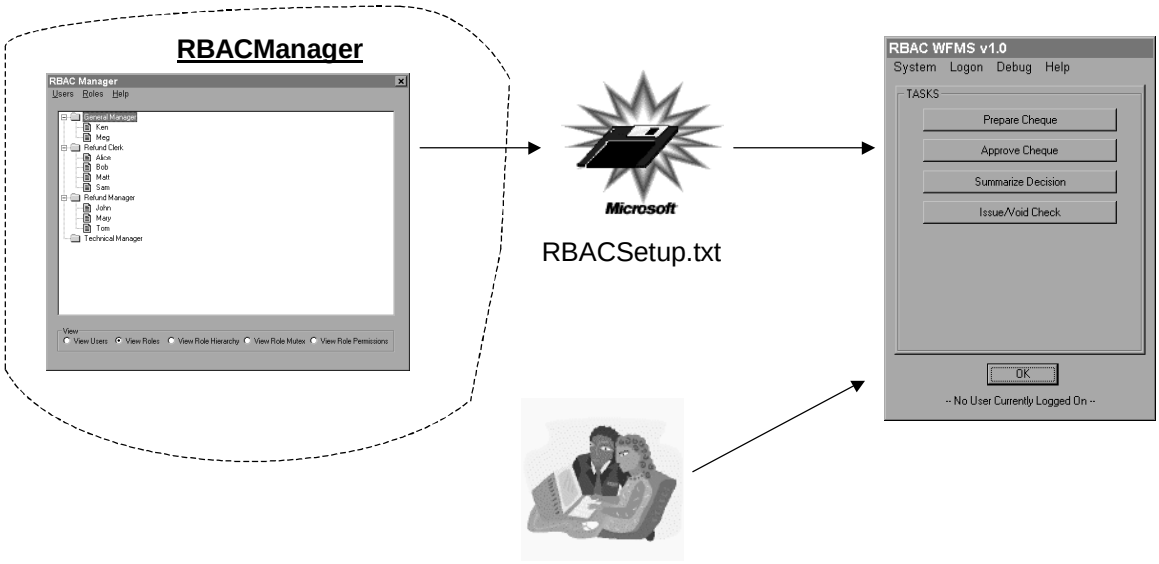
The Role Assignment Graph maintains the possible role assignments to each task in the workflow in such a way that no constraints are violated. This graph is generated before the workflow begins and is pruned as the workflow is executed.

## User Assignment Graph

The User Assignment Graph maintains the possible user assignments to each task in the workflow in such a way that no constraints are violated. This graph is generated before the workflow begins and is pruned as the workflow is executed.

## RBAC Setup

The designed system relies on the system's current RBAC setup (i.e., roles, users, permissions & assignments). The system will simply be provided with a text file produced by *RBACManager*. The following diagram shows the two inputs that the system requires: RBACSetup.txt and Workflow Definition Language.




## Task Processing Entities

Each task defined in the workflow is associated with a Task Processing Entity. A Task Processing Entity is simply an executable program (or component) that will be invoked by the Constraint Analysis and Enforcement Module if the requesting user is authorized.

## Workflow Definition Language

The Workflow Definition Language is a high level language suitable for use by security administrators. The Workflow Definition Language is used to:

- Define the workflow.

- Encode (static and dynamic) constraints imposed on role and user assignments.

The Workflow Definition Language is described after the section on the Constraint Analysis and Enforcement Module section.

### User Requests

A user interacts with the WFMS by requesting a task execution. The WFMS then forwards the request to the Constraint Analysis and Enforcement Module. The Constraint Analysis and Enforcement Module determines whether the user playing a certain role is allowed to execute the requested task according to the possible assignments specified by the User Assignment Graph and Role Assignment Graph. If the user is allowed the Task Processing Entity is executed.

## Constraint Analysis and Enforcement Module

The **constraint analysis and enforcement module** is responsible for assigning roles and users to the tasks of a workflow according to the constraints specified by the workflow specification language. Diagram 2 illustrates the phases executed by the Constraint Analysis and Enforcement Module throughout the lifetime of a workflow.

The fist phase, referred to as **Static Analysis**, determines whether the static part of the CB, that is, the subset of the CB containing only static rules, is consistent. A CB is consistent if and only if the constraints it encodes are satisifable. If the check fails, the constraints specified for the workflow are inherently inconsistent, and therefore no assignment to tasks is generated. Thus, the system security officer has to modify role assignments to tasks and/or the constraints.

If the static analysis phase succeeds, the **Pruning** phase is executed. This phase modifies the workflow role specification to take into account the results of the static analysis phase. Moreover, the CB is modified to eliminate redundant rules. Pruning will make the execution of the subsequent phases more efficient.

The **Planning** phase receives the modified workflow and the modified CB generated by the pruning phase as input, and generates roles/users assignments to tasks that satisfy the constraints, that is, roles/users that, when assigned to tasks, make the CB consistent. If no assignment can be generated, an error is returned to the system security officer. To limit the number of choices to be evaluated by the planner, we make the assumption that all the tasks successfully execute.

At **Run-time**, when a task aborts, constraints involving abort predicates are analyzed to verify whether the planner must be invoked again. The planner is also re-activated if the number of activations of a task exceeds the number stated in the workflow role specification.

### Safety Factor

It is quite common to most organizations that the number of users is much larger than the number of roles. Therefore, it may not be efficient for the **Planning** phase to perform an exhaustive check of all possible users who are allowed to execute the tasks in the workflow. This is especially true in cases where roles have a large number of users authorized to play them. Thus, instead of using an exhaustive approach, we allow the security administrator to specify a safety factor. The security administrator may decide on this value based on several parameters such as allowed number of simultaneous workflow executions, percentage of available users among the total users at any given time, etc.

User planning is performed for only those roles, $R_i$, such that the following formula is not satisfied:

$$U_{worstcase}(R_i) + safety\_factor <= U_i$$

where:

- $U_{worstcase}(R_i)$
  The maximum number of users required, for each role $R_i$, to execute the workflow tasks according to the plan.
- safety_factor
  The safety factor provided by the security administrator.
- $U_i$

The number of users authorized to play role $R_i$.

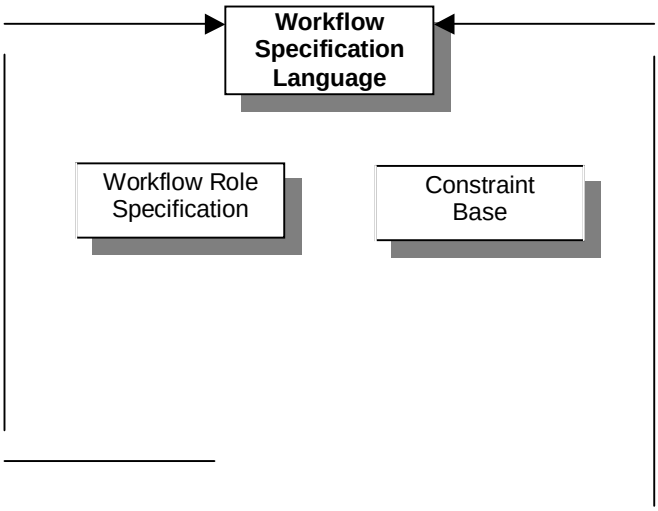
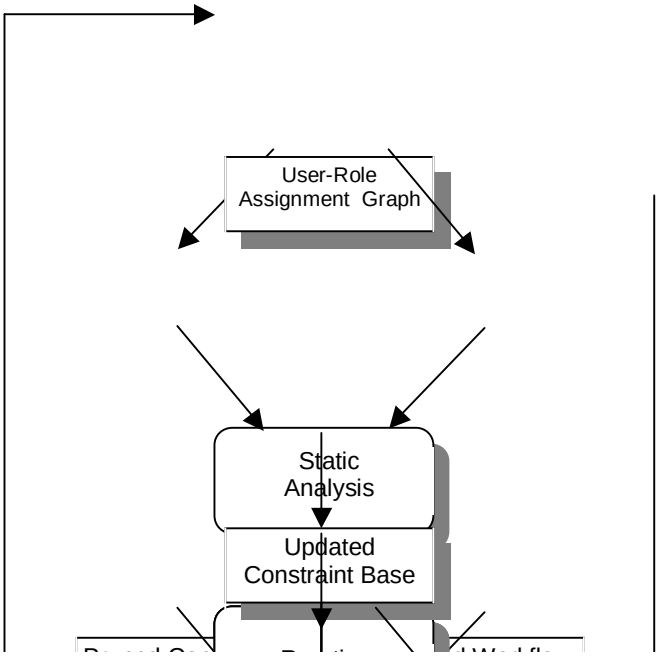



**Diagram 2: Constraint Analysis and Enforcement Module Phases.**

## Workflow Definition Language Overview

Bertino et al [3] present a normal logic program for defining constraints on role and user assignments to tasks in a workflow. The prototype will transform this normal logic program into a high-level language suitable for use by system security officers. For this prototype the high-level language will also be extended to allow the workflow to be defined. This resulting language has been named the **Workflow Definition Language.**



The Workflow Definition Language will define the workflow system and constraints placed upon role and user assignments. An awareness of the following components is recommended to utilize and understand the Workflow Definition Language:

- Workflow Role Specification;
- Constraints;
- Formal Model of Constraints;
- Rules;
- Constraint Base.

The high level language resulting from the combination of these components is described later in the Design Section. Each component is described below.

## Workflow Role Specification

A workflow role specification defines the workflow and associates roles with the tasks in the workflow.

A **Workflow Role Specification** associates roles with tasks in a workflow.

Workflow Role Specification = $\{TRS_1, TRS_2, \ldots TRS_n\}$

*where*

each $TRS_l$ is a 3-tuple $(T_i, (RS_i, \succ_i), act_i)$

*where*

$T_i$ is a task

$RS_l$ is the set of roles authorized to execute $T_l$

$\succ_i$ is a local role order relationship

acti is the number of activations of task $T_i$

**Example,**

*W* = { (T1, ({Refund Clerk}, {}), 1),
(T2, ({Refund Clerk, General Manager}, {}), 2),
(T3, ({Refund Clerk, General Manager}, {}), 1),
(T4, ({Refund Clerk}, {}), 1)
}

## Constraints

Constraints on role and user assignments to tasks in a workflow can be of several different types. The constraints have been categorized into three main categories according to the time at which they can be evaluated.

**Static Constraints**. These constraints can be evaluated without executing the workflow.

**Dynamic Constraints**. These constraints can be evaluated only during the execution of the workflow, because they express restrictions based on the execution history of the workflow.

**Hybrid Constraints**. These are constraints whose satisfiability can be partially verified without executing the workflow.

