Automated Synthesis of Run-time Monitors to Enforce Authorization Policies in Business Processes*

Clara Bertolissi
Fondazione Bruno Kessler
University of Marseille
clara.bertolissi@lif.univ-mrs.fr

Daniel Ricardo dos Santos
Fondazione Bruno Kessler
SAP Labs France
University of Trento
dossantos@fbk.eu

Silvio Ranise
Fondazione Bruno Kessler
ranise@fbk.eu

ABSTRACT

Run-time monitors are crucial to the development of security-aware workflow management systems, which need to mediate access to their resources by enforcing authorization policies and constraints, such as Separation of Duty. In this paper, we introduce a precise technique to synthesize run-time monitors capable of ensuring the successful termination of workflows while enforcing authorization policies and constraints. An extensive experimental evaluation shows the scalability of our technique on the important class of hierarchically specified security-sensitive workflows with several hundreds of tasks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Run-time Enforcement, Workflow Satisfiability

1. INTRODUCTION

It is common that workflow management systems support the execution of business processes. A workflow specifies a collection of tasks and the causal relationships between them. The execution of tasks is initiated by humans or software agents executing on their behalf. Security-related dependencies are specified as additional constraints on the execution of the various tasks. In an organization, a workflow task is executed by a user who should be entitled to do so; e.g., the teller of a bank may create a loan request whereas only a manager may accept it. Additional authorization constraints are usually imposed on task execution, such as Separation of Duty (SoD) or Bound of Duty (BoD) whereby two distinct users or the same user, respectively, must execute two tasks. Below, following [2], we call "security-sensitive" this kind of workflows.

The *Workflow Satisfiability Problem* (WSP) consists of checking if there exists an assignment of users to tasks such that a security-sensitive workflow successfully terminates while satisfying all authorization constraints. Such a problem has been studied in several papers; see, e.g., [27, 20]. The run-time version of the WSP consists of answering sequences of user requests at execution time and ensuring successful termination together with the satisfaction of authorization constraints. This problem has received less attention and only an approximate solution is available [3, 4].

The **main contribution** of this paper is an automated technique to synthesize run-time monitors capable of ensuring the successful termination of workflows while enforcing authorization policies and SoD/BoD constraints, thus solving the run-time version of the WSP. Changes in the authorization policies can be accommodated without re-running from scratch the approach. Section 2 illustrates the main steps underlying the technique on a simple example. Section 3 gives full details and states several theorems guaranteeing the correctness of the approach. **Another contribution** of the paper is an extensive experimental evaluation of the proposed technique on hierarchically structured workflows, i.e. complex workflows that can be decomposed in subflows. Section 4 describes a prototype implementation of the technique and how the structure of hierarchic specifications can be exploited to make our approach scale to large workflow systems containing hundreds of tasks. Section 4.1 discusses the performances of our technique on two workflow systems inspired by realistic use-cases while Section 4.2 studies its scalability on synthetic benchmarks inspired by those in [12] containing up to 500 tasks. Our findings clearly show the scalability of the proposed technique on hierarchic workflows. Section 5 discusses related work and Section 6 concludes the paper by giving hints to future work.

2. A TRIP REQUEST EXAMPLE

We describe our approach to synthesize run-time monitors for security-sensitive workflows on a trip request process. The workflow is composed of five tasks—each one indicated by a box labeled by Trip request ($t1$), Car rental

---

*This work was partly supported by the EU under grant FP7-PEOPLE-SECENTIS and the RESTATE Programme, co-funded by the European Union under the FP7 COFUND Marie Curie Action—Grant agreement no. 267224.

ASIA CCS'15, April 14–17, 2015, Singapore.

ACM 978-1-4503-3245-3/15/04 $15.00.
http://dx.doi.org/10.1145/2714576.2714633.

