Approvability

Jon A. Solworth
University of Illinois at Chicago
851 S. Morgan Street, M/C 152
Room 1120 SEO
Chicago IL 60607-7053
solworth@cs.uic.edu

ABSTRACT

Consider a set of users who collectively perform a sequence of actions to complete a task. Separation of duty constraints hold when there are restrictions which are intended to require that not all actions are performed by the same user.

The approvability graph is introduced to describe the sequences of actions which correspond to one or more tasks. The graph can represent multiple possible outcomes (different completions from the same starting point) as well as allowing for repeated actions. Hence, the graph describes a set of sequences, not necessarily finite, which define when a task is complete.

The graph-based mechanism also describes separation of duty constraints between different actions, ensuring that different actions are performed by different users. (It can also require different actions to be performed by the same user.)

Algorithms are presented to analyze the number of users needed to ensure that any such sequence can be completed, even in the presence of loops or alternative outcomes. The various properties that arise in approval sequences are then explored to characterize well formed systems and to examine their complexity. In particular, we show how to achieve bounds on the number of users which must be members of each role.

Determining the minimum number of users to complete a dynamic separation of duty task is proven to be NP-Complete.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls

General Terms

Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ASIACCS '06, March 21–24, 2006, Taipei, Taiwan.
Copyright 2006 ACM 1-59593-272-0/06/0003...$5.00.

Keywords

Authorization, Authorization Properties, Separation of Duty

1. INTRODUCTION

Consider a *task* that consists of multiple *actions*, where each action is performed by a single user. *Separation of Duty (SoD)* constraints are intended to ensure that a single user does not perform every action in a task. Specifying separation of duty problems, in general, involves (1) specifying the actions that are part of the task (since if actions could be arbitrarily bypassed it would be impossible to enforce separation of duty) and (2) the constraints between actions.

Separation of duty problems are important for several reasons. First, they are reserved for the most sensitive tasks and therefore are crucial to security. Second, they are by definition too important for a single person to perform making them unique among authorization issues. Third, they must be described by specialized mechanisms. And fourth, they are broadly used, for example in financial systems to confirm large transactions; in military applications such as to control the launch of nuclear weapons; and in medicine for second opinions before non-emergency surgery.

We will define an *approval sequence* (in Section 4.1) which completes a *task* as a sequence of *actions* along with the users who performed them. For example, an accounting approval sequence might require the issuing of a purchase order by a purchase clerk, the receipt at the loading dock of the purchased item by a shipping clerk, the receipt by the end user of the item, followed by the payment by an accounts payable clerk. To an organization, it is a completed approval sequence that denotes finished work, that is a completed task. In general, the approvability sequence is neither fixed—for example, multiple revisions may be necessary—nor is there a single possible outcome—for example, a loan application may be approved or denied.

There are two major types of Separation of Duty (SoD)[1]:

Static SoD (SSoD): The individuals are partitioned into groups or roles, and different roles may be assigned to different actions of an approval sequence. SoD constraints ensure mutual exclusion between the roles[2].

[1] Sandhu [25] attributes to [9] the static vs. dynamic SoD terminology.

[2] Note that pairwise-disjoint roles are sufficient, but not

Dynamic SoD (DSoD): Dynamic SoD ensures that for a given task, different actions are performed by different individuals, even if both actions are governed by the same role.

When there are SoD constraints, then by definition two or more actions are required and the actions must be performed by more than one individual. It reduces fraud for at least two reasons. First, it is necessary for two or more individuals to fail to act in the interest of the organization, which has an inherently lower probability than the failure of a single individual. Second, collusion requires that one party proposes fraud to another. The second party may report the first party to authorities; alternatively, if the second party does not report the first party he runs the risk that the first party was testing him at the behest of authorities (and hence the first party's offer was not genuine). Therefore, it is not safe for either party to cooperate or even discuss an offer of collusion.

While SSoD is easy both to specify and to analyze, dynamic SoD is neither. But DSoD is far more flexible. It enables the protection to be tuned so that for the same number of people, either more separation can be obtained or it is easier to find someone who can perform an action (or some combination of these two).

In other authorization mechanisms, it suffices to assign one sufficiently trusted user to perform each action. Unfortunately, this is insufficient for DSoD since even if a user is highly trusted, she may not be able to perform a given action because she has performed some other conflicting action earlier: In this sense, DSoD actions *consume* users since, as a consequence of performing an action a user is unable to perform future actions to which she would otherwise be permitted. Therefore, an organization's approval sequence for DSoD depends on both organizational size and structure. If only n people are associated with an organization, then n is the maximum number of users that can be involved in a task. Smaller organizations will need to decide, for example, between lesser SoD and having executives perform actions which in a larger organization would be performed by clerks. Hence, organizations must design and analyze their own approvability sequences with the desired degrees of SoD.

It is important to analyze the (maximum) number of users needed to complete a task to ensure that tasks do not become *stuck* and therefore unable to complete—a kind of denial of service. The only paper which we are aware of that analyzed completion of tasks is the seminal paper of Bertino, Ferrari, and Atluri (BFA) [3]. BFA modeled a workflow as a collection of actions, described constraints between actions and a bound on the number of times an action could be executed. Given this description, they developed an algorithm which provides an exact solution of the assignment of users to actions, if one exists. Unfortunately, if the approval sequence can have loops or alternative outcomes then the number of times each action executes is unknown. BFA suggested that, in this case, an estimate of the number of actions is used and, if exceeded, the algorithm could be rerun. However, under such circumstances the algorithm could fail to find a solution and a workflow could get stuck.

necessary to obtain SoD. Consider the following sequence of groups of individuals a, b, c which exhibit SoD $[\{a, b\}, \{b, c\}, \{c, a\}]$ since at least two individuals must be involved in approval.

