Analyzing Workflows in Business Processes for Obstructions Due to Authorization Policies

Nick Spear, Srekenath Malladi and Sandeep Lakkaraju
College of Business & Information Systems
Dakota State University
Madison, SD - 57042
USA
Email: ngs3738@truman.edu, {malladis,slakkaraju}@pluto.dsu.edu

Abstract

It is well-known that aligning security policies with business objectives is a difficult task. To address this, we present a new approach to analyze work-flow instances for obstructions due to static and dynamic authorization policies. We give a new algorithm that allows organizations to properly assign users to tasks without the policies causing obstructions (e.g. deadlocks). Our work is novel since we consider loops, conditions and parallelism in workflows, through a new concept called “release” events. We illustrate our approach on some real-world workflows in healthcare and financial industries.

1 Introduction

Workflows are used to describe the pattern of tasks to be executed by users to achieve business objectives. A workflow can be implemented in many ways, possibly unboundedly many ways, called instances. An instance has an obstruction if authorization policies make the instance invalid.

For example, consider the workflow in Figure 1.(a). One of its instance in Figure 1.(b) has an obstruction, if either Alice or Bob are not allowed to play roles $r_1$ and $r_2$ respectively. It would also have an obstruction, if the authorization policy states that the tasks $t_1$ and $t_2$ must be executed by the same user.

It is not always possible to determine if a workflow instance has obstructions, with just casual inspection. Automated analysis is very much desirable, since it gives a high degree of confidence about the analysis. Literature concerning analysis of other similar infinite-state models such as cryptographic protocols is replete with security violations that could only be found with automatic and formal approaches [9].

This is the problem we consider in this paper. Our main contribution is an algorithm that answers the following question:

Given a workflow $W$, an instance $\omega$ of it, and a set of authorization policies $E$, is $\omega$ obstruction-free with respect to $E$?

We can use this in turn to answer the question, If $U$ is a set of users playing the roles $R$ of tasks $T$ in $W$, what are the set of substitutions of $U$ to $R$ that are obstruction-free?

These algorithms enable hospitals and banks for instance, to implement workflows without obstructions, while securely enforcing the authorization policies.

\*

\*Worked at DSU as part of the NSF-REU program.
Though there has been considerable work reported in literature on this topic, our work is novel, since we allow workflows to have conditions, loops and parallelism. We justify the newness of our work more in Section 6.1 (related work). We also have a small, fast Java implementation.

Organization In Section 2, we introduce BPMN, a workflow in the healthcare industry, and explain the specification of authorization constraints. In Section 3, we give our main algorithm to determine if a given workflow instance has an obstruction. In Section 4, we give results of our implementation on sample workflows. In Section 5, we analyze the complexity of the algorithm and conclude with a discussion of related and future works.

2 Background

2.1 BPMN

We first introduce BPMN (Business Process Modeling Notation), which is a visualization for business process workflows [7]. The various elements of BPMN are given in Figure 2.

We use five types of elements, defined as follows:

- **Events** that can be start, intermediate or finish, distinguishable by size or number of concentric circles;

- **Tasks and Groups** where the former have an “id” (t1, t2 etc.) and possibly user icons on the upper right corner of the rectangle, while the latter are groupings of tasks;

- **Gateways** are diamond shaped, with multiple input channels. Conditional gateways are plain diamonds with only one output channel that is based on the evaluation of a condition. Parallel gateways have diamonds with a ‘+’ sign inside. They have multiple output channels indicating that control flows on those channels in parallel;

- **Sequences** link tasks together. Dotted arrows are used to associate (different) events, tasks and groups;

- **Extra Events** are used to denote “release events”, that capture “dynamic authorization constraints” (explained in Section 2.2).

A sample workflow specification in a healthcare setting is given in Figure 3. We will later present an analysis of this workflow for obstruction-free instances in Section 4.