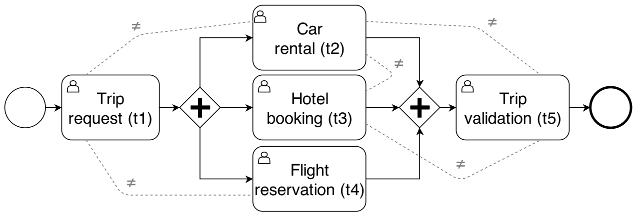

Figure 1: Workflow in extended BPM notation

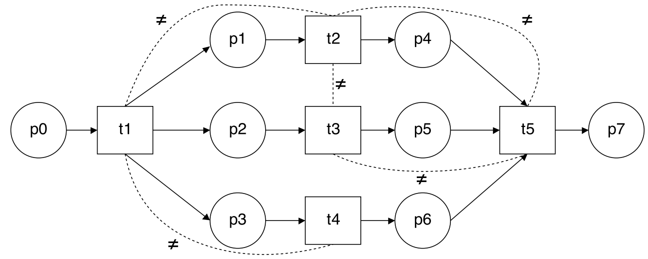

Figure 2: Workflow as an extended Petri net

($t2$), Hotel booking ($t3$), Flight reservation ($t4$), and Trip validation ($t5$)—whose execution is constrained as follows (cf. solid arrows and diamonds labeled with $+$): $t1$ must be executed first, then $t2$, $t3$ and $t4$ can be executed in any order, and when all have been performed, $t5$ can be executed, thereby terminating the workflow. Additionally, each task is executed under the responsibility of a user (indicated by the small icon inside the boxes corresponding to the various tasks) who has the right to execute it according to some access control policy—not shown in Figure 1—and the five authorization constraints depicted as dashed lines labeled by the symbol $\neq$ for Separation of Duty (SoD). So, for example, the authorization constraint connecting the boxes of $t1$ and $t2$ requires the user executing $t2$ to be distinct from the one that has executed $t1$, i.e. the user who requests the trip cannot also rent a car.

Our goal is to synthesize a run-time monitor, capable of ensuring that all execution and authorization constraints are satisfied. Our approach is organized in two phases: off-line and on-line.

Off-line. We first construct a symbolic transition system S whose executions correspond to those of the security-sensitive workflow. Then, we use a symbolic model checker to explore all possible terminating executions of the workflow which satisfy both the causality and the authorization constraints. We assume the model checker to be able to return a symbolic representation R of the set of all states, called reachable, encountered during the exploration of the terminating executions of S. We use particular classes of formulae in first-order logic to be the symbolic representations of S and R.

On-line. We derive a Datalog program M from the formulae R, representing the set of states reachable in the terminating executions of S and the policy P specifying which user can perform which task. The Datalog program M derived in this way is the monitor capable of guaranteeing that any request of a user to execute a task is permitted by P, satisfies the authorization constraints (such as SoD), and the workflow can terminate its execution.

We illustrate the two phases on the security-sensitive workflow in Figure 1.

2.1 Off-line phase

First of all, we build the symbolic transition system S in two steps: (i) we adopt the standard approach (see, e.g., [24]) of using (extensions of) Petri nets [19] to formalize the semantics of workflows and (ii) we adapt the well-known translation of Petri nets to symbolic transition systems (see, e.g., [21]) to the class of extended Petri nets used in this paper.