Our focus here is on determining sufficient conditions so that a SoD task never gets stuck. In this paper, we

- introduce the *approvability graph* to model tasks. The approvability graph can represent multiple possible completions (e.g., the purchase order was issued or was cancelled), loops, and linear sequences as well as the DSoD constraints between actions. It is the first model of SoD which allows loops.
- show well-formed conditions on the approvability graph, so that given a sufficient number of users per role, the task cannot get stuck.
- show a polynomial-time algorithm to determine a sufficient number of users per role. Note that the assignment of users to roles varies with personnel changes; the approvability graph (i.e., the workflow), on the other hand, does not. Minimum cardinality per role ensures that the tasks do not get stuck even as personnel change[3].
- show that the analysis of even very simple DSoD problems is NP-Complete.

This last issue is especially interesting for two reasons: (1) it indicates that runtime DSoD scheduling may be too expensive and (2) it is a lower bound on the *intrinsic* complexity of a SoD problem. The only other complexity analysis of SoD we know of shows that the complexity of a particular SSoD *solution* is NP-Complete [19]. Of course, showing the *problem* has high complexity implies that any solution must have high complexity.

The remainder of this paper is organized as follows: Section 2 describes related work; Section 3 describes the approvability graph and gives examples of its use examples; Section 4 describes the analysis of these systems; and finally we conclude.

2. RELATED WORK

SoD has a very long history, going back at least to Multics in the computer security literature [23] (where it is called separation of privilege). Saltzer and Schroeder attribute its first use in computer systems to Needham in 1973. But its use outside of computers is far older, going back thousands of years. We shall not attempt to trace the non-computer history here.

Lipner described a use of SSoD with respect to program development so that those who installed software were separate from those who developed it. Its purpose is to make it harder

[3]This assumes that same user constraints are not used or can be overridden, for if an individual leaves the company she is not longer available to perform future tasks.

for a programmer to insert surreptitious code [20]. Clark-Wilson used it explicitly to counter fraud [5]. Both Clark-Wilson and Lipner have their antecedents in non-computer systems.

Sandhu developed an elegant notation for describing action sequences [24]. The notation allowed for DSoD, and could require two different actions to be performed by different users or the same user. (Sandhu's notation even allowed weighted approvals: For example, a vice president could approve something that otherwise would require approvals by two different supervisors.) However, other than weighted approvals, each task consisted of a fixed sequence of actions.

Crampton has modeled SoD as a partially ordered set of actions enabling parallel actions but requiring all actions to be performed (that is, it supports neither alternation nor loops) [7]. He describes a runtime implementation model using blacklist sets to ensure that constraints are not violated [6]. Knorr and Weidner use Petri Nets to represent SoD, but it is clear from their analysis that they did not consider loops [17].

Hitchens and Varadharajan [13] describe a set-based language called Tower for modeling RBAC that can describe sequences of SSoD and DSoD, as well as several other types of authorizations.

Nash and Poland [21] introduce object-based SoD in which the separation is on an object-by-object basis. This is a fine-grained instance of DSoD. Since the purpose of DSoD is to increase flexibility while maintaining appropriate constraints, this fine-grained version is superior to coarser-grained versions.

Karger describes an operating system implementation using capabilities and lattices [16]. He describes the system both in terms of its verification needs and user interface as being extremely complex. Foley [11] describes SoD using lattices and high water marks. The lattices used by both Foley and Karger are more restricted than our approvability graph since they must be acyclic.

Simon and Zurko presented a taxonomy of SoD and gave a description of their specification [27]. The language appears to be purely for policy specification, it does not seem to be implementable.

Gligor, Gavrila, and Ferraiolo describe a formal definition of various SoD policies, and show the relationship between them [12]. They also investigated composition of SoD. Furthermore, they note that SoD mechanisms have not been successfully implemented in operating systems, which is one of the goals for the model presented here. We present operating system level mechanisms which can implement approvability graphs in a second paper in this proceedings [22].

SoD has been extensively explored in the context of Role-Based Access Controls (RBAC) [10, 26], including [9, 18]. Ahn and Sandhu used first-order predicate logic to describe SoD [1]. DSoD is typically represented by permissions which cannot be taken on by the same user within a session.

Joshi, Bertino, Shafiq and Ghafoor have examined how to specify SoD in GTRBAC [15].

Li, Bizri, and Tripunitara (LBT) showed that verifying a SSoD policy—that is, at least n people are required to complete a task—using mutually exclusive rules is NP-Complete [19]. Their complexity result is about a particular mechanism to implement SoD, whereas the complexity result here is inherent in SoD. As described in the introduction, DSoD is fundamentally different from SSoD in that only DSoD consumes users.

Jaeger and Tidswell [14, 28] used constraints and inheritance in their Dynamically Typed Access Control model to implement SoD.

Chinese Wall [2, 4] is a similar problem to SoD in that they share a "different from" constraint. The Chinese Wall constraint, however, is on objects while the SoD constraint is on people.

Vimercati, Paraboschi, and Samarati [8] describe the general mechanisms necessary to support SoD in their survey of access controls.

3. APPROVABILITY

In an approvability graph each *action specifier*—which specifies the type of an action such as receipt of item by end user—is modeled as a directed edge and labeled with a role, or group. Paths starting from initial nodes in the approvability graph specify allowable sequences of actions and the role that a user must assume, or be a member of, to perform an action. Each traversal of an action specifier edge in the path corresponds to a unique action. The nodes of the approvability graph summarize the state of the approval.

The approvability graph directly represents DSoD through constraints on action specifiers. In addition, if the sets of *mutually exclusive roles*—in which a user can never be a member of multiple roles in a set—are known, the approvability graph can also represent SSoD. Hence, SSoD is represented by different roles associated with different action specifiers; DSoD is represented by "different user" constraints between two edges and the roles associated with these edges need not be the same.