The workflow starts when a patient inquires about the diagnostic procedures (vision, hearing etc.) that he has to go through, or tasks that need to be done by the hospital staff, billing staff, and the doctor. The tasks of the workflow are as follows:

- **Task 1 (t1)** is to identify the requirements for the diagnostic procedures to be performed (e.g. tests to evaluate a patient’s health). Here the doctor will identify the tests and treatments for the patient;

- **Task 2 (t2)** states that once the diagnostic procedures are identified in Task 1, the recommended diagnostic procedures will be validated in comparison with the available Medicaid/Medicare regulations. Therefore, t2 is to conduct the procedure validation to check for appropriate diagnosis in comparison with the Medicaid/Medicare regulations. After that, if the procedure validation is correct, the workflow will be carried on to Task 5, otherwise diagnostic procedures will be reviewed and will be recommended based on the Medicaid/Medicare regulations;

- **Task 3 (t3)** is where a patient’s insurance will be evaluated by the hospital staff (e.g. by a receptionist). Based on the patient’s request, his insurance will be evaluated to verify whether his current insurance policy would cover that particular test (for example, an eye or a dental exam). If his insurance covers that particular test, workflow will move on to the next task t4. If his insurance won’t cover the test, the patient may go back and check for a secondary insurance policy which will cover that test;

- **Task 4 (t4)** is to calculate the patient’s deductibles, once the patient’s insurance is validated. This includes the percentage of the total bill that the insurance company would bear (e.g. 85%-15% deductible);

- **Task 5 (t5)** calculates the patient responsibility. This task is a combination of results from t1-t2 flow and t3-t4 flow. The computed quote will be compared to the patient’s deductibles and by doing this, the total amount that is to be billed to the patient will be calculated.

2.2 Authorization constraints

An authorization constraint is put into a system in order to control the access to particular components of the system. This access is based upon an authorization policy that defines the actions that are allowed in the given
Figure 2. BPMN Elements

Figure 3. Healthcare: Work-flow to calculate patient deductibles

Figure 4. Role Based Static Authorizations

system. There are three types of authorization constraints that we consider here, in regards to workflows: static authorizations, dynamic Separation of Duties, and dynamic Binding of Duties. Dynamic constraints remain in effect until the workflow reaches a release point, or release event, a concept first introduced by Basin et al. in [2]. When the workflow encounters a release event, all previous associations between users and tasks in the associated security policy are removed. In our workflows, a release event is represented by a person exiting a door, adapted from [2].

Static authorizations describe a security policy that simply maps each user with a set of permissible tasks. This policy prevents users with inappropriate clearance from executing tasks above their authorization level, demonstrated in Figure 4, showing a visual representation of the user-role-task authorization mappings. We can see that each user is associated with an authorization
level, which is then in turn tied to specific tasks which are allowed to be executed by users at that level. For example, Mary is a doctor, and as such is allowed to perform only task 1 (identifying diagnostic requirements). Mary does not, however, have access to the patient’s insurance records, and cannot then perform tasks 3 or 4. This must be done by a user authorized to execute these tasks, such as Jones or Dennis.

In this paper, static authorizations are not outlined in the workflow models, yet will be defined before-hand as a set of user-task assignments, given as a relation $UT = \{(t,u) | t \in T, u \in U\}$ where $U$ represents the set of users, $T$ represents the set of tasks. In this case, if $(t,u) \in UT$, then we say that the user $u$ has static authorization to perform task $t$. For Figure 4, the user-task assignment relation would be defined as

$$UT = \{(t_1,Tresa), (t_2,Dennis), (t_3,Dennis), (t_4,Dennis), (t_1,Mary), (t_3,Jones), (t_4,Jones)\}$$

Dynamic Separation of Duties (SoD) defines a widely accepted security policy enforced to prevent fraud in the system by ensuring that no single user can access all components in the workflow [6]. This system works by preventing conflicts of interest within the workflow. For example, consider the scenario of a purchase order being placed within a business, shown in Figure 6. The task of placing the order, $t_1$, is in conflict with the task of approving the order, $t_2$, since a user wanting to commit fraud could both place and approve a phony order, while simply pocketing the money. SoD constraints would prevent any user previously associated with executing $t_1$ from executing $t_2$.