## Formal Model of Constraints

Bertino et al [3] represent constraints as clauses in a normal logic program in order to provide a semantic foundation for a constraint model and formally prove consistency. Bertino et al [3] formally define a language to express constraints placed upon a workflow.

A rule is of the form $H \leftarrow A_1, \ldots A_n$, not $B_1, \ldots$ not $B_m$, $n, m \geq 0$

where H, $A_1, \ldots, A_n$ and $B_1, \ldots B_m$ are atoms and not denotes negation by failure. H is the head of the rule, whereas $A_1, \ldots A_n$, not $B_1, \ldots B_m$ is the rule body.

Rules that can be expressed in the language can be classified into a set of categories according to the predicate symbols they contain. The following table summarizes these categories, followed by a table showing the predicates belonging to each category.

| Rule | Head | Body |
|---|---|---|
| explicit | execution or specification atom | empty |
| Assignment | must_execute$_u$, must_execute$_r$, cannot_do$_u$ or cannot_do$_r$ atom | specification, execution, or comparison literals, or aggregate atoms |
| static checking | statically_checked atom | specification, or comparison literals, or aggregate atoms. Each literal in an aggregate atom is a specification or a comparison literal |
| Integrity | panic atom | specification, execution, or comparison literals, or aggregate atoms |
| Static | planning or specification atom | specification or comparison literals, or aggregate atoms. Each literal in an aggregate atom is a specification or comparison literal |
| Dynamic | planing, specification or execution atom | specification execution, or comparison literals, or aggregate atoms. At least a literal in the rule must be an execution literal |

| Specification Predicates | |
|---|---|
| **Predicate** | **Meaning** |
| role (*R*, *T*) | States that the role *R* is authorized to execute the task *T*. |
| user (*U*, *T*) | States that the user *U* is authorized to execute the task *T*. |
| Belong (*U*, *R*) | States that the user *U* is a member of the role *R*. |
| $R_i \succ R_j$ | States that role $R_i$ dominates $R_j$ in the global role order. |

| Execution Predicates | |
|---|---|
| **Predicate** | **Meaning** |
| Execute$_u$ (*U*, *T*, *n*) | States that the user *U* executed the *n-th* activation of task *T*. |
| Execute$_r$ (*R*, *T*, *n*) | States that the role *R* executed the *n-th* activation of task *T*. |
| Abort (*T*, *n*) | States that the *n-th* activation of task *T* was aborted. |
| Success (*T*, *n*) | States that the *n-th* activation of task *T* was successfully executed. |

| Planning Predicates | |
|---|---|
| **Predicate** | **Meaning** |
| Cannot_do$_u$ (*U*, *T*) | Sates that the user *U* is among the set of users that are not allowed to execute task *T*. |
| Cannot_do$_r$ (*R*, *T*) | States that the role *R* is among the set of roles that are not allowed to execute task *T*. |
| Must_execute$_u$ (*U*, *T*) | States that the user *U* is among the set of users that must execute task *T*. |
| Must_execute$_r$ (*R*, *T*) | States that the role *R* is among the set of roles |
| Panic | States that there exits a workflow constraint which is not satisfiable. |

| Aggregate Predicate | |
|---|---|
| **Predicate** | **Meaning** |
| Count (*W*, *n)* | Counts the number of different answers to of the query *W* and returns this value as *n*. |
| avg (*x*, *W*, *n)* | Computes the average of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |
| min (*x*, *W*, *n)* | Computes the minimum of the values of variable *x* obtained from all the different answers to the query *w*, and returns this value as *n*. |
| max (*x*, *W*, *n)* | Computes the maximum of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |
| sum (*x*, *W*, *n)* | Computes the sum of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |

## Rules

**Explicit Rule**
Explicit rules are fact as their bodies are always empty. An explicit rule may have either a specification or execution atom as the head.

Explicit rules whose head is a specification atom express the workflow role specification, such as the roles assigned to each task and the local role order. These rules are automatically generated from the workflow role specification.

Explicit rules whose head is an execution atom are generated either by the system during workflow execution, or by the planner to generate the role (user) assignments.

**Assignment Rule**
An assignment rule must have a `must_execute`$_u$, `must_execute`$_r$, `cannot_do`$_u$ or a `cannot_do`$_r$ atom as the head and a specification, execution, or comparison literal, or an aggregate atom as the body.

An assignment rule expresses the restrictions imposed by the workflow constraints on the set of roles/users that can execute a given task. Intuitively, assignment rules having a `must_execute`$_u$ or `must_execute`$_r$ as a head state that a role/user must execute a given task in order to ensure constraint consistency Assignment rules having a `cannot_do`$_u$ or a `cannot_do`$_r$ as a head state that a role/user must be prevented from executing a given task.

**Static Checking Rule**
A static checking rule has `statically_checked(`$C_i$`)` as the head where $C_i$ refers to a constraint. The body may be is either a specification or comparison literal, or an aggregate atom. Each literal appearing in an aggregate atom of a static checking rule must be either a specification or comparison literal.

A static checking rule states that the satisifability of a workflow constraint can be verified without executing the workflow. These rules are automatically generated by the system to avoid redundant checks at execution time.

**Integrity Rule**
An integrity rule has `panic` as the head and either a specification, execution, or comparison literal, or aggregate atom. Integrity rules are used to model the non-satisfiability of a given constraint.

**Static Rule**
A static rule is either an explicit, assignment, static checking or integrity rule such that all literals appearing in the rule body are either specification, aggregate or comparison literals. Each literal appearing in an aggregate atom of a static rule must be either a specification or comparison literal.

Intuitively, static rules are those that can be evaluated without executing the workflow. They are used to encode that static constraints associated with the workflow.

**Dynamic Rule**
A dynamic rule is either an explicit, assignment, or integrity rule containing at least an execution literal. The execution literal can appear either explicitly or as argument of an aggregate literal. Dynamic rules must be evaluated during the workflow execution. These rules are used to encode dynamic and hybrid constraints.

**Example**

```
panic ← count(role(Ri,Tj),n), n<3
cannot_dor(Ri,T2) ← executer(Rj,T1,k), Rj ≥ Ri, Ri ≠ General Manager;
cannot_dor(Ri,T2) ← executer((Rj,T1,k), Rj ≻ Ri,Ri ≻ Rj;
cannot_dor(Ri,T4) ← execute(Rj,T2,k), Ri ≥ Rj, Ri ≠ General Manager;
cannot_dor(Ri,T4) ← executer(Rj,T2,k), not Rj ≻ Ri, not Ri ≻ Rj;
cannot_dou(Ui,T4) ← belong(Ui, Refund Clerk), executeu(Ui,T1,k);
cannot_dou(Ui,T3) ← executeu(Ui,T2,k)
cannot_dou(Ui,T2) ← executeu(Ui,T2,k);
cannot_dou(Uj,T1) ← count(abort(T1,k), executeu(Uj,T1,k),n), n>4;
cannot_dou(Bob,T4) ← executeu(Bob,T2,k).
```

**Diagram 3: Example Constraints encoded by Normal Logic Program**

## Constraint Base

A constraint base (CB) contains the set of rules encoding the constraint defined on the workflow. The CB consists of a set of explicit, assignment and integrity rules. The rules are specified by either the Workflow Definition Language provided by the system security officer or automatically generated by the system to reflect the system's current state.

## ***Design Details***

The functionality provided by the prototype system is contained within a dynamic link library (DLL) so that external applications can utilize the functionality provided. By placing the system within a dynamic link library it not only allows other applications to utilize the functionality but also resembles an API that could be incorporated into future operating systems.

The system security officer is responsible for defining the workflow (Workflow Role Specification) and the constraints using the high language designed. This language, named the **Workflow Definition Language**, is presented and explained in detail throughout the following sections. The high level language is a context free grammar derived and extended from the normal logic program proposed by Bertino et al [3].

The Workflow Definition Language will be loaded into the DLL via a function. This function will initialize and populate internal data structures accordingly. The DLL provides a method to execute the loaded workflow. This method simply executes the workflow by providing a default interface based on the Workflow Role Specification.

It is envisaged that the DLL will provide alternative methods that allow customizing the interfaces and executing the workflow under programmer control. This approach would be would be required if a well-defined API is to be produced to export the functionality provided by this project.

The following important components have been identified and explained in the following sections:

- Workflow Definition Language
- Constraint Base
- Constraint Analysis and Enforcement Module
  - Static Analysis Phase
  - Pruning Phase
  - Planner Phase
  - Runtime Phase

## Workflow Definition Language Details

The Workflow Definition Language is specified using a context free grammar. The language requires that the Workflow Role Specification be defined first, followed by the constraints on role and user assignment to tasks in the defined workflow.

The language is partitioned into the following two distinct sets of statements:

- Workflow Role Specification;
- Constraint Base.

The Workflow Role Specification defines the workflow. The Workflow Role Specification is needed by the system to define each task and the associated roles and processing entity. The Constraint Base specifies the constraints on user and role assignments throughout the lifetime of the workflow.

When the program defined by the Workflow Definition Language is loaded the parser will check that the program is syntactically correct. If the program contains error the workflow cannot be executed and an appropriate error will be returned.

