Completeness, Versatility, and Practicality in Role–Based Administration

Slobodan Vukanović

svuk002@ec.auckland.ac.nz

Abstract

Applying role–based administration to role–based access control systems has gained some attention in recent literature. Scoped Administration for Role–Based Access Control (SARBAC) puts forward what is claimed to be a complete, versatile, and practical role–based administrative model. These attributes are deemed to be key for SARBAC's dynamic, flexible nature. However, SARBAC shuns alternative approaches to role–based administration. One such alternative, Graph–Based Formalism, is hastily dismissed on the basis that it is not complete, versatile, and practical. This term paper looks at Graph–Based Formalism (in particular, using administrative scope in graph–based frameworks) from the completeness, versatility, and practicality perspectives, as defined in SARBAC literature. It discusses whether it is reasonable to dismiss Graph–Based Formalism in favour of SARBAC in the light of these attributes.

1 Introduction

Completeness, versatility, and practicality are important notions in role–based access control. They are qualities one undoubtedly keeps in mind when designing role–based access control models. Completeness describes how much dynamic change a model allows. Versatility and practicality, although not as clearly defined as completeness, also play an important role in a model's success. This report looks at completeness, versatility, and practicality in SARBAC, a role–based access control model introduced in [1], and graph–based formalism, which was introduced in [3].

Our motivation is to dismiss claims that graph–based formalism is not on equal footing with SARBAC completeness–, versatility–, and practicality–wise. We show why it is not reasonable to make direct comparisons of SAR-

BAC and graph–based formalism. We then turn to administrative scope in a graph–based framework, and compare that to SARBAC. Our discussion shows that using administrative scope in graph structures is equivalent to SARBAC, in terms of completeness and versatility. We argue that practicality of administrative scope in graph–based frameworks is greater, because of the way operations in the model have been defined.

We feel that it is important to distinguish between flexibility and versatility. These are used interchangeably in some literature ([1], for example). For our purposes, flexibility measures how a role–based access control model responds to dynamic changes. We are interested in how much work one needs to do to preserve correctness of the model after a dynamic change. If this extra work is negligible, then we are more likely to call such a model flexible. It is a desirable quality – if model A is more flexible than model B, then its administration is likely to be easier, as we shall see. Versatility, on the other hand, measures the permissiveness of a model, that is, which operations will succeed and which will fail. There are no formal definitions of these terms, and we will constrain our discussion on relative flexibility and versatility of the models presented herein. That is, instead of claiming flexibility and versatility, we will examine how they are different in one model compared to another.

The report is structured as follows. Section 2 provides some background on role–based access control and administration for completeness' sake. Section 3 looks at SARBAC, while Section 4 examines graph–based formalism. The last Section concludes the report.

2 Background

2.1 Role–Based Access Control

Users of computer systems perform their tasks by accessing system's resources (data and services). *Access control* defines the ways in which and by whom these resources can be manipulated. Initial research in access control resulted in two models [2]. In *Discretionary Access Control* (DAC), access control was left to users themselves. A notion of ownership exists in DAC: it was up to the owner (a user) of a particular resource to grant or revoke access to that resource. A drawback of such a mechanism is precisely this concept of ownership. It is nearly always the case that the party who owns the system owns the resources. Individual end–users own very little on the system, if anything. The second type, *Mandatory Access Control* (MAC), defined on each resource a label representing its level of sensitivity. Only

users with a clearance level corresponding to the label of a resource were granted access to that resource. The applications of MAC outside military systems were rare [2]. The definition of ownership in DAC and narrow applicability of MAC made them unsuitable for use in domains that, like military, require control of acccess to its information.

A definition of role–based access control (RBAC) was given by Ferraiolo and Kuhn in [2] (note that this publication *does not* pioneer role–based access control). The aim was to provide a standard definition for a suitable model of access control for civilian government and commercial firms. In this model, *roles* have permissions to act on resources in some way. Users are assigned to roles; users' abilities to act upon resources are defined by roles which they are assigned to. Roles can be added, edited, or removed without a need to explicitly add or revoke permissions of individual users. They can be grouped into hierarchies, with parent roles inheriting privileges of children roles. Role hierarchies are orthogonal to authority or responsibility hierarchies found in organizations.