Formally, an **approvability graph system** is a 6-tuple $\langle V, E, C_d, C_s, L, R\rangle$ where $\langle V, E\rangle$ is a directed graph called the **approvability graph** and C_d and C_s are sets of pairs of edges, L is a set of edges, and $R : E \rightarrow \mathit{Role}$ where Role is a set of roles. We shall be informal and not distinguish between the approvability graph system and the approvability graph it contains, calling them both approvability graphs.

The approvability graph contains the following types of nodes:

initial A starting point in the approval process.

final A completion of a task. Final nodes have no outgoing edges.

intermediate Nodes which are neither initial nor final nodes. Intermediate nodes have both incoming and outgoing edges.

There can be multiple nodes of each type.

The edges E of the approvability graph correspond to action specifiers. Each edge e is labeled with a role $R(e)$ whose members can perform the specified action. Without loss of generality, we will assume that every edge and node is reachable from some initial node.

Finally, constraints between action specifiers are used to represent SoD constraints. The constraints between two distinct edges (action specifiers) e_0, e_1 are of two forms, C_d and C_s respectively:

different user meaning that the users who perform an action specified by e_0 must be disjoint from those who perform an action specified by e_1 (thus representing DSoD) [4].

same user All of the actions specified must be performed by the *same* user. *Same user* constraints must be between edges labeled with the same role.

In addition, there is a constraint for a single edge (L):

self-same user the actions corresponding to the action specifier are all performed by the same user.

Note that the *same user* and *self-same user* constraints inherently requires a particular user to perform some *future* action. As this may affect the urgency with which such tasks can be completed if the specified user is (temporarily) unavailable, the *same user* constraint should be used sparingly.

The following notation is used for approvability graphs: The successors of a node v, $Succ(v)$ are a set of nodes such that if $v' \in Succ(v)$ there is an edge from v to v'. $Succ^*(v)$ is the transitive closure of $Succ$ on v. Finally, I is the set of initial nodes and F is the set of final nodes.

An approval sequence captures a traversal of the approval graph starting with an initial node and ending with a final node.

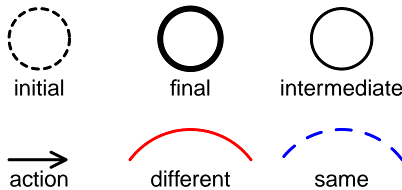

Figure 1: Approvability key

The key for the approvability graph figures is summarized in Figure 1. Initial nodes are drawn as a dashed circle; final nodes are indicated as circles with a wide boundary; intermediate nodes have normal solid boundary; and edges are drawn as solid arrowed lines. *Different user* constraints are indicated by a solid (red) undirected arc between two edges while *same user* and *self-same user* are indicated by a dashed (blue) undirected arc between two edges. Note that in the examples that follow, the only constraints are *different user*.

[4]This is worded to take into account edges being traversed 0, 1, or multiple times.

The approvability graphs are next illustrated through a series of examples.

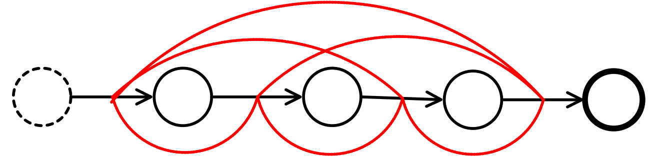

Figure 2: Purchase order DSoD

3.1 Simple DSoD

In Figure 2 an example of a 4-action sequence is shown in which the same role performs every action, and no user can perform more than one action. Hence, there is a DSoD *different user* constraint between every pair of actions. This graph corresponds to the 4 stage purchase sequence in the introduction of this paper: Each edge is labeled with the role *clerk*, and hence constraints are needed between each pair of actions (edges).

Note that if instead we had separate, pairwise disjoint roles for purchase order clerk, shipping clerk, end user, and payables clerk, then no constraints would be needed.

3.2 Medical consultation

We next describe a medical example, summarized in Figure 3. In a medical consultation the primary physician decides that consultations are needed with one or more consulting physicians. She creates a role containing the physicians she wishes to consult. The cardinality of the created role determines the initial node in the approvability graph. Each consultant performs tests and then inserts a report, thus performing their part in the approval sequence (The consultants count down in the order they finish, there is no predetermined order).

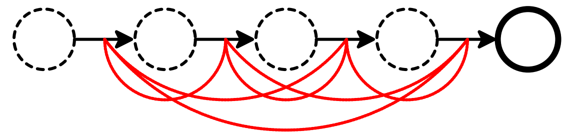

Figure 3: Consulting physicians

The approvability graph ensures that each physician inserts their report exactly once, since there is a SoD request in the "count down" portion of the approvability graph. Note that this approvability graph is capable of supporting consultations with between 1-4 physicians; an approvability graph supporting up to N consultations could be constructed with $N + 1$ states.

3.3 Weighted approval

We now show how alternative paths can be used to provide Sandhu's weighted approval (see Section 2). Figure 4

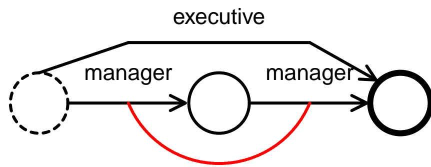

Figure 4: Weighted approval

shows a simple approval graph in which either two different managers or a single executive can approve.

3.4 Document revision

We next describe an undoubtedly familiar problem to the reader; preparing and revising documents in which multiple people take part in the authorship, editing, and release. There are many interesting concurrency issues associated with this problem, which we ignore here. Instead we focus on tracking who did what, and ensuring drafts are not shared until they are proofed and approved.

We next describe the document states for authors to edit a document and then have it approved by the manager.

Draft Document is undergoing revision.

Edited The author has finished editing the document, and is ready to move onto the next stage.

Proof Someone other than the author of the last edit checks that the document is sound. This can be generalized to multiple proofreaders.

Released The manager has accepted the document.

Reject The manager rejects the document and work ceases on it.