As the parser is checking the syntax it also dynamically constructs the Symbol Table and Constraint Base

As described in the Analysis Section, Bertino et al [3] proposed a normal logic program for defining constraints. This normal logic program has been transformed into a high level program (Workflow Definition Language). Each predicate in the proposed normal logic program has been converted to a corresponding statement that may be used in the Workflow Definition Language. Diagram 4 shows the mapping from each predicate to the corresponding statement.

| Specification Predicates | | |
|---|---|---|
| **Predicate** | **Statement** | **Meaning** |
| Role (*R*, *T*) | RoleCanExecuteTask (*role, task*) | States that the role *R* is authorized to execute the task *T*. |
| User (*U*, *T*) | UserCanExecuteTask (*user, task*) | States that the user *U* is authorized to execute the task *T*. |
| Belong (*U*, *R*) | UserBelongsToRole (*user, role*) | States that the user *U* is a member of the role *R*. |
| $R_i \succ R_j$ | RoleDominates (*role1, role2*) | States that role $R_i$ dominates $R_j$ in the global role order. |

| Execution Predicates | | |
|---|---|---|
| **Predicate** | **Statement** | **Meaning** |
| $Execute_u$ (*U*, *T*, *n*) | UserExecutedTask (*user, task, activation*) | States that the user *U* executed the *n-th* activation of task *T*. |
| $Execute_r$ (*R*, *T*, *n*) | RoleExecutedTask (*role, task, activation*) | States that the role *R* executed the *n-th* activation of task *T*. |
| Abort (*T*, *n*) | TaskAborted (*task, activation*) | States that the *n-th* activation of task *T* was aborted. |
| Success (*T*, *n*) | TaskSucceeded (*task, activation*) | States that the *n-th* activation of task *T* was successfully executed. |

| Planning Predicates | | |
|---|---|---|
| **Predicate** | **Statement** | **Meaning** |
| $Cannot\_do_u$ (*U*, *T*) | UserCannotExecuteTask (*user, task*) | Sates that the user *U* is among the set of users that are not allowed to execute task *T*. |
| $Cannot\_do_r$ (*R*, *T*) | RoleCannotExecuteTask (*role, task*) | States that the role *R* is among the set of roles that are not allowed to execute task *T*. |
| $Must\_execute_u$ (*U*, *T*) | UserMustExecuteTask (*user, task*) | States that the user *U* is among the set of users that must execute task *T*. |
| $Must\_execute_r$ (*R*, *T*) | RoleMustExecuteTask (*role, task*) | States that the role *R* is among the set of roles |
| Panic | Panic | States that there exits a workflow constraint which is not satisfiable. |

| Aggregate Predicate | | |
|---|---|---|
| **Predicate** | **Statement** | **Meaning** |
| Count (*W*, *n)* | Count (*statement*) | Counts the number of different answers to of the query *W* and returns this value as *n*. |
| Avg (*x*, *W*, *n)* | Avg (*statement*) | Computes the average of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |
| Min (*x*, *W*, *n)* | Min (*statement*) | Computes the minimum of the values of variable *x* obtained from all the different answers to the query *w*, and returns this value as *n*. |
| Max (*x*, *W*, *n)* | Max (*statement*) | Computes the maximum of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |
| Sum (*x*, *W*, *n)* | Sum (*statement*) | Computes the sum of the values of variable *x* obtained from all the different answers of the query *W*, and returns this value as *n*. |

**Diagram 4: Mapping from Predicate to Statement**

## Variable Parameters

The following variables may be used as parameters in a Workflow Definition Language statement:

- SYS_USER1, SYS_USER2
  Indicates that any user may be used in order to satisfy the statement.
- SYS_ROLE1, SYS_ROLE2
  Indicates that any role may be used in order to satisfy the statement.
- SYS_TASK1, SYS_TASK2
  Indicates that any task may be used in order to satisfy the statement.

**Example**
The following dynamic rules illustrate the use of variable parameters in statements.

```
/*-------------------------------------------------*/
/* A role that executed Task1 cannot execute Task2 */
/* (dynamic separation of duties)                  */
/*-------------------------------------------------*/
CONSTRAINT 1
  IF RoleExecutedTask (SYS_ROLE1, Task1, 0) THEN
           RoleCannotExecuteTask(SYS_ROLE1, Task2);
  END;

/*-------------------------------------------------*/
/* If a user belonging to the role Clerk executed  */
/* Task1, then that user cannot execute Task4      */
/*-------------------------------------------------*/
CONSTRAINT 2
  IF (UserBelongs(SYS_USER1, “Clerk”) AND
       UserExecutedTask(SYS_USER1, Task1)) THEN
           UserCannotExecute(SYS_USER1, Task4);
  END;
```

The rule specified by CONSTRAINT 1 uses SYS_ROLE1 to state that any role that has executed Task1 may not execute Task2.

The rule specified by CONSTRAINT 2 uses SYS_USER1 to state that any user that belongs to Clerk role that has executed Task1 is not allowed to execute Task4.

## Context Free Grammar

The Workflow Definition Language is defined by a context free grammar, which is defined in Diagram 5. Diagram 6 is an example of an actual Workflow Definition Program using this language.

With respect to diagram 5, the following should be noted:

- The context free grammar does not include productions for explicit rules as these are generated by system from the workflow role specification.
- The context free grammar does not include productions for static checking rules as these are generated by the system.
- The user of the high-level language defined by the context free grammar only needs to specify assignment and integrity rules.
- Assignment and integrity rules can have specification, execution , or comparison literal, or an aggregate atom as their body.
- The set of explicit rules are automatically generated by the system. The set of explicit rules is generated from:
  - The Workflow Role Specification and are determined according to the conditions specified in Diagram 7;
  - The current system security setup (managed by *RBACManager*).

```
Workflow            =   WORKFLOW ROLE SPECIFICATION
                            WorkflowRoleSquence
                        CONSTRAINT BASE
                            ContraintSeq
                        END ; .

WorkflowRoleSeq     =   WorkflowRoleSpeq {WorkflowRoleSpeq} .
WorkflowRoleSpec    =   TASK ident
                            TASK NAME:      "StringLiteral"
                            GLOBAL ROLES:   "StringLiteral"
                            LOCAL ROLES:    "StringLiteral"
                            ACTIVATION:     int
                            PROGRAM:        "StringLiteral"

ConstraintSeq       =   Constraint {Constraint} . Constraint
Constraint          =   CONSTRAINT ident Rule {Rule}
Rule                =   IF Condition THEN StatementSequence END ;
Condition           =   RuleBody { ( AND | OR ) RuleBody }
RuleBody            =   [not]
                        (
                            (specificationLiteral | executionLiteral)           |
                            "(" Condition ")"                                   |
                            AggregateLiteral "(" Condition ")"                  |
                            SYS_ROLE1, SYS_ROLE2 comparisonLiteral roleIdent    |
                            SYS_USER1, SYS_USER2 comparisonLiteral userIdent    |
                            SYS_TASK1, SYS_TASK2 comparisonLiteral taskIdent
                        )
StatementSequence   =   StatementSequence {StatementSequence}  .
Statement           =   ExecutionLiteral ";" .


SpecificationLiteral =  "CanUserExecuteTask(" UT "," TT "," NT ")" |  "CanRoleExecuteTask(" RT "," TT "," NT ")" |
                        "UserBelongs" | "glb" | "lub" | "RoleDominates"
ExecutionLiteral    =   "UserExecutedTask | "RoleExecutedTask" |
                        "TaskAborted(" TT "," NT ")" | "TaskSucceeded(" TT "," NT ")"
ComparisonLiteral   =   "=" | "#" | "<" | ">" | "<=" | ">" | ">="
AggregateLiteral    =   Count, Sum, Avg, Min, Max
```

**Diagram 5: Workflow Definition Language Context Free Grammar**

**Context Free Grammar Example**

```
WORKFLOW ROLE SPECIFICATION

    TASK Task1
        TASK NAME: "Prepare Cheque"
        GLOBAL ROLES:  "Refund Clerk"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:   "..\..\App1\Debug\App1.exe"

    TASK Task2
        TASK NAME: "Approve Cheque"
        GLOBAL ROLES:  "Refund Manager", "General Manager"
        LOCAL ROLES:
        ACTIVATIONS:   2
        PROGRAM:   "..\..\App2\Debug\App2.exe"

    TASK Task3
        TASK NAME: "Summarize Decision"
        GLOBAL ROLES:  "Refund Manager", "General Manager"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:   "..\..\App3\Debug\App3.exe"

    TASK Task4
        TASK NAME: "Issue/Void Check"
        GLOBAL ROLES:  "Refund Clerk"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:   "..\..\App4\Debug\App4.exe"

CONSTRAINT BASE

    /*-----------------------------------------------------------*/
    /* At least 3 roles must be associated with the workflow. */
    /*-----------------------------------------------------------*/
    CONSTRAINT Constraint1
        IF Count(RoleCanExecuteTask(SYS_ROLE1, SYS_TASK1)) < 3 THEN
            Panic;
        END;

    /*-------------------------------------------------------------*/
    /* Task4 must be executed by a role dominating the role that */
    /* executed Task 1, unless executed by the General Manager   */
    /*-------------------------------------------------------------*/
    CONSTRAINT Constraint2
        IF RoleExecutedTask (SYS_ROLE2, Task1, 0) AND
           (RoleDominates (SYS_ROLE2, SYS_ROLE1) OR
               (SYS_ROLE2 = SYS_ROLE1 AND SYS_ROLE1 # "General Manager")) THEN
            RoleCannotExecuteTask(SYS_ROLE1, Task4);
        END;

    /*-----------------------------------------------------------------*/
    /* If a user belongs to General Manager and performed Task1, then */
    /* they cannot perform Task4.                                      */
    /*-----------------------------------------------------------------*/
    CONSTRAINT Constraint3
        IF UserBelongsToRole(SYS_USER1, "General Manager") AND
           UserExecutedTask(SYS_USER1, Task1, 0) THEN
            UserCannotExecuteTask(SYS_USER1, Task4);
        END;

    /*-------------------------------------------------------------*/
    /* If a person has performed Task2, they cannot perform Task3 */
    /*-------------------------------------------------------------*/
    CONSTRAINT Constraint4
        IF UserExecutedTask(SYS_USER1, Task2, 0) THEN
            UserCannotExecuteTask(SYS_USER1, Task3);
        END;

END WORKFLOW ROLE SPECIFICATION;
```

**Diagram 6: Example Workflow Definition Program**

## Constraint Base

The constraint base (CB) contains the set of rules specified by and generated from the Workflow Definition Program (diagram 6). The CB will be represented internally by the following two data structures:

- Linked List.

  The linked list will contain the set of explicit rules. The explicit rules are generated before and during workflow execution.

- Abstract Syntax Tree (AST).

  The Abstract Syntax Tree will contain the hybrid and dynamic constraints specified by the Workflow Definition Program provided. That is, it contains the assignment and integrity rules defining the constraints on user and role assignments to tasks in the workflow.