2.2 Administration in Role–Based Systems

The level of abstraction allowed by RBAC makes it an appealing model for administration. Administration in role–based systems defines operations that change the hierarchy. They are a vital part of an RBAC systems, as one needs ways to assign new users to roles, remove users, assign and revoke permissions, etc.

Typical administrative operations are summarized in Table 1. These are the basic operations we expect to be available in role–based systems. We have chosen these as basic operations because all of them are required to ensure correct administration, as defined by RBAC in [2]. Different models may define additional operations, and many of them do (for example adding and removing of sessions). In addition to these, some operations for changing the way administration is carried out need to be established. For example, we need ways to add or remove roles from administrators' administrative ranges. Note that, if we use RBAC to administer RBAC systems, then these operations will be variations of those in Table 1. For example, to stop a role r from being administered by some administrative role a, we may execute `RemoveEdge` with a and r as arguments.

For the remainder of this report, we will say that a role r *administers* (or *controls*) some role r' if r is able to perform at least one operation from Table 1 on r'. We will consider the terms *administers* and *controls* to refer to the same concept, so we will use them interchangeably.

Operation	Effects
`AddRole`	A role is added to the hierarchy.
`DeleteRole`	A role is removed from the hierarchy.
`AddEdge`	An edge between two roles in the hierarchy is inserted.
`RemoveEdge`	An edge between two roles in the hierarchy is deleted.
`AssignUser`	Assigns a user to a role.
`RevokeUser`	Revokes a user from a role.
`AssignPermission`	Assigns a permission to a role.
`RevokePermission`	Revokes a permission from a role.

Table 1: Sample administrative operations. Operations above the line change the hierarchy. Operations below the line deal with user–role and permission–role assignments.

3 SARBAC

3.1 Motivation

The motivation behind the development of SARBAC is to build a model which uses RBAC principles to administer RBAC systems. That is, administrative roles are a part of the hierarchy, and they use operations similar to those listed in Table 1 to administer user roles. SARBAC aims at defining a complete, versatile, and practical model, with minimal work needed to preserve the correctness when dynamic changes take place.

3.2 Administrative Scope

The notion of *administrative scope*, which serves as the basis for the SARBAC model, defines administrative responsibilities of roles in a role hierarchy. Administrative scope of a role r is a set of roles that r administers.

A role hierarchy $H = (R, \leq)$ is a partial ordering over the set R of roles. As such, it can be represented graphically by a Hasse diagram – see Figure 1(a) for an example. Let $\uparrow r = \{r' \in R : r' \geq r\}$, and $\downarrow r = \{r' \in R : r' \leq r\}$. That is, $\uparrow r$ denotes the set of all ancestors of r in H, and $\downarrow r$ denotes the set of all descendants of r in H (by definition of ancestor/descendant, r is an ancestor/descendant of itself). Administrative scope of a role r is then defined as:

$$S(r) = \{s \in R : s \leq r, \uparrow s \setminus \uparrow r \subseteq \downarrow r\}$$

Figure 1(b) is an example from [1], showing the administrative scope of role PL1.