We note also, that one of the possible actions is that the document is referred back to the author with comments.

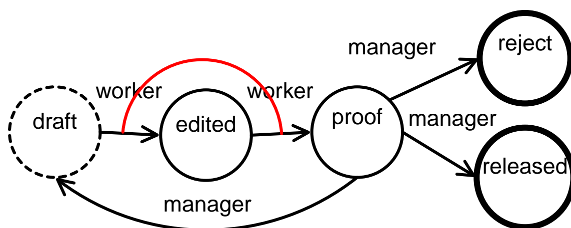

Figure 5: Document revision approval graph

In Figure 5, the approvability graph is shown. The DSoD specification is quite small, since the proof reading should be done by someone other than the author.

There are two interesting issues in this approvability graph. The first is multiple final states, released and rejected. This results in different possible approval sequences when tracing the approval graph. The second is that there is a loop consisting of three action specifiers (and three nodes); if the loop is traversed x times, then the 3 action specifiers correspond to $3x$ actions.

4. ANALYSIS

In this section we consider approvability sequences—which describe actions and the users who perform them. While the approvability graph specifies both the sequencing of actions and the roles that can perform the actions, it does not describe which users are associated with which roles. Hence, to generate approvability sequences, both the *user-role assignment*—that is, determining the membership of each role—and an approvability graph must be given.

We describe *user-action assignment*—how users can be assigned to perform actions while ensuring that approvability sequences can be completed along any path from the node after that action to a final node in the approvability graph. In general, the user-action assignment requires that a user can perform an action only if (1) she is a member of the corresponding role, (2) no constraints are violated with earlier actions in the action sequence, and (3) assignment of the user to the action does not prevent the completion of any task which contains the action sequence as a prefix. The first two points concern *past actions* while the third point considers *future actions*. Two methods of determining the user-action assignment are considered here:

scheduled approvability depends on both past and future actions

unscheduled approvability depends only on past actions.

In scheduled approvability, the user that will perform a given action is selected *when the task begins* thus locking in specific users to future actions[5]. In unscheduled approvability, the user who performs an action is determined when the action is performed. While unscheduled approvability may require more users per role, it avoids delaying a task because the selected user is unavailable at the time the action is to be performed. Thus, as long as *some* user is available which does not violate past constraints, unscheduled approvability does not delay the task, and hence the urgency with which the task can be completed is not affected. BFA considered only scheduled approvability.

We will show modest restrictions on the approvability graph which ensure that both scheduled and unscheduled approvability are possible given sufficient number of users per role. Given such approvability graphs we will

1. show how to determine the exact bound on the number of users for unscheduled approvability when all *different user* constraints are between edges labeled with the same role;
2. give upper bounds on the cardinality of the user-role assignment so that approvability graphs can be unscheduled approvable; and
3. show that the problem of DSoD is NP-Complete.

[5]One can try to reschedule the task later, but an alternate schedule may not be possible and in addition we shall show that scheduling is NP-Hard.

4.1 Scheduled and unscheduled approvability

We begin by defining the nodes which are reachable from a given node, if there are enough eligible users to approve.

DEFINITION 1. *An* ***action sequence*** *for approvability graph system S and user-role assignment A is an alternating sequence of nodes and users beginning and terminating with a node and denoted $[v_0, u_0, v_1, u_1, \ldots, v_n]$ where $v_0 \in I$ and for $0 \le i < n$*

- $v_{i+1} \in Succ(v_i)$,
- *for all $j \neq i$, if the edge $[v_j, v_{j+1}]$ has a different user constraint with $[v_i, v_{i+1}]$, then $u_i \neq u_j$.*
- *for all $j \neq i$, if the edge $[v_j, v_{j+1}]$ has a same user or a self-same user constraint with $[v_i, v_{i+1}]$, then $u_i = u_j$.*
- *u_i is a member of the role for action $[v_i, v_{i+1}]$.*

An action sequence is an ***approval sequence*** *if $v_n \in F$.*

An action sequence corresponds to the current state of agreement. An approval sequence corresponds to a completed action sequence, meaning that no more agreement is needed (or possible). Of course, there can be many possible final nodes, for different sorts of agreement that can be reached.

The goal is to be able to extend every action sequence into an approval sequence, and thus complete the task. The next definition describes what it means to do that for scheduled approvability (called s-approvable) and for unscheduled approvability (called u-approvable).

DEFINITION 2. *An* ***action sequence***

$$seq = [v_0, u_0, v_1, u_1, \ldots u_{i-1}, v_i]$$

for approvability graph system S and user-role assignment A is u-approvable (resp. s-approvable) to *$v_n \in F$, written $U(seq, v_n)$ (resp. $S(seq, v_n)$), if either*

- *$v_i = v_n$ or*
- *for any $v \in Succ(v_i)$ such that $v_n \in Succ^*(v)$ and for all u (resp. exists a u) such that $seq' = seq|[u, v]$ (where '$|$' is the sequence concatenation operator) is an action sequence then $U(seq', v_n)$ (resp. $S(seq', v_n)$).*

The second point says that any action specified (by the approvability graph) from v_i and which is a predecessor to v_n must be traversable. Hence, all paths to v_n must be possible.

Note that s-approvable says there is some sequence of user actions to reach state v_n. It does not say that *any* sequence can reach approval. In Figure 6 for example, consider the user-role assignment r_0 has members u_0, u_1; r_1 has members u_1, u_2; and r_2 has members u_0, u_2. Then $[v_0, u_0, v_1, u_1, v_2]$ is s-approvable to v_3 since $[v_0, u_0, v_1, u_1, v_2, u_2, v_3]$ but

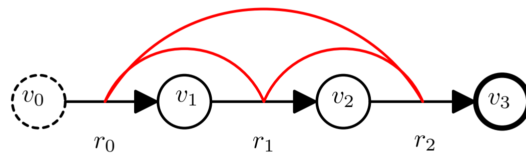

Figure 6: An approvability graph which is s-approvable but not u-approvable (depending on role assignment)

$[v_0, u_0, v_1, u_2, v_2]$ is *not* s-approvable to v_3. Note that sequences which are only s-approvable may lock *specific* individuals into performing future actions, even in the absence of *same user* constraints, and hence the completion of the task may depend upon the availability of specific users.