### Linked List

The linked list will contain the explicit rules of the workflow. The explicit rules are generated before and during workflow execution. Most of the explicit rules that are generated before workflow execution are based on the current RBAC setup. The set of explicit rules generated before the workflow executes are determined according to the table below:

| Rule | Condition |
|---|---|
| $Ri \succ R_j \leftarrow$ | $\forall R_i, R_j \in$ such that $R_j$ precedes $R_i$ in the global role order |
| $Role(R_i,T_j) \leftarrow$ | $\forall R_i \in R, \forall T_j \in T$ such that $R_l \in RS_j$ |
| $User(u_l,T_j) \leftarrow$ | $\forall u_i \in U, \forall T_j \in T$ such that $\exists R_k \in RS_j$ with $ui \in U_k$ |
| $Belong(u_i,R_j) \leftarrow$ | $\forall u_i \in U, \forall R_j \in R$ such that $u_l \in U_j$ |
| $Glb(R_j,T_i) \leftarrow$ | $\forall R_j \in R, \forall T_i \in T$ such that $R_j = glb(RSi)$ |
| $Lub(R_j,T_i) \leftarrow$ | $\forall R_j \in R, \forall T_i \in T$ such that $R_j = lub(RSi)$ |
| $Execute_r(R_i,T_j,k) \leftarrow$ | $\forall R_i \in \forall T_j \in T, \forall k \in N$ such that $R_i$ executes the k-the activation of $T_j$ |
| $Execute_u(u_i,T_j,k) \leftarrow$ | $\forall u_i \in U, \forall T_j \in T, \forall k \in N$ such that user $u_i$ executes the k-th activation of $T_j$ |
| $Abort(T_i,k) \leftarrow$ | $\forall T_i \in$ , $\forall k \in N$ such that the k-th activation of $T_i$ aborts |
| $Success(T_i,k) \leftarrow$ | $\forall T_i \in T, \forall k \in N$ such that the k-th activation of $T_i$ successfully executes |

**Diagram 7: Generated Explicit Rules**

As the workflow progresses explicit rules are inserted into the linked list to record the current state of the workflow. For example, the Run-time phase inserts the rules $execute_r(\ldots) \leftarrow$, $execute_u(\ldots) \leftarrow$, `success` $\leftarrow$ or `abort` $\leftarrow$ depending on the outcome of a task.

### Abstract Syntax Tree (AST)

The parser produces an explicit tree as a byproduct of the recognition process, and a separate tree-walking automaton then visits the nodes of the tree in whatever order is required to produce the desired final effect. An **abstract syntax tree (AST)** is an attractive way of representing strings in a language since they make explicit the structure of the string in a particular economical way.

Once the parser has successfully parsed a rule and created the appropriate rule object, this object is placed in the CB. If the rule is a static rule the CB evaluates the rule. If the rule is successfully evaluated each statement in the rule is placed in the linked list portion of the CB. If the parsed rule is hybrid or dynamic the rule is placed in the AST. The CB then dynamically creates a branch in the AST for this rule.

The Interface Description Language in Diagram 8 describes the AST for the Workflow Definition Language.

```
Structure Workflow Definition Language
Root CB is

    CB ::= statements;
    statements => as_statements : seqOf RULE ;

    RULE ::= statement | MULTISTATEMENT ;
    MULTISTATEMENT ::= and | or ;

    statement =>  lx_string                 :  STRING ;

    and       =>  as_left_statement         :  RULE ;
                  as_right_statement        :  RULE ;

    or        =>  as_left_statement         :  RULE ;
                  as_right_statement        :  RULE ;

end.
```

**Diagram 8: IDL for Workflow Definition Language**

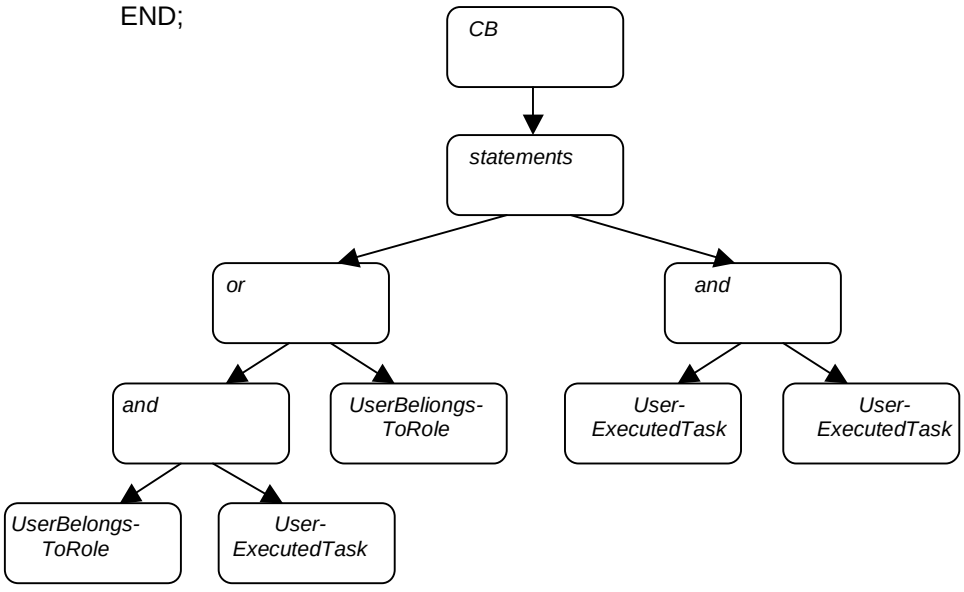
Note that this abstraction is simply implemented as a list of rules. The rules themselves are in the form of an AST. The dynamic rules in the constraint base may be evaluated by traversing the list of rules and evaluating each rule (see the General Operation/Interaction Section for details on how rules are evaluated).

**Abstract Syntax Tree Example**

The following diagram shows the AST that would be generated from the following rules:

CONSTRAINT 1
IF (UserBelongs(SYS_USER1, “Clerk”) AND
UserExecutedTask(SYS_USER1, Task1)) OR
UserBelongs(SYS_USER1, “Manager”) THEN
*statements*
END;

CONSTRAINT 2
IF UserExecutedTask (SYS_USER1, Task1, 0) AND
UserExecutedTask(SYS_USER1, Task2, 0) THEN
*statements*
END;



## General Operation/Interaction

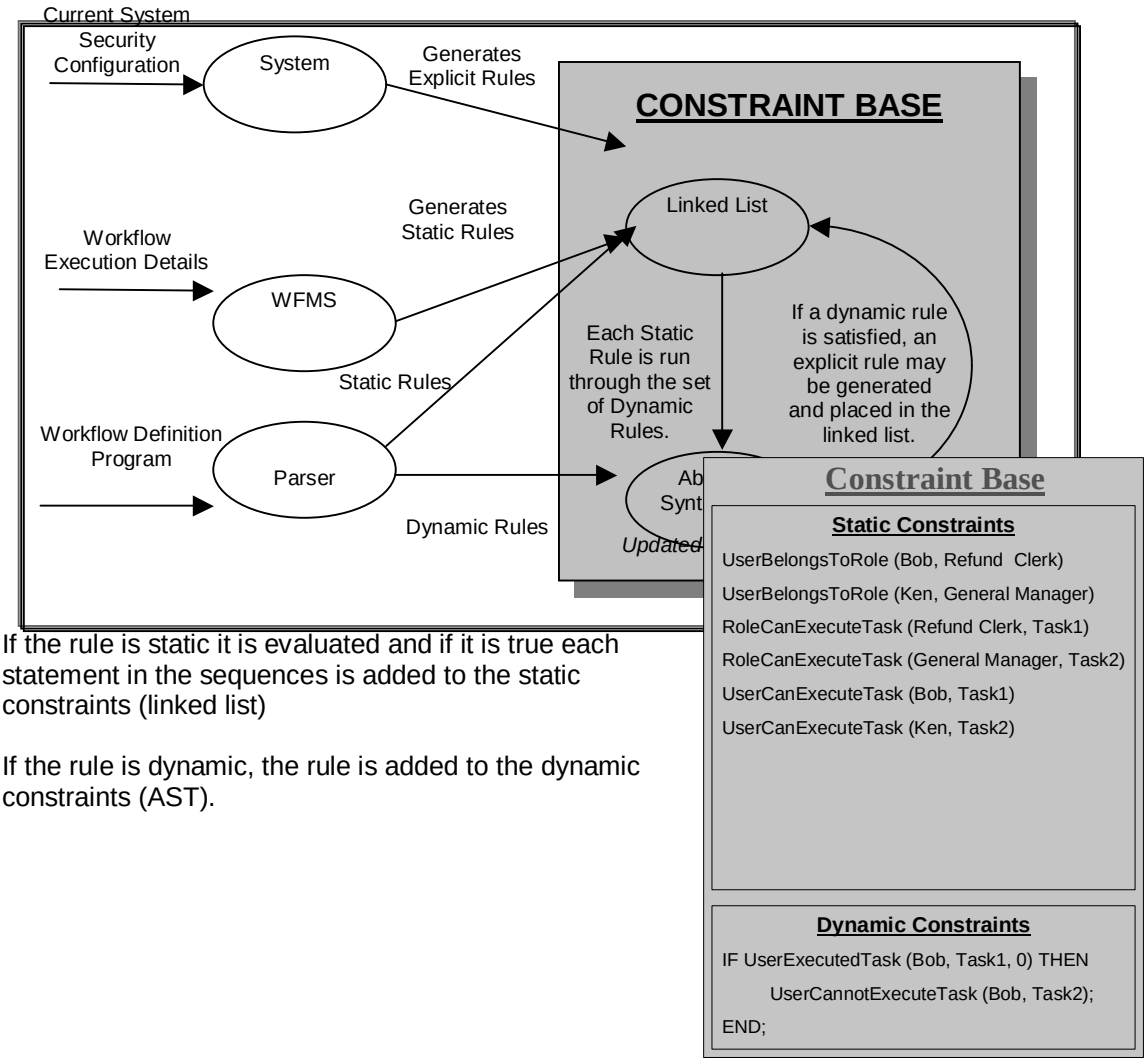
Once the AST has been built and the workflow progresses, each rule is evaluated in turn. All the static rules in the linked list will be traversed to determine if the rule may be satisfied. If a rule is satisfied after substitution the resulting static rule will be inserted into the linked list. This is illustrated in the following diagram:

**Diagram 9: General CB Operation/Interaction**

### Example

The constraint base is initialized with the static rules according to the current RBAC setup. These rules are inserted into the static constraints (linked list).

Each rule is parsed.




If the rule is static it is evaluated and if it is true each statement in the sequences is added to the static constraints (linked list)

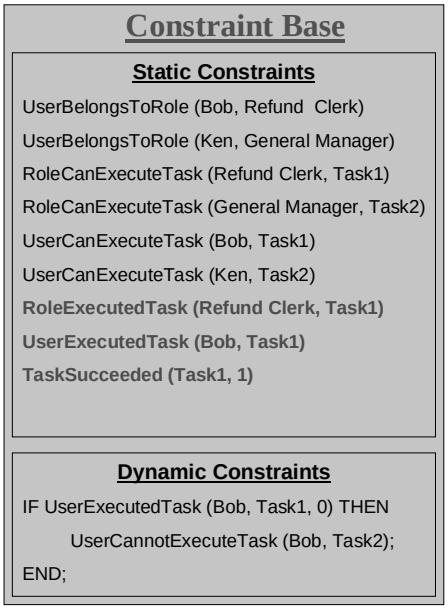
If the rule is dynamic, the rule is added to the dynamic constraints (AST).

After a task is executed 3 static rules are inserted into the constraint base to show details about the executed task.