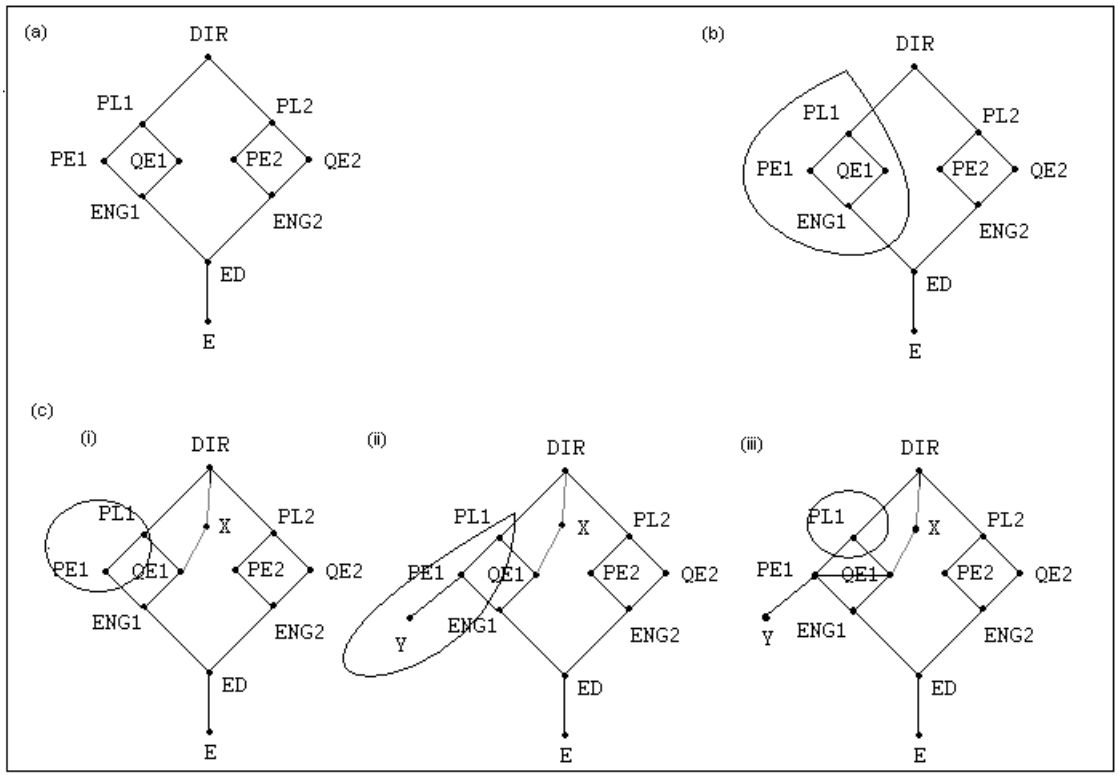

Figure 1: Administrative scope example, taken from [1]. (a) A role hierarchy. (b) Administrative scope of PL1 is circled. (c) Changes to administrative scope of PL1 as: (i) role X is added, (ii) role Y is added, and (iii) edge between PE1 and QE1 is added.

We agree with [1] that administrative scope provides a natural unit of administration. It is an intuitive and a simple enough concept to describe informally: a descendant of some role belongs in the administrative scope of that role if all paths from the descendant to the top of the hierarchy pass through that role. As the hierarchy dynamically changes, so too does administrative scope of the roles. There is no additional effort involved in checking whether violations of administrative scope exist after a hierarchy–changing operation completes. See Figure 1(c) for an example. From our earlier discussion on flexibility, we can see that a model utilizing administrative scope

has the potential to be very flexible.

3.3 RHA and SARBAC

This section briefly looks at Role Hierarchy Administration (RHA) models, and SARBAC, so that further discussion is possible. [1] defines four RHA models in increasing order of sophistication, with RHA_1 being the simplest, and RHA_4 being the most complex.

RHA_1 is simply a straight application of administrative scope on the role hierarchy. Each role in the hierarchy has its administrative scope as defined earlier. RHA_1 is not sufficiently expressive to be of much use: every role is an administrative role – it administers precisely those roles that are in its administrative scope. This model does not provide ways to circumvent this in applications in which such administration is undesirable.

RHA_2 extends RHA_1 by assigning to every role an administrative flag. The value of this flag then determines whether roles are administrators or not. Thus, RHA_2 offers finer granularity than the previous model.

RHA_3 introduces an additional binary relation termed `admin-authority`. Each pair $(a, r) \in$ `admin` $-$ `authority` defines an administrative role a, and a role r which a administers. Administrative roles can be viewed as an extension to the original role hierarchy: for each pair (a, r), there is an additional node a and an edge from a to r. Administrative scope of administrative roles now simply becomes the administrative scope as defined earlier, but taken over the existing and extended hierarchies as a whole.