We use the notation $s = (T_1, T_2, o_j)$ to represent an SoD constraint, where $T_1$ and $T_2$ are disjoint sets of tasks, and $o_j$ represents a particular release event tied to $s$. In our workflow models, SoD constraints are represented by the “$\neq$” notation. This symbol is associated (via a dotted line) to the task or grouping of tasks that are constrained by an SoD constraint. In the above healthcare workflow example, Figure 3 contains the authorization constraint $s_1 = (t_1, t_2, o_1)$, meaning that any user who executes $t_1$ cannot then execute $t_2$ until release event $o_1$ is reached, removing all associations between users and tasks involved in $s_1$ that have been made up to that point.

It is important to note that the placement of release points is crucial to the meaning of the workflow. For example, in Figure 3, if release point $o_2$ had instead been placed on the ‘yes’ path, the user that performed $t_3$ would then be released from executing $t_4$, which would not force the two tasks to be executed by the same user.

Dynamic Binding of Duties (BoD) is a policy put into place in order to control the number of users executing particular tasks. For example, in a workflow in which sensitive data is required to perform a task $t_1$, and the same sensitive data is required to execute a subsequent (though not necessarily successive) task $t_2$, the authorization policy will bind the tasks in such a way that they must be executed by the same user.

In a similar notation to that used for SoD constraints, we represent a BoD constraint as the tuple $b = (T, o_j)$ such that any user who executes a task in $T$ is then exclusively bound to all tasks in $T$, and no other user is allowed to execute these tasks until release point $o_j$ has been reached. For the workflows in this paper, we represent BoD constraints with an “$=$” symbol, linked to a set of tasks, $T$, by a dotted line.

In Figure 3, we see that tasks $t_3$ and $t_4$ are bound with a BoD constraint, indicating for instance that if Dennis were to execute $t_3$, he must then be the one to execute $t_4$, and no one else. If, however, Dennis were to execute $t_3$, and the patient is not insured, such that the next event is the release event $o_2$, then any user statically authorized to execute $t_3$ can do so without constraint.

3 Our approach and the algorithm

As explained in the Introduction, authorization constraints on a workflow can interfere with the implementation of the workflow by causing potential deadlocks.

The workflow given in the Introduction (Fig 1) is a trivial one. Hence, it is easy to detect an obstruction in an instance of it, simply by manual inspection. However, workflows such as the Healthcare workflow (Fig 3) in Section 2.1 are complicated. Real-world workflows are even more complicated. It is difficult to precisely determine if a given assignment of users to roles results in obstructions in such workflows, since there are many tasks, users, conditions, parallelism and loops. Automated analysis of such workflows is very much necessary to gain assurance that their instances are obstruction-free under a given set of user to role assignments. In this section we will give an algorithm for automated analysis and in Section 4, we will describe its implementation on examples.

We will first define some terms, followed by symbols used and then give the algorithm.

Definition 1. A workflow template is a sequence of events, starting with a start, ending with a finish, and any number of $(t,r)$ events in between where $t$ belongs to the set of tasks and $r$ belongs to the set of roles.

We can have a more detailed definition of a workflow template, splitting it into different sets of events, specify
their ordering, loops etc. using calculi such as CSP [11] or the π-calculus [10], but it is not needed for our algorithm or implementation. Those calculi are useful when expressiveness to describe the details of workflows is needed, but not to check properties of the workflow instances.

Roles are nothing but variables of type “user” that are instantiated with users. The intention is that generating different instances of the workflow is as simple as applying different substitutions of users to roles.

**Definition 2.** A workflow instance is an instantiation of a workflow template by assigning users to roles. Formally, ω is an instance of a workflow template W, if there is a substitution σ of users to tasks that can be applied on W to yield ω, i.e., Wσ = ω.