In contrast, u-approvability says that any user u which is the member of a role and which does not have a SoD conflict with a previous action can perform that action while ensuring that any reachable action can be performed. Hence, u-approvability looks only at past actions yet also ensures that the workflow does not get stuck. (We shall develop suitable restrictions both on the approvability graph and on the cardinality of each role associated with an edge to show that u-approvability holds.)

The above deals with particular final nodes; the next definition defines system-wide properties.

DEFINITION 3. *The* **system is s-approvable** *(resp.* **system is u-approvable***) if for all $v \in I$ and $v' \in F$ it holds that $S([v], v')$ (resp. $U([v], v')$).*

4.2 Bounding the paths in the approvability graph

Note that although the number of states and successors is bounded, the length of paths is not bounded since there can be cycles. We next analyze cycles to bound the number of edges that need to be considered in a user-action assignment.

DEFINITION 4. *An edge e in an approval graph* **consumes a user** *if there exists e' reachable from e in the approvability graph and there is a different user constraint between e and e'.*

If an edge consumes a user, then the ability to complete the approval sequence depends on more than just the role being nonempty. Note that in SSoD users are not consumed, and hence the role must only be non-empty for the approval sequence to complete. Of course, more users can be hired to complete a DSoD task but it is neither practical nor secure to add new users because your access control system is stuck!

DEFINITION 5. *An edge e in the approval graph* **cyclically consumes a user** *if e is part of a cycle, e consumes a user, and there is not a self-same user constraint on e.*

Without the *self-same user* constraint, successive iterations of an action (in a loop) may be performed by different users.

Hence, these users become unavailable for any future action which has a *different user* constraint with e. This limits the organization's flexibility to assign users to actions, and hence delays tasks until a specific individual is available.

PROPOSITION 6. *If there exists an edge in an approvability graph which cyclically consumes a user, then the number of users necessary for u-approvability may be unbounded.*

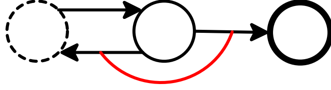

Figure 7: Unbounded number of users needed

PROOF. Consider the loop in the Figure 7. Assume that there are n users in a role: By traversing the loop n times and using a different user on each iteration of the bottom edge of the loop it will be impossible to reach the final node. □

Note that the graph in Figure 7 is s-approvable, since the same user can be used for the action corresponding to a given action specifier.

PROPOSITION 7. *If no edge in the approvability graph cyclically consumes a user, then it suffices to consider sequences of actions in which duplicate actions have been removed to determine whether it is u-approvable or s-approvable.*

PROOF. We note that each edge in a successive iteration either has no SoD constraint with successive edges or is required to be performed by the same user each time. Hence, multiple iterations of the loop do not change the assignment problem for later actions. □

The previous lemma gives sufficient conditions to ensure that multiple traversals of an edge are the same as a single traversal with respect to approvability. The next theorem gives minimum role cardinality to ensure tasks do not get stuck for approvability graphs in which constraints are always between edges labeled with the same role.

THEOREM 8. *If an approval graph does not have any edges which cyclically consume a user, and all of the different user constraints are between edges labeled with the same role, there is an algorithm which can determine the minimum number of users in each role and which guarantees u-approvability, if u-approvability is possible.*

PROOF. Let N be the number of edges in the approvability graph. Consider a sequence of edges $[e_0, e_1, \ldots, e_n]$ where each edge in the approval graph appears at most once. A sequence is used to list the edges in an approval sequence in the order they were first visited. Then if there is a path in the approval graph which traverses the edges in sequence, then that path can have at most N edges between each edge in the sequence (since cycles would be superfluous). It is sufficient to consider all such paths where e_0's *from* node is an initial node and e_n's *to* node is a final node, since other paths just repeat edges which by the conditions of the theorem have no effect.

For each such sequence s, compute the minimum cardinality $n_{s,r}$ of each role r for which unscheduled approvability holds using its definition. We then compute for each r the maximum over all s of $n_{s,r}$. (Note that $n_{s,r} \leq N$). □

To obtain the more general case which allows *different user* constraints between different roles, it is sufficient to merge together the roles and then apply the resulting cardinality constraints to each of the roles. The algorithm for these cardinality constraints is given in the next theorem. The difference between the next theorem and Theorem 8 is that this one allows actions labeled with different roles to have *different user* constraints between them and in turn gives up minimality.

THEOREM 9. *If an approval graph does not have any edges which cyclically consume a user, there is an algorithm which can determine the number of users in each role and which guarantees u-approvability, if u-approvability is possible.*

PROOF. We once again consider the sequences $[e_0, e_1, \ldots, e_n]$ of Theorem 8 and show a bound on role size so they are u-approvable. First we merge together all roles which have a *different user* constraint between them (recall there are no *same user* constraints possible between different roles) and compute by Theorem 8 the cardinality of each role. The merged roles conceptually constitute a new role.

If after a set of roles $\{r_0, r_1, \ldots r_n\}$ have been merged, the new composite role requires user cardinality c then it suffices if each one of the roles r_i, $i = 0, 1, \ldots n$, has cardinality c. The set of *different user* constraints on each role can be satisfied since each role has a subset of the constraints as the merged role but the same number of users. □

For example, to apply Theorem 9 to the example in Figure 6, roles r_0, r_1, and r_2 are merged; the result is that three users per role are sufficient to ensure u-approvability. If each role contains the same users, three users are necessary. In general, however, tighter bounds on role size can be obtained if more is known about the relationship of the roles (for example, if a user in role r_1 can never have been a user in role r_2), but that is beyond the scope of this paper.