We note that existence (nonexistence) of administrative roles provides a variation of the *effects* of `AddRole` (`DeleteRole`) operation from Table 1, from the point of view of the extended (administrative) part of the hierarchy. What is required is some way in which administrative roles can be added, removed, and modified. Indeed, RHA_3 does not say how the `admin-authority` relation changes. RHA_4 includes two further operations, `AddAdminAuthority` and `DeleteAdminAuthority`, thus defining how the relation is modified. `AddAdminAuthority` adds an edge to the extended hierarchy, so it is a variation of the `AddEdge` operation we considered in Table 1. Similarly, `DeleteAdminAuthority` removes an edge from the extended hierarchy, and is thus a variation of the `RemoveEdge` operation from Table 1. There is no need for `AddRole` and `DeleteRole` operations to change, as we do not expect them to make direct modifications to `admin-authority`. (Their execution however may have effects on the relation; this is discussed soon.) Therefore, they may be reused in the extended hierarchy by providing suitable arguments.

[1] describes direct updates (i.e. updates via `AddAdminAuthority` and `DeleteAdminAuthority`), as well as indirect updates to the extended hierarchy. An indirect update to the extended part of the hierarchy occurs as a consequence of some other changes in the hierarchy, for example when other roles and edges are added or removed. We will not detail cases in which these updates occur, but it is worthwhile to consider their effects on flexibility. When the extended hierarchy is indirectly modified, there may be violations to the `admin-authority` relation. For example, we may remove some role r by executing `DeleteRole`. If it is the case that $(a, r) \in \texttt{admin} - \texttt{authority}$, i.e. there is an edge from r to some administrative role a, a needs to be reconnected to all the other roles lower in the hierarchy that can be reached via r, in order to preserve its administrative scope after r has been removed. Correctness is ensured simply by adding tuples (a, r') to `admin-authority`, where r' are direct descendants of r. The extra work required is a simple, "clean" check on the data structures used to implement the model. This is the way side effects of all operations that cause indirect updates are mitigated, so RHA_4 has the potential for greater flexibility over models that are not as elegant.

What is missing from RHA_4 is a mechanism to perform user–role and permission–role assignments. SARBAC extends RHA_4 by introducing the notion of constraints and defining two additional relations, `ua-constraints` and `pa-constraints`. `ua-constraints` is a set of $(user, role)$ tuples that keeps a track of user to role assignments. Similarly, `ua-constraints` keeps a track of permission to role assignments. A SARBAC constraint is a conjunction of roles from the hierarchy. For a user to satisfy a constraint, it must be assigned to all the roles in the constraint (hence the conjunction), either explicitly or implicitly (by inheritance). Similarly, for a permission to satisfy a constraint, it must be present in all the roles listed in the conjunction, either explicitly or implicitly. (In the case of permissions, the implicit list is a list of ancestors rather than predecessors, since permissions do not inherit but are inherited.) Together with administrative scope, constraints act as an enabling mechanism for the execution of operations that modify `ua-constraints` and `pa-constraints` relations. If an assign or a revoke user (role) operation satisfies the administrative scope rules and all the constraints, then it may be executed. For our discussion on flexibility, we note that correctness of `ua-constraints` and `pa-constraints` must be preserved during indirect updates, as it was the case with `admin-authority`. Handling of this is similar to the handling of effects of indirect updates to `admin-authority` (i.e. we add/remove tuples from `ua-constraints` and `pa-constraints`), but some operations cause undesirable changes to con-

straints. Since handling of the effects of indirect updates is non–uniform (i.e. what to do to preserve correctness depends on particular indirect operation, as opposed to RHA_4), we feel that some flexibility may be lost.