From now on, we will just call a workflow template simply a workflow.

The symbols we use in our algorithm are given in Table 1.

Some points to note:

- The release events o_j correspond to a unique security policy in E. The notation t_n and u_n will be used to refer to the task and user of the node n respectively. This does not mean that the task number must correspond to the user number, but simply that t_n refers to the task in the node n, and u_n refers to the user in node n;

- This algorithm uses an array L of size |T| where T is the set of all tasks. The purpose of this array is to track which users have performed which tasks in order to check that no security obstructions occur.

---

**Table 1. Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>The given workflow</td>
</tr>
<tr>
<td>ω</td>
<td>An instance of W</td>
</tr>
<tr>
<td>e</td>
<td>An event in an instance ω of workflow W</td>
</tr>
<tr>
<td>T</td>
<td>The set of all tasks in the workflow W</td>
</tr>
<tr>
<td>U</td>
<td>The set of all users of a workflow W</td>
</tr>
<tr>
<td>O</td>
<td>All release events: { o \mid o is a release event of W }</td>
</tr>
<tr>
<td>UT</td>
<td>{ (t, u) \mid user u is statically authorized to execute task t }</td>
</tr>
<tr>
<td>S</td>
<td>SoD policies: { (T_1, T_2, o) \mid T_1 \cup T_2 \subseteq T \land o \in O }</td>
</tr>
<tr>
<td>B</td>
<td>BoD policies: { (T', o) \mid T' \subseteq T \land o \in O }</td>
</tr>
<tr>
<td>E</td>
<td>All authorization policies: UT \cup S \cup B</td>
</tr>
<tr>
<td>N</td>
<td>All nodes of the instance: { n \mid n \in ω }</td>
</tr>
</tbody>
</table>

The array will be set up in such a way that the n_th position in the array will correspond to the n_th task in the workflow. Thus, for example, when a t_n, u_n event is encountered in ω, if no security obstruction is found, the n_th position in L will be set to u_n;

- We write e \in ω if e is an element of the sequence ω.

- The assumption is made that the given instance ω is a valid instance of the workflow, ensuring the existence of a start and finish event, no skipped or extra tasks, and also proper flow (ensuring the workflow is followed in a proper direction).

- The assumption is also made that the policies given are valid polices. For instance, a policy that both binds t_1 and t_2 and also separates the two is not considered. In this case, however, our algorithm would...
simply return ‘NO’ at the first encountered obstruction.

**Algorithm IsObstructionFree**(\(\omega, E\))

**Input:** Workflow instance \(\omega\), Authorization policy \(E\)

**Output:** Yes/No, Is \(\omega\) obstruction-free wrt \(E\)?

1. foreach \(e \in \omega\)
2. if \(e \in N \land e \notin UT\)
3. return NO;
4. foreach \(e \in \omega\)
5. if \(e \in O\)
6. foreach \(s \in S\)
7. if \(o_j \in s\)
8. foreach \(t_n \in s\)
9. \(L[a] = NULL;\)
10. else
11. foreach \(b \in B\)
12. if \(o_j \in b\)
13. foreach \(t_a \in b\)
14. \(L[a] = NULL;\)
15. elseif \(e \in N\)
16. foreach \(s \in S\)
17. if \(t_n \in S\)
18. if \(t_n \in T_1\)
19. foreach \(t_n \in T_2\)
20. if \(L[a] = u_n\)
21. return NO;
22. elseif \(t_n \in T_2\)
23. foreach \(t_n \in T_1\)
24. if \(L[a] = u_n\)
25. return NO;
26. \(L[a] = u_n;\)
27. foreach \(b \in B\)
28. if \(t_n \in b\)
29. foreach \(t_c \in b\)
30. if \(u_n \neq L[c] \land L[c] \neq NULL\)
31. return NO;
32. \(L[a] = u_n;\)
33. return YES;