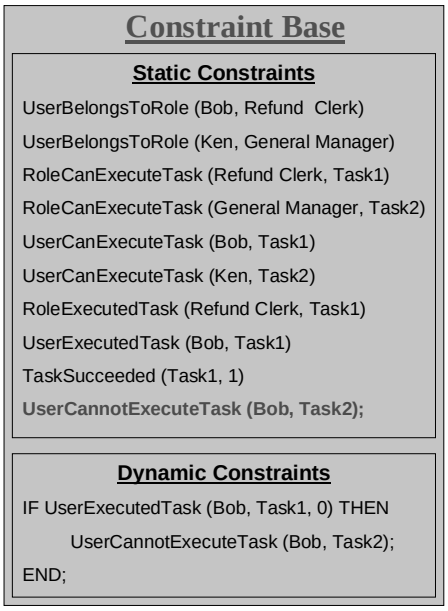
The 3 rules are:

- RoleExecutedTask
- UserExecutedTask
- TaskAborted *or* TaskSucceeded

The dynamic constraints are now evaluated. Each rule in the dynamic constraints is evaluated with all the static constraints in order to determine if the dynamic rule can be satisfied with any combination of the static rules.




If a dynamic rule is satisfied, each statement in the sequence of statements is inserted into the static constraints (linked list).




Let's have a look now at all of this in practice. Two case studies are presented. Case Study 1 illustrates a simple activity dealing with tax refunds. Case study 2 is exactly the same as Case Study 1, however, it demonstrates the use of the safety factor in order to increase efficiency.

## Case Study 1: Tax Refund 1

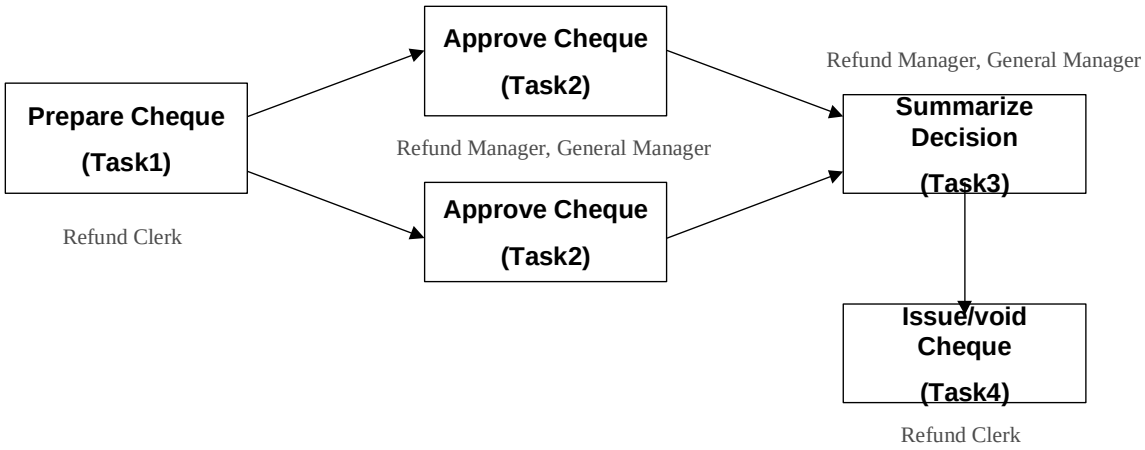
### Description

The activities dealing with a tax return can be modeled by a simple workflow consisting of four tasks.

- Task PrepareCheque: A clerk prepares a cheque for a tax refund.
- Task ApproveCheque: The cheque is forwarded to two different managers, that have the privilege to approve or disapprove it. The cheque will be issues if both the managers approve it, it will be voided, otherwise.
- Task SummarizeDecision: The decisions of the managers are collected and the final decision is made. The manager who collects the results must be different from those executing T2.

- Task IssueVoidCheque: A clerk issues or voids the cheque based on the result of task T3; the clerk issuing the cheque must be different from the clerk who prepared the check.

The tax refund workflow is graphically illustrated below:




This workflow example illustrates the use of roles and both *static* and *dynamic* separation of duties. Each task is assigned a role; namely T1 and T4 must be executed by a role 'clerk', whereas T2 and T3 by a role 'manager'. Therefore, the various duties, and the corresponding authorizations, are statically separated by imposing that different roles execute different tasks. An example of dynamic separation of duties is the constraint that a particular clerk must not execute both tasks T1 and T4 for the same cheque. However, he/she can perform task T1 on some cheques, while performing task T4 for other cheques. Therefore, a clerk cannot issue a cheque that he/she prepared.

## Workflow Role Specification

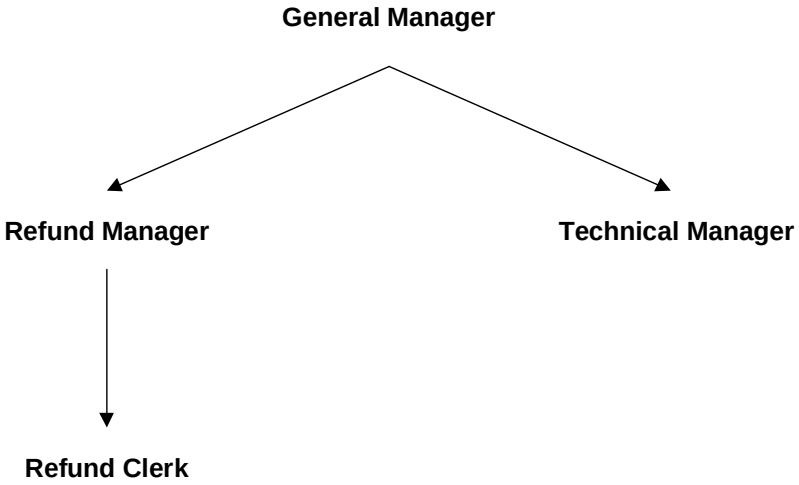
The workflow role specification for the tax refund workflow is:

```
W = {
        (PrepareCheque, ({Refund Clerk}, {}), 1),
        (ApproveCheque, ({Refund Manager, General Manager}, {}), 2),
        (SummarizeDecision, ({Refund Manager, General Manager}, {}), 1),
        (IssueVoidCheque, ({Refund Clerk}, {}), 1)
    }
```

In this role order the role 'Refund Clerk' is associated with task *PrepareCheque* of our tax refund example. This means that, by default, the task of preparing a check is assigned to 'Refund Clerk'. However, if no user authorized to play role 'Refund Clerk' is available to execute by any user playing role 'Refund Manager' or role 'General Manager', since both 'Refund Manager' and 'General Manager' precede 'Refund Clerk' in the role hierarchy.

## Role Hierarchy

The diagram below illustrates the role order and role members:




General Manager = {Ken, Meg}
Refund Manager = {John, Mary, Tom}
Refund Clerk = {Bob, Sam, Matt, Alice}

## Constraints

The following constraints will be modeled in the tax refund workflow:

- Constraint1: At least three roles must be associated with the workflow.
- Constraint2: Task4 must be executed by a role dominating the role that executed Task1, unless executed by the General Manager.
- Constraint3: If a user belonging to the role 'General Manager' has executed task Task1, then they cannot perform Task4.
- Constraint4: If a user has performed Task2, then they cannot execute Task3.
- Constraint5: If 'Ken' executes Task2, then he cannot execute Task4.
- Constraint6: If an activation of Task4 aborts then Task4 must be completed by the General Manager.

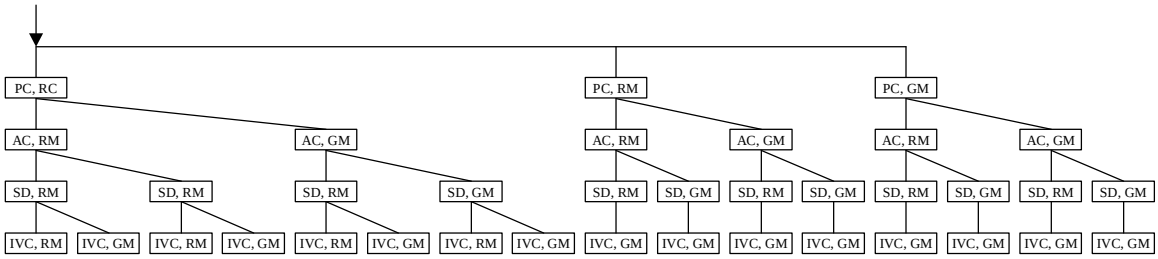
## Workflow Definition Language

The Workflow Definition Language Program that defines the tax refund workflow and associated constraints is shown below:

```
WORKFLOW ROLE SPECIFICATION
    TASK PrepareCheque
        TASK NAME: "Prepare Cheque"
        GLOBAL ROLES:  "Refund Clerk"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:    "..\..\App1\Debug\App1.exe"

    TASK ApproveCheque
        TASK NAME: "Approve Cheque"
        GLOBAL ROLES:  "Refund Manager", "General Manager"
```

```
        LOCAL ROLES:
        ACTIVATIONS:   2
        PROGRAM:   "..\..\App2\Debug\App2.exe"

    TASK SummarizeDecision
        TASK NAME: "Summarize Decision"
        GLOBAL ROLES:  "Refund Manager", "General Manager"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:   "..\..\App3\Debug\App3.exe"

    TASK IssueVoidCheque
        TASK NAME: "Issue/Void Check"
        GLOBAL ROLES:  "Refund Clerk"
        LOCAL ROLES:
        ACTIVATIONS:   1
        PROGRAM:   "..\..\App4\Debug\App4.exe"


CONSTRAINT BASE

    /*-----------------------------------------------------------*/
    /* At least 3 roles must be associated with the workflow. */
    /*-----------------------------------------------------------*/
    CONSTRAINT Constraint1
        IF Count(RoleCanExecuteTask(SYS_ROLE1, SYS_TASK1)) < 3 THEN
            Panic;
        END;

    /*-------------------------------------------------------------*/
    /* Task4 must be executed by a role dominating the role that */
    /* executed Task 1, unless executed by the General Manager   */
    /*-------------------------------------------------------------*/
    CONSTRAINT Constraint2
        IF RoleExecutedTask (SYS_ROLE2, PrepareCheque, 0) AND
           (RoleDominates (SYS_ROLE2, SYS_ROLE1) OR
               (SYS_ROLE2 = SYS_ROLE1 AND SYS_ROLE1 # "General Manager")) THEN
            RoleCannotExecuteTask(SYS_ROLE1, IssueVoidCheque);
        END;

    /*-----------------------------------------------------------------*/
    /* If a user belongs to General Manager and performed Task1, then */
    /* they cannot perform Task4.                                      */
    /*-----------------------------------------------------------------*/
    CONSTRAINT Constraint3
        IF UserBelongsToRole(SYS_USER1, "General Manager") AND
           UserExecutedTask(SYS_USER1, PrepareCheque, 0) THEN
            UserCannotExecuteTask(SYS_USER1, IssueVoidCheque);
        END;

    /*-----------------------------------------------------------------*/
    /* If a person has performed Task2, they they cannot perform Task3 */
    /*-----------------------------------------------------------------*/
    CONSTRAINT Constraint4
        IF UserExecutedTask(SYS_USER1, ApproveCheque, 0) THEN
            UserCannotExecuteTask(SYS_USER1, SummarizeDecision);
        END;

    /*-------------------------------------------------------*/
    /* If Ken executed Task1, then he cannot execute Task4 */
    /*-------------------------------------------------------*/
    CONSTRAINT Constraint5
        IF UserExecutedTask(Ken, PrepareCheque, 0) THEN
            UserCannotExecuteTask(Ken, IssueVoidCheque);
        END;

    /*---------------------------------------------------------------*/
    /* If Task4 aborts, then the General Manager must complete Task4 */
    /*---------------------------------------------------------------*/
    CONSTRAINT Constraint6
        IF TaskAborted(IssueVoidCheque, 0) THEN
            RoleMustExecuteTask("General Manager", IssueVoidCheque);
        END;


END WORKFLOW ROLE SPECIFICATION;
```