3.4 Completeness, Versatility, and Practicality

An access control model is made up of components. For example, the components of SARBAC are its set of roles, its set of permissions, the `admin-authority` relation, etc. For dynamic changes to take place, some of these components must change over time. Such components are called dynamic. For example, the set of roles changes dynamically when `AddRole` and `DeleteRole` operations are executed. The components that are not changed dynamically are called static. The state of a model at some particular time step are the contents of the dynamic components. When a dynamic change occurs, the model undergoes a change in its state. Completeness is defined in [1] in terms of a function that maps states to states, that is, a function which specifies the state transitions. A model is said to be complete if such a function is defined, or if it can be defined. Versatility and practicality are not formally defined. Versatility is taken to be the "permissiveness" of a model: if a model A permits the execution of some operations that are restricted in some other model B, then A is said to be more versatile than B. It makes sense that versatility is a desirable quality; it is desirable to have a model which will permit intuitive operations to take place than a model which will require an execution of a few operations before a desired change is made, or a model that will not permit these operations at all. Of course, what is intuitive and what is not is fairly subjective. Practicality is the ability of a model to be implemented and applied to particular situations. We agree that this is another key quality, but we are skeptical about claims of practicality prior to implementation attempts. We are more comfortable with the latter part, since situations in which a model may be applied can be deduced from the specifications of the components. However, we believe that this too is related to implementation: potential applications of a model remain potential until that model is implemented.

[1] details a comparison between SARBAC and ARBAC97 [5]. It looks at completeness, practicality, and versatility of both.[1] The argument for completeness is the dynamic nature of SARBAC relations. ARBAC97's equivalents are not dynamic components. "For example, if a new role is added to the hierarchy, how can constraints be imposed on the assignment of users

[1]The comparison also includes simplicity. We have omitted this, as we believe that simplicity is even more subjective than practicality.

(and permissions)?", [1] (p22). We agree with this argument, as completeness defined in [1] certainly enables dynamic changes to occur without any extra work. The argument for versatility is based on looking on the possible ways that operations may fail. It is concluded that many "reasonable" operations fail in ARBAC97, while they are allowed to occur in SARBAC. Hence, SARBAC is claimed to be more versatile than ARBAC97. The practicality argument comes from the lack of detail in specification of operation of ARBAC97. SARBAC does contain specifications on how to build and administer role hierarchies , but "... it is not obvious how ARBAC97 is intended to work", [1] (p26). We will take these arguments as sufficient to convince us that SARBAC is complete, and more versatile and practical than ARBAC97.

4 Graph–Based Formalism

4.1 Graph–Based Formalism and RBAC

This section will briefly overview graph–based formalism for RBAC, which was introduced in [3], and discuss the differences of the two approaches.

A graph is a pair (V, E), where V is a set of vertices and E is a set of vertex pairs, representing the edges. In graph–based formalism, each node and each edge have a type. We represent RBAC by a type graph – see Figure 2(a). Some graph G belongs to a class of graphs defined by a graph type if one can find for each node and edge in G the corresponding edge and node type in the type graph [3]. Figure 2(b) is taken from [3], and it shows an RBAC state graph which can be reduced to the type graph in Figure 2(a). Graphs are transformed by *graph rules*. A graph rule states how a graph G is transformed to a graph H by detailing the changes to an applicable subgraph of G, and the conditions that need to occur in G for the rule to apply. See Figure 2(c) for an example.

Differences between SARBAC and graph–based formalism prevent us from comparing the two approaches directly. We will insist on referring to graph–based formalism as a formalism and SARBAC as a model from now on. Let us make a distinction between what we mean by a formalism and what we mean by a model. We will take the term formalism to refer to a theoretical description (however rigorous it may be) of some sort. We take the term model to mean an abstraction detailing the rules that must be followed in an implementation. As we have seen, graph–based formalism is a formalism of RBAC as an idea – it uses graph theory and graph transformations to formalize the concepts and operations in RBAC. For every