We will use the collateral evaluation workflow given as a running example in [2] (Figure 5). Using this workflow helps in justifying that our approach can handle workflows with release events that were handled by another approach in [2]. The workflow is used by a financial institution to approve the acquisition of the collateral provided by the borrower. Note that in this workflow:

- Tasks \(t_1\) and \(t_2\) belong to SoD policy \(s_1\);
- Also, tasks \(t_3\) and \(t_4\) are grouped into the BoD policy \(b_1\), meaning that these tasks must be performed by the same user;
- Lastly, all four of these tasks are grouped into the SoD policy \(s_2\). The static authorization policy is also given in [2].

The complete set of constraints is as follows:

\[
\begin{align*}
\omega_1 &= \{t_1.Alice, o_3, t_3.Bob, t_2.Bob, o_1, \\
& \quad t_1.Alice, t_4.Bob, t_2.Claire, t_5.Dave, \\
& \quad t_1.Dave, t_2.Dave, t_3.Dave, t_4.Dave, \\
& \quad t_5.Dave\} \\
\omega_2 &= \{t_1.Alice, o_3, t_3.Bob, t_2.Alice, o_1, \\
& \quad t_1.Bob, t_2.Claire, t_4.Bob, t_5.Claire\}
\end{align*}
\]

We first trace through our algorithm with the obstruction-free instance \(\omega_1\). The first step of the algorithm is to check that each node in the instance is in \(UT\). For \(\omega_1\), this is easily verifiable. After this check is completed, assuming that no obstructions have occurred, the first event reached is the node \(n = t_1.Alice\). There are 9 steps that the algorithm goes through in order to process this event.

1. Check if the event is a release event.
2. Since it is not, we check that this event is a node.
3. It is, so we enumerate through each SoD policy \(s_i\) to check if the task of the node is involved in \(s_i\).
4. We find that \(t_1\) is involved in \(s_1\) and \(s_2\).
5. For \(s_1\), we check which set of tasks that \(t_1\) is a part of. This way, we know which tasks are meant to be separated from \(t_1\).
6. We check that the corresponding entries in \(L\) for each task in \(T_2\) are not set to the user of the node, \(Alice\).
7. Since \(L[2] \neq Alice\), there is no obstruction for this event relative to security policy \(s_1\), and we set \(L[1] = Alice\).
8. We repeat steps 5 - 7 for \( s_2 \), this time comparing ‘Alice’ to \( L[5] = \text{NULL} \).

9. Check if \( t_1 \) is involved in any Bo\( \text{D} \) policy. Since it is not, we move on to the next event in \( \omega \).

The next event encountered is the release event \( o_3 \), putting us at line 5 in the algorithm. The purpose of this portion of code is to determine the authorization policy that the encountered release event is linked to, in order to set each of the tasks involved in that policy to \text{NULL}, effectively performing the ‘reset’ operation of the release event. Note that this only affects the tasks associated with that particular security policy. Since \( o_3 \in b_1 \), and \( b_1 \) contains tasks \( t_3 \) and \( t_4 \), we set the corresponding \( L[3] \) and \( L[4] \) to \text{NULL} and continue on to the next event.

The next event encountered is the node \( n = t_3.Bob \), which follows the same pattern of steps above to result in setting \( L[3] = \text{Bob} \). The difference in the above step with this event is with step 9. We find that \( t_3 \) is indeed involved in a Bo\( \text{D} \) policy, \( b_1 \). The algorithm then follows these steps to process this:

9. Check if \( t_3 \) is involved in a Bo\( \text{D} \) policy.

10. It is, so we enumerate through each Bo\( \text{D} \) policy \( b_i \) to check if the task of the node is involved in \( b_i \).

11. We find that \( t_3 \) is involved in \( b_1 \).

12. We check that the corresponding element in \( L \) to each of the tasks in \( b_1 \) is either the same user of the node we are looking at, Bob in this case, or \text{NULL}, meaning no other user has executed these tasks that are bound together.