## Role Assignment Graph




PC = PrepareCheque

AC = ApproveCheque

SD = SummarizeDecision

IVC = IssueVoidCheque

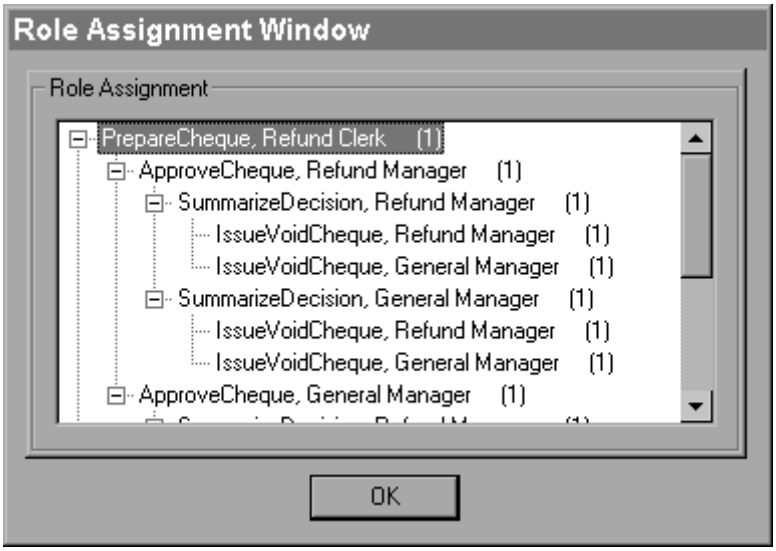
The Role Assignment Graph (RAG) contains all possible assignments of roles to tasks so that all the role related constraints associated with the workflow are satisfied. The above diagram shows the RAG generated from the workflow defined by the Workflow Definition Language presented in the previous section.

Each path in the RAG represents a role plan. A role plan is derived by selecting a path that contains one and only one node for each task in the workflow. In the above RAG, 16 possible role plans exist. The following table shows the role assignments for each task of the 16 possible paths illustrated by the above RAG.

| | | Tasks | | | |
|---|---|---|---|---|---|
| | | **PrepareCheque** | **ApproveCheque** | **SummarizeDecision** | **IssueVoidCheque** |
| **Role Paths** | **1** | Refund Clerk | Refund Manager | Refund Manager | Refund Manager |
| | **2** | Refund Clerk | Refund Manager | Refund Manager | General Manager |
| | **3** | Refund Clerk | Refund Manager | General Manager | Refund Manager |
| | **4** | Refund Clerk | Refund Manager | General Manager | General Manager |
| | **5** | Refund Clerk | General Manager | Refund Manager | Refund Manager |
| | **6** | Refund Clerk | General Manager | Refund Manager | General Manager |
| | **7** | Refund Clerk | General Manager | General Manager | Refund Manager |
| | **8** | Refund Clerk | General Manager | General Manager | General Manager |
| | **9** | Refund Manager | Refund Manager | Refund Manager | General Manager |
| | **10** | Refund Manager | Refund Manager | General Manager | General Manager |
| | **11** | Refund Manager | General Manager | Refund Manager | General Manager |
| | **12** | Refund Manager | General Manager | General Manager | General Manager |
| | **13** | General Manager | Refund Manager | Refund Manager | General Manager |
| | **14** | General Manager | Refund Manager | General Manager | General Manager |
| | **15** | General Manager | General Manager | Refund Manager | General Manager |
| | **16** | General Manager | General Manager | General Manager | General Manager |

Examination of the generated RAG shows that all role related constraints are satisfied. In particular, constraint 2 states that the *IssuseVoidCheque* task must be executed by a role dominating the role that executed the *PrepareCheque* task, unless executed by a General Manager. The above table illustrating the generated RAG does not violate this constraints.

The following screen shows the Role Assignment debugging window provided by the developed prototype.




## User Assignment Graph

The User-Role Assignment Graph (URAG) contains all possible assignments of users to tasks so that all the constraints associated with workflow are satisfied. A role plan in a generated RAG will likely result in several user plans. From the RAG, we therefore construct the User-Role Assignment Graph (URAG), which complements the RAG with information about the user plans.

Similarly to RAGs, all the plans deliverable from URAG are obtained by selecting all paths of length equal to the number of tasks in the workflow and containing one and only one node for each workflow task.

There are 1232 possible user paths. Therefore, the following table only shows the corresponding user paths for the first role path for user *Bob*.

| User Paths | Tasks | | | |
|---|---|---|---|---|
| | PrepareCheque | ApproveCheque | SummarizeDecision | IssueVoidCheque |
| 1 | Refund Clerk(Bob) | Refund Manager(John,John) | Refund Manager(Mary) | Refund Manager(John) |
| 2 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | Refund Manager(Mary) |
| 4 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | Refund Manager(Tom) |
| 5 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | General Manager(Meg) |
| 6 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | General Manager(Ken) |
| 7 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Tom) | Refund Manager(John) |
| 8 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Tom) | Refund Manager(Mary) |
| 9 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Tom) | Refund Manager(Tom) |
| 10 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Tom) | General Manager(Meg) |
| 11 | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Tom) | General Manager(Ken) |
| 12 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Meg) | Refund Manager(John) |
| 13 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Meg) | Refund Manager(Mary) |
| 14 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Meg) | Refund Manager(Tom) |
| 15 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Meg) | General Manager(Meg) |
| 16 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Meg) | General Manager(Ken) |
| 17 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Ken) | Refund Manager(John) |
| 18 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Ken) | Refund Manager(Mary) |
| 19 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Ken) | Refund Manager(Tom) |
| 20 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Ken) | General Manager(Meg) |
| 21 | Refund Clerk (Bob) | Refund Manager(John,John) | General Manager(Ken) | General Manager(Ken) |
| 21 | Refund Clerk (Bob) | Refund Manager(John,Mary) | Refund Manager(Tom) | Refund Manager(John) |
| 22 | Refund Clerk (Bob) | Refund Manager(John,Mary) | Refund Manager(Tom) | Refund Manager(Mary) |
| 23 | Refund Clerk (Bob) | Refund Manager(John,Mary) | Refund Manager(Tom) | Refund Manager(Tom) |
| . | . | . | . | . |
| . | . | . | . | . |
| . | . | . | . | . |
| . | . | . | . | . |
| 1219 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(John) | General Manager(Meg) |
| 1220 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Mary) | General Manager(Meg) |
| 1221 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Tom) | General Manager(Meg) |
| 1222 | General Manager(Ken) | General Manager(Meg,Meg) | General Manager(Ken) | General Manager(Meg) |
| 1223 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(John) | General Manager(Meg) |
| 1224 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Mary) | General Manager(Meg) |
| 1225 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Tom) | General Manager(Meg) |
| 1226 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(John) | General Manager(Meg) |
| 1227 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Mary) | General Manager(Meg) |
| 1228 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Tom) | General Manager(Meg) |
| 1229 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(John) | General Manager(Meg) |
| 1230 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Mary) | General Manager(Meg) |
| 1231 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Tom) | General Manager(Meg) |
| 1232 | General Manager(Ken) | General Manager(Ken,Ken) | General Manager(Meg) | General Manager(Meg) |

Examination of the generated user paths shows that all constraint defined by the workflow are satisfied. For example, constraint 4 states that is a user has approved a cheque (*ApproveCheque* task) then they cannot summarize the decision (*SummarizeDecision* task). Examination of user paths 1 through 21 illustrate that these constraints are enforced. In particular, user paths 1 through 21 show a possible scenario in which John executes the *ApproveCheque* task, but is forbidden from executing the *SummarizeDecision* task.

Another example is illustrated by constraint 3 which states if a user belongs to General Manager and performed *PrepareCheque* then they cannot perform *IssueVoidCheque*. User paths 1219 through 1232 enforce this constraint by stating that Meg must execute *IssueVoidCheque* task because Ken executed *PrepareCheque*.

## Workflow Execution Example 1

**Scenario:** Bob playing role Refund Clerk executes the *IssueVoidCheque* task.

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the URAG contains a vertex indicating that Bob playing role Refund Clerk may execute the *IssueVoidCheque* task.
3. The Constraint Analysis and Enforcement Module discovers the required vertices does exist and therefore executes the task processing entity.
4. When the task completes the Constraint Analysis and Enforcement Module updates the URAG by removing the vertices corresponding to assignments that are disabled as a result of the executed task. The following table shows the remaining paths as a result of the URAG being pruned.

**Scenario:** Alice playing role Refund Clerk attempts to execute the *ApproveCheque* task.

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the (pruned) URAG contains a vertex indicating that Alice playing role Refund Clerk may execute the *ApproveCheque* task.
3. The Constraint Analysis and Enforcement Module discovers that no such vertex exist and therefore forbids Alice from executing the requested task.


**Scenario:** John playing role Refund Manager executes the *ApproveCheque* task.

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the URAG contains a vertex indicating that John playing role Refund Manager may execute the *ApproveCheque* task.
3. The Constraint Analysis and Enforcement Module discovers the required vertices does exist and therefore executes the task processing entity.


4. When the task completes the Constraint Analysis and Enforcement Module updates the URAG by removing the vertices corresponding to assignments that are disabled as a result of the executed task. The following table shows the remaining paths as a result of the URAG being pruned.