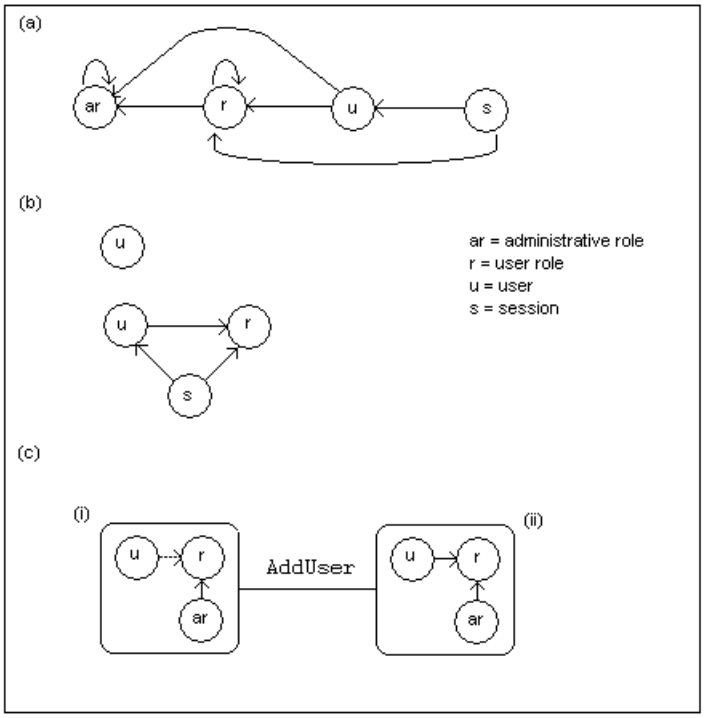

Figure 2: (a) A type graph for RBAC, showing how components interact together. (b) An RBAC state graph that reduces to the RBAC type graph. (c) `AddUser` graph rule. (i) is the subgraph of G that will be modified, and (ii) is the result of the rule. The dashed line in (i) represents a condition for the rule to take place: such an edge must not exist in graph G for the rule to be applicable.

operation given in Table 1, there is a graph–theoretical equivalent. Graph–based formalism can thus be used to provide a graph–based correctness of RBAC (see Section 5 of [3]). It is also used to define, and prove correctness of models, for example ARBAC97, or a model based on administrative scope (discussed in the next subsection).

We feel that it is unjust to dismiss graph–based formalism in favor of SARBAC on the grounds of it not being complete, versatile, and practical. We believe that graph–based formalism and SARBAC serve different purposes (one being a formalism and the other a model), and are thus not able to be compared directly. It seems to us that completeness, versatility, and practicality are attributes exhibited by models. What we can do is compare a model described by graph–based formalism and SARBAC. Obviously, if we map ARBAC97 to graph–based formalism (as it was described in [3]), then this comparison becomes the one in [1], and we conclude that SARBAC is superior because it exhibits completeness, versatility, and practicality. It would be interesting to consider a model using the notion of administrative scope described by graph–based formalism, and compare that with SARBAC. This is done in the next subsection.

4.2 Graph–Based Formalism and Administrative Scope

[4] shows an application of the concept of administrative scope in graph–based frameworks. It maps SARBAC to graph structures. We make a summary of the mapping below, and discuss its completeness, versatility, and practicality in the next subsection.

The role hierarchy is a directed graph. There is an edge from a node r to a node r' if r is an immediate ancestor of r' (that is, r is superior to r'). The `admin-authority` relation is modeled the same (i.e. by an extended hierarchy), although these edges in extended hierarchy are of a different type than those edges in the original hierarchy; this is useful when preserving correctness after an operation executes. The definition of administrative scope is the same, and the property is checked by the `adScope(a, r)` procedure. `adScope(a, r)` recursively checks if every ancestor of r is controlled by a – this is the same as saying that every path from r to the top of the hierarchy goes through a, which is our informal notion of administrative scope, as mentioned earlier.

Let's see how operations in SARBAC are mapped to graph structures. *AddRole* and *DeleteRole* use graph rules from [3] to perform addition and deletion of roles. A check is in place in order to enforce administrative scope. If an administrative role a is adding a role r, then a needs to have admin-

istrative scope over the roles which will be neighbors of r. Similarly with $DeleteRole - a$ needs to have r in its administrative scope (by definition, it also has all the neighbors of r, so this is not checked). [4] also specifies that we need to do some extra work when adding and deleting roles. Namely, when adding a role, we need to make sure that all redundancy is avoided (this is done by a `clean` graph rule, as defined in [4]), and when deleting a role, we need to make connections between roles which were previously connected via r (this is done by a `complete` graph rule, as defined in [4]). `AddEdge` and `RemoveEdge` are very similar to `AddRole` and `DeleteRole`. In this case, appropriate graph rules will be applied, with a forced consistency check. The `AddAdminAuthority` (`DeleteAdminAuthority`) rule lets an administrative role a add (remove) some role r from the administrative scope of another administrative role a'. In the case of adding r to the scope of a', a check is needed to ensure that both r and a' are a's scope, and that r is outside the scope of a'. In the case of removing r, a check whether a' is in a's administrative scope is required. These two operations then behave exactly the same as `AddEdge` and `RemoveEdge`, but on the special, administrative edges (found in the extended part of the hierarchy).