13. Since \( L[3] = \text{Bob} \) and \( L[4] = \text{NULL} \), there is no obstruction for this event relative to the security policy \( b_1 \).

14. Since \( t_3 \) is not involved in any other Bo\( \text{D} \) policies, we have no need to repeat steps 11 - 13, and move on to the next event in \( \omega \).

The next event in \( \omega_1 \), \( t_2.Bob \), is a similar operation to previous events. At this point, the non-\text{NULL} elements in \( L \) are \( L[1] = \text{Alice} \), \( L[2] = \text{Bob} \), and \( L[3] = \text{Bob} \). We then encounter the release event \( o_1 \), which is tied to tasks \( t_1 \) and \( t_2 \), so these entries in the array \( L \) are set to \text{NULL}. The three subsequent events result in \( L[1] = \text{Alice} \), \( L[2] = \text{Claire} \), \( L[3] = \text{Bob} \), \( L[4] = \text{Bob} \). The last event, \( t_5.Dave \) is involved in \( s_2 \), and in the set \( T_2 \), so we check each element in \( L \) corresponding to \( t_1 \), \( t_2 \), \( t_3 \), and \( t_4 \) to be sure that \( Dave \) has not executed any of these tasks in \( T_1 \). Since none of the four elements in \( L \) is equal to ‘Dave’, we set \( L[5] = \text{Dave} \) and return ‘YES’, indicating that this instance is obstruction-free.

For \( \omega_2 \), the first three events are the same from \( \omega_1 \), so using the steps above, after the third event in \( \omega_2 \), the non-\text{NULL} elements in \( L \) are \( L[1] = \text{Alice} \) and \( L[3] = \text{Bob} \). The next event in \( \omega_2 \) is \( t_2.Alice \). We see that \( t_2 \) is involved in the security policy \( s_1 \) and \( s_2 \). We first check for violations in \( s_2 \). Since \( t_2 \in T_2 \subset s_1 \), we check the corresponding element in \( L \) of each task in \( T_1 \) for a conflicting assignment of tasks to users. In this case, \( L[1] = \text{Alice} \), which makes the event \( t_2.Alice \) a violation, since \( t_1 \) and \( t_2 \) are not authorized to be executed by the same user. Thus, this instance is not obstruction free, and the algorithm returns ‘NO’.

4 Implementation

Figure 6 is an example workflow for a business purchase. In this example, Task 1 is to first send the request to the proper management, and at the same time to execute Task 4, being to request the funds from the budget. Task 2 is to have the request from Task 1 approved and Task 5 verifies that the funds are in the budget. When all of these tasks have been completed, Task 3 is to actually make the purchase. Note that Tasks 2 and 3 are grouped, and connected by Bo\( \text{D} \) policy \( b_1 \) and release event \( o_1 \), indicating that Tasks 1 and 4 must be done by the same user, and cannot be performed by another user until release event \( o_1 \) is reached. The next group of tasks, 2 and 3, are linked to this first group of tasks \( (t_1 \) and \( t_4) \) by Bo\( \text{D} \) policy \( s_1 \) and release event \( o_2 \). This policy ensures that the user that is bound to executing tasks 1 and 4 is then prohibited from executing tasks 2 and 3. This second group is also a part of the Bo\( \text{D} \) policy \( b_2 \) with release event \( o_3 \). The complete set of constraints is as follows:

\[
\begin{align*}
T_1 &= \{ t_1, t_4 \}, T_2 = \{ t_2, t_3 \}, o_1, o_2 \\
S_1 &= \{ \{ t_1, t_4 \}, o_1 \} \\
S_2 &= \{ \{ t_2, t_3 \}, o_3 \} \\
UT &= \left\{ \begin{array}{l}
(t_1.Claire, t_4.Claire, t_5.Claire), (t_1.Dave, t_2.Dave, t_3.Dave), \\
(t_4.Dave, t_5.Dave) \\
\end{array} \right\}
\end{align*}
\]

Our first example implementation of our algorithm is using this workflow.