**Scenario:** John playing role Refund Manager attempts to execute the *SummarizeDecision* task.

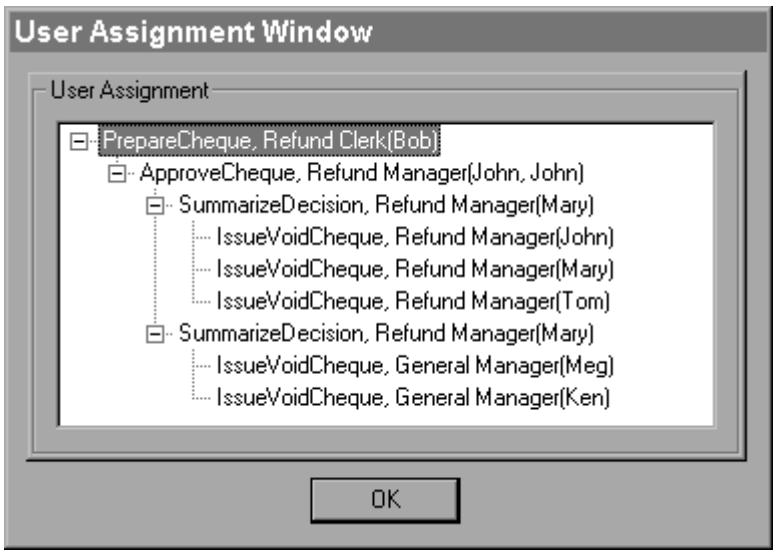
**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the (pruned) URAG contains a vertex indicating that John playing role Refund Manager may execute the *SummarizeDecision* task.
3. The Constraint Analysis and Enforcement Module discovers that no such vertex exist and therefore forbids John from executing the requested task (**Constraint 4**).


**Scenario:** Mary playing role Refund Manager executes the *SummarizeDecision* task.

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the URAG contains a vertex indicating that Mary playing role Refund Manager may execute the *SummarizeDecision* task.
3. The Constraint Analysis and Enforcement Module discovers the required vertices does exist and therefore executes the task processing entity.
4. When the task completes the Constraint Analysis and Enforcement Module updates the URAG by removing the vertices corresponding to assignments that are disabled as a result of the executed task. The following table shows the remaining paths as a result of the URAG being pruned. The following screen shows the resulting URAG.



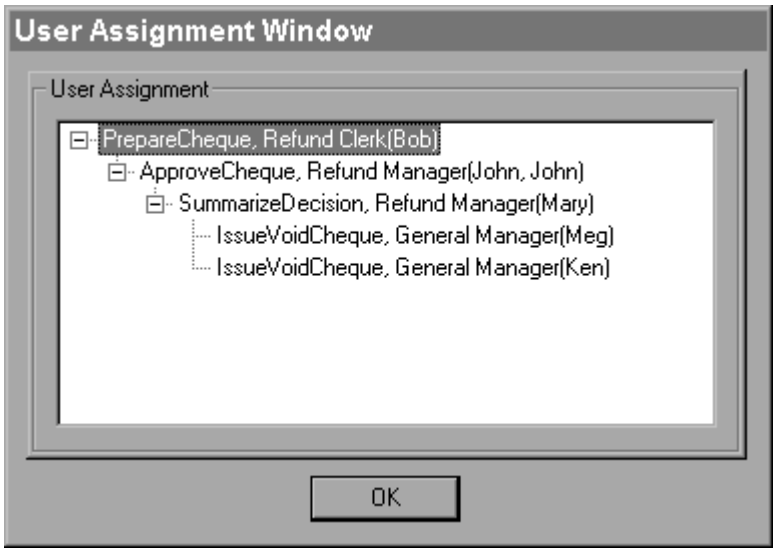

**Scenario:** Tom playing role Refund Manager executes and aborts the *IssueVoidCheque* task (Constraint6).

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the URAG contains a vertex indicating that Tom playing role Refund Manager may execute the *IssueVoidCheque* task.
3. The Constraint Analysis and Enforcement Module discovers the required vertices does exist and therefore executes the task processing entity.

4. Tom aborts the task. The planner assigns role/users to tasks under the assumption that all the tasks in the workflow successfully execute. If a task aborts, constraints involving abort predicates may no longer be satisfiable by the current plan. Therefore, the planner is invoked again to determine whether the current plan needs to be modified. The following table shows the updated paths as a result of the URAG being re-evaluated. The table shows that the *IssueVoidCheque* must now be executed by the role General Manager.

| | | Tasks | | | |
|---|---|---|---|---|---|
| | | **PrepareCheque** | **ApproveCheque** | **SummarizeDecision** | **IssueVoidCheque** |
| **User Paths** | **1** | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | General Manager(Meg) |
| | **2** | Refund Clerk (Bob) | Refund Manager(John,John) | Refund Manager(Mary) | General Manager(Ken) |




## Workflow Execution Example 2

**Scenario:** Ken playing role General Manager executes the *PrepareCheque* task.

**Action:**

1. The WFMS request that the Constraint Analysis and Enforcement Module.
2. The Constraint Analysis and Enforcement Module checks that the URAG contains a vertex indicating that Ken playing role General Manager may execute the *PrepareCheque* task.
3. The Constraint Analysis and Enforcement Module discovers the required vertices does exist and therefore executes the task processing entity.
4. When the task completes the Constraint Analysis and Enforcement Module updates the URAG by removing the vertices corresponding to assignments that are disabled as a result of the executed task. The following table shows the remaining paths as a result of the URAG being pruned.

| User Paths | Tasks | | | |
|---|---|---|---|---|
| | PrepareCheque | ApproveCheque | SummarizeDecision | IssueVoidCheque |
| 1219 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(John) | GeneralManager(Meg) |
| 1220 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Mary) | GeneralManager(Meg) |
| 1221 | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Tom) | GeneralManager(Meg) |
| 1222 | General Manager(Ken) | General Manager(Meg,Meg) | General Manager(Ken) | GeneralManager(Meg) |
| 1223 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(John) | GeneralManager(Meg) |
| 1224 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Mary) | GeneralManager(Meg) |
| 1225 | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Tom) | GeneralManager(Meg) |
| 1226 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(John) | GeneralManager(Meg) |
| 1227 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Mary) | GeneralManager(Meg) |
| 1228 | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Tom) | GeneralManager(Meg) |
| 1229 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(John) | GeneralManager(Meg) |
| 1230 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Mary) | GeneralManager(Meg) |
| 1231 | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Tom) | GeneralManager(Meg) |
| 1232 | General Manager(Ken) | General Manager(Ken,Ken) | General Manager(Meg) | GeneralManager(Meg) |

The table above illustrating the pruned URAG shows that the following constraints are satisfied by the remaining possible user paths:

- Constraint2: Task4 must be executed by a role dominating the role that executed Task1, unless executed by the General Manager.
- Constraint3: If a user belonging to the role 'General Manager' has executed task Task1, then they cannot perform Task4.

## Case Study 2: Tax Refund 2

### Description

This case study is exactly the same as Case Study 2 however, demonstrates the use of the safety factor in order to increase efficiency. The Role Hierarchy, Constraints and Workflow Definition Language remain the same.

### Current RBAC Setup

The safety factor needs to be specified for each role in the system. Therefore, the safety factor is stated in the input file that defines the current RBAC setup. The diagram below shows the input file specifying the current RBAC setup and associated safety factors.

```
USER DETAILS
USER = "Alice"
member roles: "Refund Clerk"

USER = "Bob"
member roles: "Refund Clerk"

USER = "John"
member roles: "Refund Manager"

USER = "Ken"
member roles: "General Manager"

USER = "Mary"
member roles: "Refund Manager"

USER = "Matt"
member roles: "Refund Clerk"

USER = "Meg"
member roles: "General Manager"

USER = "Sam"
member roles: "Refund Clerk"

USER = "Tom"
member roles: "Refund Manager"
```

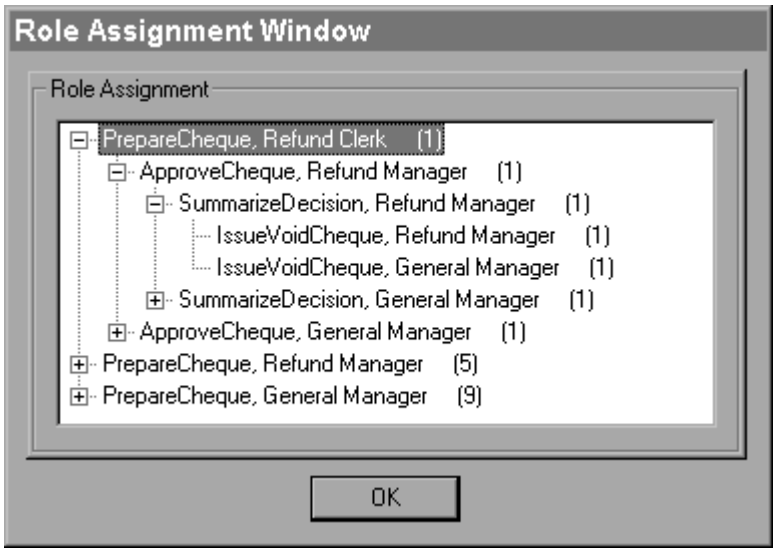
```
ROLE DETAILS
ROLE = "General Manager"
member users: "Ken" "Meg"
member roles: "Refund Manager" "Technical Manager"
parent roles:
safety factor: 1

ROLE = "Refund Clerk"
member users: "Alice" "Bob" "Matt" "Sam"
member roles:
parent roles: "Refund Manager" "General Manager"
safety factor: 1

ROLE = "Refund Manager"
Member users: "John" "Mary" "Tom"
Member roles: "Refund Clerk"
Parent roles: "General Manager"
Safety factor: 1

ROLE = "Technical Manager"
Member users:
Member roles:
Parent roles: "General Manager"
Safety factor: 0
```

## Role Assignment Graph

The Role Assignment Graph (RAG) remains exactly the same as that presented in Case Study 1.

It may not be efficient for the **Planning** phase to perform an exhaustive check of all possible users who are allowed to execute the tasks in the workflow. This is especially true in cases where roles have a large number of users authorized to play them. Therefore, for each role plan we compute the maximum number of users required, for each role $R_i$, to execute the workflow tasks according to the plan. This set is denoted as $U_{worstcase}(R_i)$. The worst case scenario arises when each task has to be executed by a different user, and moreover, each activation of a task must be executed by a different user.

The $U_{worstcase}(R_i)$ values are calculated and stored in the RAG with each vertex in the role plan. The following screen shows the calculated values (in brackets) associated with each vertex.