A note is required on indirect updates, for our running discussion of flexibility. [4] notes that, since the correctness is ensured after executing the operations by forced consistency checks, one does not have to pay special attention to indirect modifications of the `admin-authority` relation, and that, "therefore, no special treatment is necessary for this case" (p5). Although this statement makes the graph–based approach to administrative scope sound more flexible than RHA_4, the two are essentially doing the same thing. It is not possible to comment on precise differences in flexibility, since [1] does not give details on how these updates are performed, whereas [4] does define the `clean` and `complete` graph rules.

For the purposes of user–role and permission–role assignments, constraints and relations are defined in the same way as for SARBAC. Recall that a constraint is a conjunction of roles. A user satisfies some constraint if it is (explicitly or implicitly) assigned to these roles. Permissions satisfy a constraint in a similar way to users. User assignment is carried in the same way as in SARBAC: the conditions of operations check whether administrative scope holds and whether all the constraints are satisfied. No mention of handling of constraint consistency during indirect updates is made, so we will assume that they are treated the same as in SARBAC.

4.3 Completeness, Versatility, and Practicality

We have seen how RBAC operations, relations, and constraints map to graph structures in the previous subsection. We have seen that effects and enabling conditions of operations in both models are identical. Graph–based model of administrative scope does not add any extra functionality to SARBAC. We therefore conclude that both models are identical.

Since both models are identical, the completeness and versatility they exhibit must be the same. We have mentioned earlier that SARBAC is complete because the `admin-authority` relation is dynamic. This is also the case in graph–based administrative scope model [4] (p4). They are both complete, according to our earlier definition of completeness. Since all operations will fail or succeed in the same way (they are, after all, same operations), the two models have the same versatility.

Practicality is very hard to measure from a model's specification, since the abstraction does not allow all implementation and application issues to be addressed. We argue that, since SARBAC and graph–based model of administrative scope are identical, the graph–based approach must be at least as practical as SARBAC, however much that may be (see Section 3.4). SARBAC operations are not defined as clearly as operations in the graph–based approach. The authors of [1] state that they "intend to give operational semantics for [SARBAC] by writing pseudo code functions to implement the SARBAC operations" (p30). These operations are already defined in the graph–based approach, by using graph rules. This leads us to believe that the graph–based model of administrative scope is more practical than SARBAC.

5 Conclusion

Although SARBAC is quickly to dismiss graph–based formalism on the grounds that it is not complete, versatile, and practical, we have seen how it can be mapped onto a graph–based model. From our discussion, we have concluded that not only can graph–based formalism define a model on equal footing with SARBAC completeness– and versatility–wise, but it can also define a more practical model, which enjoys a more detailed operational specification.

References

[1] J. Crampton and G. Loizou. Administrative Scope: A Foundation for Role-Based Administration Models. ACM Transactions on Information and System Security, 3(4):207-226, November 2002.

[2] D. Ferraiolo, and R. Kuhn. Role-based access control. In 15th NIST-NCSC National Computer Security Conference, pages 554–563, Baltimore, MD, October 13-16 1992.

[3] M. Koch, L. V. Mancini, and F. Parisi-Pressice. A Graph Based Formalism for RBAC. ACM Transactions on Information and System Security (TISSEC), 5(3):332-365, August 2002.

[4] M. Koch, L. V. Mancini, and F. Parisi-Pressice. Administrative Scope in the Graph-based Framework. Proceedings of the ninth ACM symposium on Access Control models and technologies, p97-104, 2004.

[5] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role–based administration of roles. ACM Transactions on Information and System Security 1, 2, 105-135, 1999.