Consider the following instances of the workflow:

\[
\omega_1 = \langle t_1.Alice, t_2.Bob, o_2, t_1.Alice, t_2.Claire, t_3.Claire, t_4.Alice, t_5.Alice \rangle. \\
\]
The first instance contains three static authorization obstructions, which would be found in line 2 of the algorithm since the events $t_5.Alice$, $t_2.Claire$, and $t_3.Claire$ are nodes that are not members of the set $UT$. If static authorization obstructions do exist in a given instance, these obstructions are caught early in the algorithm, since these user-task assignments are not considered valid, and therefore render the remaining events in the instance irrelevant.

$\omega_2$ contains an SoD obstruction, which the algorithm finds in line 24 since $t_2 \in T_2$ and for $t_1 \in T_1$, $L[1] = Bob = u_1$, meaning that Bob had already executed a task in a conflicting set of tasks. Though the algorithm will have returned ‘NO’ already, this instance also contains a BoD obstruction with the event $t_4.Dave$, since $t_1$ was executed by Bob, and these two tasks are meant to be bound.

Another example implementation of our algorithm is given using the healthcare workflow shown in Figure 3, using the following example instances:

$\omega_1 = \langle t_3.Dennis, o_2, t_1.Mary, t_2.Dennis, t_3.Jones, t_4.Dennis, t_5.Jones \rangle$.

$\omega_2 = \langle t_1.Tresa, t_2.Dennis, o_1, t_3.Dennis, t_1.Mary, t_2.Dennis, t_4.Dennis, t_5.Jones \rangle$.


$\omega_1$ contains a BoD obstruction with the event $t_4.Dennis$. The algorithm will take several steps to catch this obstruction, first noting that, in the previous event, since no obstruction will occur with the execution of $t_3$ by Jones, the corresponding entry in $L$ will be set to $L[3] = Jones$. However, when Dennis attempts to execute $t_4$, our algorithm will check the BoD policies that $t_4$ is included in to be sure that the sets of tasks in each policy are being executed by the same user. In this case, if a user other than Dennis is listed in $L$ as having performed $t_3$, then this is an obstruction. We see in $L$ that Jones executed $t_3$, and therefore the BoD constraints are violated in this instance $\omega_1$. Thus, the algorithm returns ‘NO’, and the obstruction is correctly identified.

In the second instance, $\omega_2$, there are no obstructions. This is properly identified by our algorithm after lines 1 - 3 find that each node in $\omega_2$ is in $UT$, indicating that no static authorization obstructions occur, after lines 4 - 14 ‘reset’ the values in $L$ for the tasks corresponding to the event $o_1$, after lines 16 - 26 verify that each event in $\omega_2$ involved in an SoD policy is not in conflict with any previous events, tracked in the array $L$, indicating no SoD obstructions, and finally after lines 27 - 32 verify that each node in $\omega_2$ involved in a BoD policy is not in conflict with any previous tasks.

We use the last example instance $\omega_3$ as an example of the ability of our algorithm to handle the loops and conditions of a workflow. This instance follows all release events in the workflow, and tests each possible value of each condition. The algorithm correctly returns that no obstructions are found, properly handling the operations of each release event. The actual output of the program is given below:

C:\Program Files\Java\jdk1.6.0_26\bin>java IsObstructionFree
Checking for static authorization
violations ...
No static obstructions found.
Checking for dynamic obstructions ...

After displaying similar statements for other events, the trace then ends with:

For event t5.Jones, checking for policies involving t5. No dynamic obstructions found.
No obstruction found for this instance!
C:\Program Files\Java\jdk1.6.0_26\bin>

When we change $\omega_3$ slightly such that in the last three tasks, $t_3$ and $t_4$ are now performed by Dennis, the algorithm still finds the instance to be obstruction-free, again showing the ability to handle loops and conditions. If these release events were not handled properly, the event $t_3.Jones$ would have caused one of these changes in events to violate BoD policy $b_1$, however, because of our algorithm’s novel ability to handle these events, this instance is correctly found to be obstruction-free.