In the absence of loops, any approvability graph which is u-approvable is also s-approvable, since the same number of users per role who enable u-approvability will certainly enable s-approvability. In the presence of loops, s-approvability is possible where u-approvability is not: For example, the approvability graph in Figure 7 is s-approvable by assigning the *same user* each time to the edge which cyclically consumes a user. Thus, by adding a *self-same user* constraint to edges which cyclically consume a user, any s-approvable graph can be converted into a u-approvable graph.

We note that by removing duplicate edges in a path, each action specifier is effectively traversed at most once, and we can extend BFA to always find an s-approvable schedule under reasonable conditions.

4.3 Conflict graph

We say that an approvability graph is well formed if given a sufficient number of users per role it will not get stuck. In the section, the conflict graph is introduced and used to (1) determine whether an approvability graph is well formed and (2) provide a polynomial-time algorithm to determine a bound on the number of users needed for u-approvability. The **conflict graph** is an undirected graph whose nodes correspond to edges in the approvability graph. The conflict graph can be constructed only when no edge cyclically consumes a user. Further we assume, without loss of generality, that there does not exist two edges in the approvability graph with a *same user* constraint between then unless both edges can be in a path from an initial node.

The conflict graph is constructed as follows:

initialization of the graph

- nodes in the conflict graph correspond to edges in the approvability graph and
- if two nodes n and n' in the conflict graph correspond to edges in the approvability graph with a *different user* constraint between them, then there is an edge between n and n'.

collapse the graph by repeating the following step until there are no more nodes to be merged: If there are two edges in the approvability graph which have a *same user* constraint between them then merge the two corresponding nodes in the conflict graph.

For example, Figure 8 is the conflict graph corresponding to the approvability graph in Figure 6.

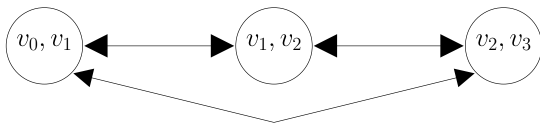

Figure 8: Conflict graph for Fig. 6. (The nodes are labeled with the edges of Fig. 6 from which they are derived.)

THEOREM 10. *The conflict graph has self loops iff the approvability graph has a sequence of edges $[e_0, e_1, \dots e_n]$ where there is a same user constraint between e_i and e_{i+1} for $0 \leq i < n$ and a different user constraint between e_n and e_0.*

PROOF. $\Leftarrow$ The approvability graph cannot have *different user* constraints between the same action specifier, therefore prior to merging nodes in the conflict graph there are no self loops in the conflict graph. Hence, to form a loop a sequence of nodes must be merged until one contains a *different user* constraint to another previously merged action. Let the *different user* constraint be between e_n and e_0. Then there must be a path of *same user* constraints in the merged node from e_0 to e_n, otherwise e_0 would not have been merged with e_n.

$\Rightarrow$ Just merge the nodes in the order $e_0, e_1, \dots, e_n$ and there will be a self loop. □

A self-cycle in the conflict graph means that the constraints can prevent some paths from being traversed. By eliminating conflict graph self cycles and approvability graph edges which cyclically consume a user, every traversal in the approvability graph is an approvability sequence.

THEOREM 11. *If the number of users at each node in the conflict graph is greater than or equal to the degree plus one of the node and there are no self loops then unscheduled approvability holds.*

PROOF. Assign one user to each node of the conflict graph so that any two adjacent nodes are assigned a different user. This can be done in a single pass over the nodes, as there is always one more user than adjacent nodes, we simply select a user who has not been assigned to any of its neighbors. □

Sufficient conditions for unscheduled approvability of a system

We note that the sufficient conditions are not necessary. Consider a conflict graph where n_0 conflicts with n_1 and n_1 conflicts with n_2. The degree of the graph is 2, and thus Theorem 11 says that 3 users will be sufficient, even though 2 users suffice.

4.4 DSoD is NP-Complete

We next show that whether a simple SoD task can complete is inherently NP-Complete. More exactly, given a conflict graph and the users that can assume a role, determine whether there is an assignment of users to actions that satisfies both the roles and the conflicts. We shall call this later problem SDSoD, for Simple DSoD. The proof is by reducing 3SAT to SDSoD.

Our goal is to show that DSoD even in its simplest form is NP-Complete. Hence, the SDSoD problem will rely on a fixed number of actions all of which need to be executed (that is, there are neither alternative outcomes nor loops). The order that actions are performed is immaterial, as there are no loops. Moreover, we shall consider only *different user* constraints between actions. Clearly, if this most simple form of SDSoD is NP-Complete, then any more complex formulation will of necessity need to be able to represent this simpler form and hence be NP-Hard.

We next review 3SAT and then show its correspondence to SDSoD.

3SAT:

Consider 3SAT, which is NP-Complete. Given a boolean expression of the form:

$$(v_{0,0} \vee v_{0,1} \vee v_{0,2}) \wedge (v_{1,0} \vee v_{1,1} \vee v_{1,2}) \wedge \cdots \wedge (v_{C,0} \vee v_{C,1} \vee v_{C,2})$$

where each $v_{i,j}$ is either x_k or $\neg x_k$ for some $0 \leq k \leq N$ and where the clauses are numbered 0 to C. The problem is whether there some boolean assignment to $x_0, x_1, \ldots, x_N$ such that the expression is **true**. For example, consider the below boolean expression:

$$(x_0 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_0 \vee \neg x_0 \vee x_1) \quad (1)$$

We now map an instance of 3SAT to into a SDSoD problem of polynomial size, thereby showing that SDSoD is NP-Hard. We describe the actions necessary to represent the 3SAT problem. For $0 \leq i \leq N$:

t_i The "user" from the role $\{x_i, \neg x_i\}$ which gets the value **true**.

f_i The "user" from the role $\{x_i, \neg x_i\}$ which gets the value **false**. Of course, t_i and f_i must have different users so that one of $\{x_i, \neg x_i\}$ gets **true** and the other gets **false**.

clause$_j$ the jth boolean clause, for $0 \leq j \leq C$.