## User Assignment Graph

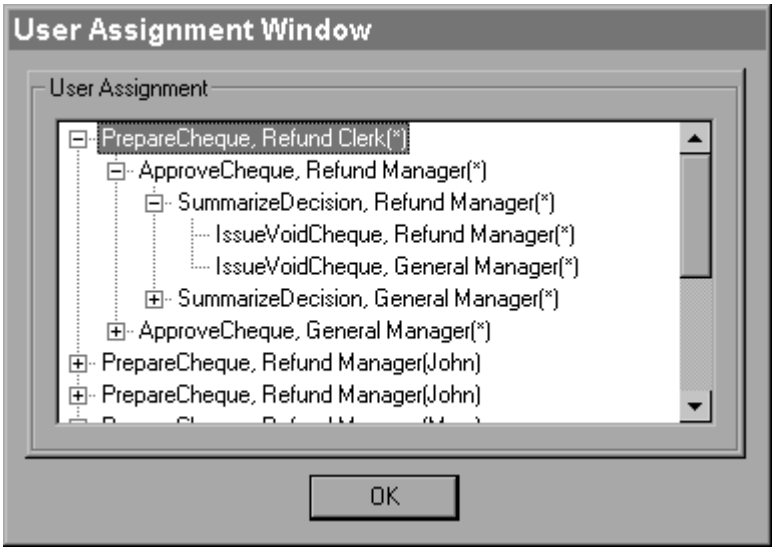
The User-Role Assignment Graph (URAG) contains all possible assignments of users to tasks so that all the constraints associated with workflow are satisfied. A role plan in a generated RAG will likely result in several user plans. From the RAG, we therefore construct the User-Role Assignment Graph (URAG), which complements the RAG with information about the user plans.

User planning is only performed for those roles that $U_{worstcase}(R_i)$ + safety_factor <= number_of_users_authorized_to_execute_ $R_i$ is not satisfied.

In this case study, the number of user plan has been reduced (and thus efficiency) from 1232 to 354. The following table only shows the corresponding user paths for the first role path. The vertices that have * as the users authorized to execute the task indicate that user planning has not been performed and that authorization will need to be dynamically checked at runtime.

| | | Tasks | | | |
|---|---|---|---|---|---|
| | | **PrepareCheque** | **ApproveCheque** | **SummarizeDecision** | **IssueVoidCheque** |
| **User Paths** | **1** | Refund Clerk(*) | Refund Manager(*) | Refund Manager(*) | Refund Manager(*) |
| | **2** | Refund Clerk(*) | Refund Manager(*) | Refund Manager(*) | General Manager(*) |
| | **3** | Refund Clerk(*) | Refund Manager(*) | General Manager(*) | Refund Manager(*) |
| | **4** | Refund Clerk(*) | Refund Manager(*) | General Manager(*) | General Manager(*) |
| | . | . | . | . | . |
| | . | . | . | . | . |
| | **341** | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(John) | GeneralManager(Meg) |
| | **342** | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Mary) | GeneralManager(Meg) |
| | **343** | General Manager(Ken) | General Manager(Meg,Meg) | Refund Manager(Tom) | GeneralManager(Meg) |
| | **344** | General Manager(Ken) | General Manager(Meg,Meg) | General Manager(Ken) | GeneralManager(Meg) |
| | **345** | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(John) | GeneralManager(Meg) |
| | **346** | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Mary) | GeneralManager(Meg) |
| | **347** | General Manager(Ken) | General Manager(Meg,Ken) | Refund Manager(Tom) | GeneralManager(Meg) |
| | **348** | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(John) | GeneralManager(Meg) |
| | **349** | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Mary) | GeneralManager(Meg) |
| | **350** | General Manager(Ken) | General Manager(Ken,Meg) | Refund Manager(Tom) | GeneralManager(Meg) |
| | **351** | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(John) | GeneralManager(Meg) |
| | **352** | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Mary) | GeneralManager(Meg) |
| | **353** | General Manager(Ken) | General Manager(Ken,Ken) | Refund Manager(Tom) | GeneralManager(Meg) |
| | **354** | General Manager(Ken) | General Manager(Ken,Ken) | General Manager(Meg) | GeneralManager(Meg) |

This screen shows the URAG corresponding to the table above.




## ***Outcome of Project***

### Difficulties Encountered

The difficulties encountered in this project may be summarised as:

- Several revisions of the rule evaluation algorithm were required before the correct functionality was obtained.

  The rule evaluation algorithm was the most difficult aspect of the designed system. Due to the difficulty several revisions were needed before a correct algorithm was devised.

- Managing large class hierarchies.

  Managing and maintaining 50 plus classes became difficult and cumbersome when revision and changes were required. This however is a research problem within itself.

- Transforming the normal logic program into a high level language.

Learning and understanding the designed normal logic program (Bertino et al [3]) required considerable time and effort.

- Role Planning and User Planning algorithms were challenging.
  The Role Planning and User Planning algorithms are highly recursive in nature and required great care and attention.

## Evaluation of Results

The project has addressed the need to distinguish the passive concept of permission assignment from the active concept of context-based activation. The prototype has provided a proof-of-concept model that successfully demonstrates that a role language provides the functionality to encode constraints on role and user assignments.

This project has addressed and proposed a method to allow current RBAC models to become active in nature. However, the need for a hybrid access control model that incorporates the advantages of broad, role-based permissions across object types, yet provides fine-grained, identity-based control on individual users in certain roles and to individual object is an important issue that still needs to be addressed.

## Problems/Deficiencies

A deficiency that may prevent the proposed solution from becoming feasible is the efficiency of the Planner Phase. The efficiency of the Planner Phase is exponential to the number of task, role and users defined in the workflow. Algorithm optimizations will only solve part of the problem. Further research needs to discover approaches to Planning that are not directly dependent on the number of tasks, roles and users in the workflow.

Another problem is that most security administrators will be unable to use or understand the Workflow Definition Language. Furthermore, for large workflows, as found in many organizations, use of the Workflow Definition Language will be become difficult to maintain and prone to errors. A better approach for defining constraints on role and user assignments is highly desirable.

## Possibilities for Further Research

The developed prototype has scope for the following additions:

- Create a well-defined API to expose the functionality.
- Provide a GUI for security administrators.
- Incorporation into distributed system.
- Refine/Extend Constraint Specification Language.
- Improve Efficiency.
- Research WFMSs and better techniques of incorporation.
- Fully integrated system.

Bertino et al [3] propose future research which includes modeling of temporal and event-based constraints, the support of more complex models of roles, along the lines discussed by Sandhu [10], and the incorporating the system into heterogeneous and distributed environments. Bertino et al [3] also plan to extensively investigate optimization techniques for role planning, based on constraint analysis. For instance, workflow constraints can be analyzed before executing the Role Planner to identify those tasks that are not involved in any constraint. To such tasks, roles can be assigned at run-time without any planning.

## Conclusion

Ferraiolo [11] states that “RBAC remains a long way from reaching its full potential as a commercially viable technology. This could only be achieved through further research and consensus on the part of researchers, vendors, and the user community”. The formation of NIST[3] and government sponsored research in the USA has been a good measure to guide the evolution of RBAC to date. As NIST has been globally accepted by the user community, it is likely that they will continue to define and guide the future evolution of RBAC.

Although there is much agreement on the basic concepts and value of RBAC, a number of issues remain that have resulted in different research and vendors proposing different models and approaches. To address these issues the area of RBAC needs to establish itself as a scientific discipline needs and produce consistent and widely used terminology and vocabulary. As RBAC is an expansive concept, the RBAC community also needs to clearly articulate what is excluded as being outside the scope of RBAC.

[3] NIST – National Institute of Standards and Technology (USA)

Although dominant literature exists that strongly suggests that RBAC is at the point of maturity and will be become predominant technology, there are concerns that currently restricts RBAC from evolving and reaching its full potential. In particular, there is a need to deal with the dynamic nature of roles and changes in authorization. The proposed research addresses this issue and endeavors to extend current research to show that RBAC is a beneficial alternative to traditional discretionary and mandatory access control methods that can address most organizations' security requirements.

This project has explored an area that has been identified as potentially preventing RBAC from reaching it's full potential. The formal model proposed by Bertino et al [3] has been investigated and shown to be a possible solution to deal with the dynamic nature of roles and changes in authorization. However, the model needs further research and attention before it becomes a feasible and acceptable approach.

## ***Bibliography***


[1] Sandhu, R., Coyne, E. J., Feinstein, H. L. & Youman, C.E. 1996, 'Role-Based Access Control Models', *IEEE Computer*, February 1996, pp. 38-47.

[2] Thomas, R. K. 1997, 'Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments', *Proceeding of the Second ACM Workshop on Role-Based Access Control*, ACM, November 1997, pp. 13-19.

[3] Bertino, E., Ferrari, E. & Atluri, V. 1997, 'A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems', *Proceeding of the Second ACM Workshop on Role-Based Access Control*, ACM, November 1997, pp. 1-12.

[4] Ferraiolo, D.F. 1996, 'An Introduction to Role-Based Access Control', *Internal Report, Computer Systems Laboratories NIST*, January 1996.

[5] Kuhn, D. R. 1997, 'Mutual Exclusion as a Means of Implementing Separation of Duty in Role-Bases Access Control System', *Proceeding of the Second ACM Workshop on Role-Based Access Control*, ACM, November 1997, pp23-40.

[6] Ferraiolo, D. F., Cugini, J. & Kuhn, D.R. 1995, 'Role Based Access Control (RBAC): Features and Motivations', *Proceedings of 11th Annual Computer Security Application Conference*, December 1995, pp. 241-48.

[7] Guiri, L. 1995b, 'Role-Based Access Control: A Natural Approach', *Proceedings of the First ACM Workshop on Role-Based Access Control*, December 1995.

[8] Barkley, J. 1995, 'Application Engineering in Health Care', *Internal Report, Computer Systems Laboratories NIST*, May 1995.

[9] Caelli, B., Rhodes, A., 'RBACManager: Implementing a Minimal Role Based Access Control Scheme (RBACM) under the Windows NT 4.0 Workstation Operating System', *Internal Report*, Information Security Research Center, QUT, December 1998.

[10] Sandhu, R. 1996b, 'Role Hierarchies and Constraints for Lattice-Based Access Control', *Computer Security – Esorics '96*, 1996, pp 65-79.

[11] Ferraiolo, D. F. 1996, 'Toward a Common Framework for Role-Based Access Control', [Online] URL http://hissa.ncsl.nist.gov/rbac/ [Accessed 9 April 1998]