5 Complexity analysis

The best-case scenario for the algorithm is a workflow with no release events. i.e., there are no dynamic policies (SoD, BoD) in the authorization policies, only static, if any. Instance $\omega_2$ of the purchase-approval workflow is such a case. Obviously, in this case, the complexity is $O(n)$ where $n$ is the number of events, since only the first loop between lines 1 to 3 is executed and the second loop on line 4 is executed as well, but without any statements inside it.

The worst-case scenario is when every release event that is tied to a BoD policy is encountered at least once. Instance $\omega_3$ of the health-care workflow is such a case. In this case, lines 4 through 14 are executed for all the events, resulting in a complexity of $O(n^5)$.

The average-case is when there are release events in the workflow that are not encountered in the instance. In this case, the complexity is $O(n^3)$. Instance $\omega_1$ of the Purchase Approval workflow is one such.

6 Conclusion

In this paper, we have given an algorithm to determine if a workflow instance is obstruction-free with respect to a given set of authorization policies and tested it on some common workflows in the financial and healthcare industries.

The input to the algorithm is rather simple: In addition to the workflow and the policies, it is just a set of user/role substitutions. This allows even naive users of our implementation to easily analyze their workflow instances. The implementation itself is just three pages of Java code and it has successfully output all the results presented in this paper.

6.1 Related work

Our algorithm decides satisfiability for bounded users in the sense of [3, 16], which states that a policy is satisfiable if there exists an assignment of users to tasks that does not violate the policy. It also decides if a workflow is “sound” in the sense of Van der Aalst [5] who calls sound workflows as those that do not have dead transitions or deadlock before completing their final tasks.

Most past works on checking authorization policies did not consider conditions, loops and parallelism in workflows including [13, 1, 3, 14, 16, 8]. Solworth [15] does consider them, but in that model, constraints in the presence of loops are restricted such that the first task must always be executed by the same person.

The only other work that considers conditions, loops and parallelism is the recent magnificent work of Basin et al. [2]. Some points to note in comparison are:

- They give interesting theoretical results basing their framework on CSP [12], while our work is applied: We have a practical implementation of an algorithm based on their concepts, and we have applied it to some commonly used workflows in the real-world;
- Our algorithm was based purely on substitutions and nodes in the workflow. We do not claim that our approach is better or worse than [2]. However, like many constraint-satisfaction approaches, it was easier to develop a practical implementation;
- They demonstrate that finding out whether a workflow can have an obstruction-free instance with respect to a set of policies is decidable but NP-Hard, by reducing the problem to graph-coloring. Ideally, one should run their algorithm to first to check if a set of policies are enforceable, and then use our algorithm to find if a particular instance is obstruction-free;
The concept of “release points” used in our algorithm was first introduced by Basin et al, extending previous work on security-annotated graphical workflow models [4].

6.2 Future work

An immediate extension of this work is to determine all the set of substitutions of users to roles for a given set of users. To accomplish this, we can enumerate all possible substitutions of users to roles and use our main algorithm for each substitution. The complexity of this would be obviously proportional to the number of users. An algorithm given in [2] to solve the same problem has polynomial complexity when the number of users is large and the static authorizations are well-distributed. It is possible that our approach would have similar complexity under similar conditions. A detailed complexity analysis is our task on hand.

We would like to improve our algorithm by adding the ability to check for inconsistent policies, such as a policy stating that a task should not be executed by the same user, while another states that it should be executed by the same user. We would also like to allow any number of tasks in our implementation (the current version allows a maximum of ten).

Acknowledgment. We would like to thank Samuel Burri (ETH Zurich) for his help and suggestions. This work was supported by NSF-REU program and the National Center for Protection of Financial Infrastructure at DSU.

References