- If both x_i and $\neg x_i$ are in clause$_j$, then there is no constraint between clause$_j$ and either of f_i or t_i otherwise
- If x_i (resp. $\neg x_i$) is in clause$_j$ then there is a *different user* constraint between clause$_j$ and f_i (resp. t_i).

The diagramming of actions and truth assignment for the general problem is shown in Figure 9 (without the *different user* constraints between truth assignment and clauses).

In Figure 10, a complete representation of 3SAT problem in Expression (1) as a SoD problem is given. For example, if clause$_0$ gets the value $\neg x_2$ then t_2 must get the value x_2 which forces f_2 to get the value $\neg x_2$ which means that clause$_1$ must get a value *different from* x_2.

Given the constraints, we now describe why SDSoD computes 3SAT. The actions t_i and f_i provide a truth assignment, since each x_i is independently chosen to be either **true** (performs t_i) or **false** (performs f_i).

The clause$_j$ is **true** if and only if at least one of its terms $v_{j,0}$, $v_{j,1}$ or $v_{j,2}$ is **true**. This is ensured by the *different user* constraint, which rules out "users" which have been assigned to **false**. Hence, iff for each clause action there is a user x_i or $\neg x_i$ which can perform it, then the expression is satisfiable. This leads to the general result:

THEOREM 12. *SDSoD is NP-Complete.*

PROOF. The above reduction shows that we can embed any 3SAT problem in a DSoD problem, showing that it is NP-Hard. It is trivial to show that it is NP-Complete: guess an assignment of users to roles and then iterate through the graph and ensure that all roles and *different user* constraints are satisfied. □

5. CONCLUSION

We considered a *task* which consists of *actions* performed by users and having Separation of Duty (SoD) constraints between these actions. We introduced an *approvability graph* system which contains:

- a directed graph whose edges correspond to action specifiers and paths, not necessarily simple, from initial to final nodes corresponding to tasks;
- associated with each edge is a role, or groups of users, which may perform the action; and
- SoD constraints between edges.

The approvability graph allows different tasks to be completed from the same initial action (e.g., approve or deny a loan application), and loops enabling arbitrary number of iterations of an action specifier (e.g., when editing a paper). Because of the loops, an approvability graph can define an unbounded number of tasks.

The approvability graph is the first model which can describe loops.

When the approvability graph is combined with the *user-role assignment*, feasible approvability sequences can be generated, if any exist. We discuss two different ways to generate these sequences, one which only uses past information called *unscheduled approvability* and the other requires both past and future ("lookahead") information called *scheduled approvability*. Although scheduled approvability may "lock in" users to perform future actions (and thus may delay task completion if such users are unavailable), in general, it requires fewer users per role. Clearly, if there are sufficient users per role it is better to use unscheduled approvability due to its increased flexibility.

We then considered sufficient well-formed conditions on the approvability graph to ensure that every task described by it can always complete. We show that for scheduled approvability, it is sufficient not to have a cycle of constraints with exactly one *different user* constraint. For unscheduled approvability there is an additional sufficiency requirement that an action edge in a loop has to have a self *Same user* constraint on it, if it cyclically consumes a user.

Given such well-formed approvability graphs, an algorithm need only consider sets of edges (that is, ignoring repeated transitions of an edge) to determine the minimum number of users per role. For these graphs, optimal schedules are possible but unfortunately, due to the below result, are NP-Hard.

We next considered (simple) algorithms which will provide bounds on the number of users but only require polynomial time. We show these algorithms to be correct even when constraints are between different roles.

Finally, we showed that analysis of even simple DSoD problems are NP-Complete. This is interesting because (1) such analysis is called computationally intractable, and hence

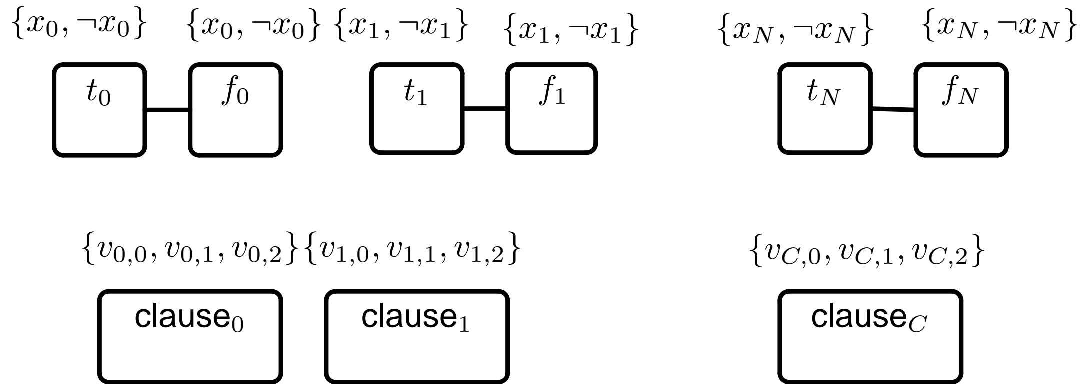

Figure 9: DSoD actions shown as rectangles, and edges between them indicate *different user* constraints. (Note edges are not shown between t_i (or f_i) and clause$_j$, since these are problem specific.) Above each action is the set of users which can perform that action.

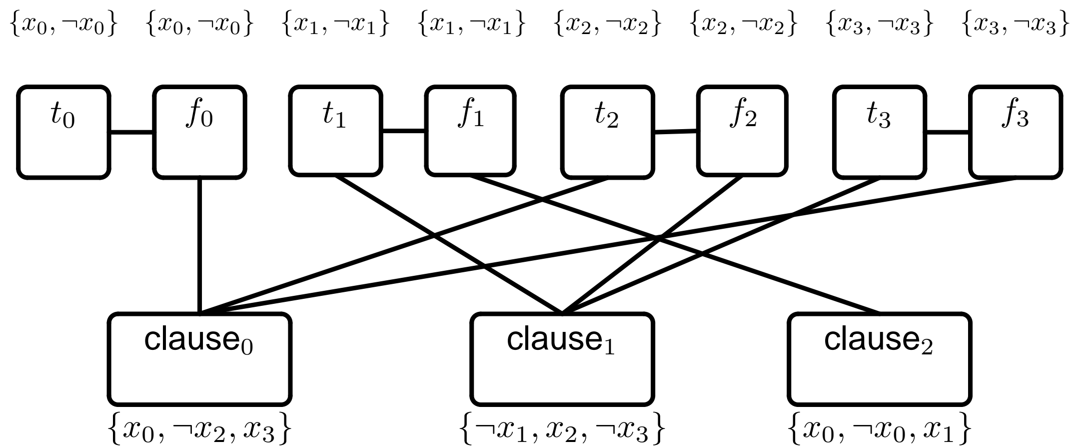

Figure 10: DSoD for Expression (1). The edges all represent *different user* constraints.

may be too expensive to perform and (2) it is the first NP-Completeness result which is inherent in a SoD *problem*.

Techniques for specifying and analyzing SoD have wide applicability, as any organization must tailor their own SoD policies. Such policies depend not only on organization size, but also on the urgency and security needs of a given task. The specification and analysis given here are valuable as they enable the SoD tasks to be clearly specified and ensure that such tasks do not ever get stuck, a condition which needs to be avoided in any practical application.

6. ACKNOWLEDGEMENTS

Special thanks to Robert Sloan who made many suggestions improving the presentation of the theory. In addition, Damian Roqueiro very carefully read the paper and helpful suggestions were made by Prof. V. N. Venkatakrishnan, Mani Radhakrishnan, Ashley Poole, Jorge Hernandez-Herrero, Mike Ter Louw, Hareesh Nagarajan, and the anonymous referees.

7. REFERENCES

[1] G.-J. Ahn and R. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proc. of the ACM Workshop on Role-Based Access Controls (RBAC)*, pages 43–54. ACM Press, 1999.

[2] Atluri, Chun, and Mazzoleni. A chinese wall security model for decentralized workflow systems. In *SIGSAC: 8th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2001.

[3] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):65–104, 1999.

[4] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proc. IEEE Symp. Security and Privacy*, pages 206–214, 1989.

[5] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symp. Security and Privacy*, pages 184–194, 1987.

[6] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 43–50. ACM Press, 2003.

[7] J. Crampton. An algebraic approach to the analysis of constrained workflow systems. In *Proceedings of the Foundations of Computer Security – FCS'04*, volume 31, Turku, Finland, June 2004. TUCS General Publication, Turku Centre for Computer Science.

[8] S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Access control: principles and solutions. *Softw, Pract. Exper*, 33(5):397–421, 2003.

[9] D. Ferraiolo, J. Cugini, and R. Kuhn. Role based access control (RBAC): Features and motivations. In *Annual Computer Security Applications Conference*. IEEE Computer Society Press, 1995.

[10] D. F. Ferraiolo and R. Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, 1992.

[11] S. N. Foley. Separation of duty using high water marks. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW)*, pages 79–88. IEEE, 18-20 June 1991.

[12] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proc. IEEE Symp. Security and Privacy*, pages 172–185, 1998.

[13] M. Hitchens and V. Varadharajan. Tower: A language for role based access control. In *POLICY*, pages 88–106, 2001.

[14] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):158–190, 2001.

[15] J. B. D. Joshi, E. Bertino, B. Shafiq, and A. Ghafoor. Dependencies and separation of duty constraints in GTRBAC. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2003.

[16] P. A. Karger. Implementing commercial data integrity with secure capabilities. In *Proc. IEEE Symp. Security and Privacy*, pages 130–139, 1988.

[17] K. Knorr and H. Weidner. Analyzing separation of duties in petri net workflows. In *Proceedings of the Information Assurance in Computer Networks - Methods, Models, and Architectures for Network Security (MMM-ACNS 2001)*, volume Lecture Notes in Computer Science (LNCS) vol 2052, St. Petersburg, Russia, 2001. Springer Verlag.

[18] D. R. Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proc. of the ACM Workshop on Role-Based Access Controls (RBAC)*, pages 23–30. ACM Press, 1997.

[19] N. Li, Z. Bizri, and M. V. Tripunitara. On mutually-exclusive roles and separation of duty. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 42–51. ACM, 2004.

[20] S. B. Lipner. Non-discretionary controls for commercial applications. In *Proc. IEEE Symp. Security and Privacy*, pages 2–10, 1982.

[21] M. J. Nash and K. R. Poland. Some conundrums concerning separation of duty. In *Proc. IEEE Symp. Security and Privacy*, pages 201–207, 1990.

[22] M. Radhakrishnan and J. A. Solworth. Application security support in the operating system kernel. In *ACM Symposium on InformAtion, Computer and Communications Security (AsiaCCS'06)*, page to appear, Taipei, Taiwan, May 2006.

[23] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[24] R. Sandhu. Transaction control expressions for separation of duties. In *Fourth Aerospace Security Applications Conference*, pages 282–286, 1988.

[25] R. S. Sandhu. Role activation hierarchies. In *ACM Workshop on Role-Based Access Control*, pages 33–40, 1998.

[26] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[27] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW)*, pages 183–194. IEEE, 1997.

[28] J. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 154–163, 2000.