Figure 2 shows the extended Petri net that can be automatically derived from the BPM notation of Figure 1. Tasks are modeled as transitions or events (the boxes in the figure) whereas places (the circles in the figure) encode their enabling conditions. At the beginning, there will be just one token in place $p0$ which enables the execution of transition $t1$. This corresponds to the execution constraint that task $t1$ must be performed before all the others. The execution of $t1$ removes the token in $p0$ and puts a token in $p1$, another in $p2$, and yet another in $p3$; this enables the execution of $t2$, $t3$, and $t4$. Indeed, this corresponds to the causality constraint that $t2$, $t3$, and $t4$ can be executed in any order after $t1$ and before $t5$. In fact, the executions of $t2$, $t3$, and $t4$ remove the tokens in $p1$, $p2$, $p3$ and put a token in $p4$, $p5$, and $p6$ which, in turn, enables the execution of $t5$. This removes the token in $p4$, $p5$, $p6$ and put a token in $p7$ which enables no more transitions. This corresponds to the fact that $t5$ is the last task to be executed. The fact that there is at most one token per place is an invariant of the Petri net. This allows us to symbolically represent the net as follows: we introduce a Boolean variable per place (named as the places in Figure 2) together with a Boolean variable representing the fact that a task has already been executed (denoted by d_t and if assigned to true implies that task t has been executed). So, for instance, the enabling condition for the execution constraint on task $t1$ can be expressed as $p0 \wedge \neg d_{t1}$ meaning that the token is in place $p0$ and transition $t1$ has not yet been executed. The effect of executing transition $t1$ is to assign F(alse) to $p0$ and T(rue) to $p1$, $p2$, $p3$, and d_{t1}; in symbols, we write $p0, p1, p2, p3, d_{t1} := F, T, T, T, T$. The other transitions are modeled similarly.

Besides the constraints on the execution of tasks, Figure 2 shows also the same authorization constraints of Figure 1. These are obtained by taking into consideration both the access control policy P granting or denying users the right

Table 1: Workflow as symbolic transition system

event	enabled		action	
	CF	Auth	CF	Auth
$t1(u)$	$p0 \wedge \neg d_{t1}$	$a_{t1}(u)$	$p0, p1, p2, p3, d_{t1}$ $:= F, T, T, T, T$	$h_{t1}(u)$ $:= T$
$t2(u)$	$p1 \wedge \neg d_{t2}$	$a_{t2}(u) \wedge \neg\, h_{t3}(u)$ $\wedge \neg\, h_{t1}(u)$	$p1, p4, d_{t2}$ $:= F, T, T$	$h_{t2}(u)$ $:= T$
$t3(u)$	$p2 \wedge \neg d_{t3}$	$a_{t3}(u) \wedge \neg h_{t2}(u)$	$p2, p5, d_{t3}$ $:= F, T, T$	$h_{t3}(u)$ $:= T$
$t4(u)$	$p3 \wedge \neg d_{t4}$	$a_{t4}(u) \wedge \neg h_{t1}(u)$	$p3, p6, d_{t4}$ $:= F, T, T$	$h_{t4}(u)$ $:= T$
$t5(u)$	$p4 \wedge p5 \wedge$ $p6 \wedge \neg d_{t5}$	$a_{t5}(u) \wedge \neg\, h_{t3}(u)$ $\wedge\ \neg\, h_{t2}(u)$	$p4, p5, p6, p7, d_{t5}$ $:= F, F, F, T, T$	$h_{t5}(u)$ $:= T$

to execute tasks and the SoD constraints between pairs of tasks. To formalize these, we introduce two functions a_t and h_t from users to Boolean, for each task t, which are such that $a_t(u)$ is true iff u has the right to execute t according to the policy P and $h_t(u)$ is true iff u has executed task t. Notice that a_t is a function that behaves as an abstract interface to the policy P whereas h_t is a function that evolves over time and keeps track of which users have executed which tasks. For instance, the enabling condition for the authorization constraint on task $t1$ is simply $a_{t1}(u)$, i.e. it is required that the user u has the right to execute $t1$, and the effect of its execution is to record that u has executed $t1$, i.e. $h_{t1}(u) := T$ (notice that this assignment leaves unchanged the value returned by h_{t1} for any user u' distinct from u). Notice that it is useless to take into account the SoD constraints between $t1$ and $t2$, $t4$ when executing $t1$ since $t2$ and $t4$ will always be executed afterwards. As another example, let us consider the enabling condition for the authorization constraint on $t2$: besides requiring that u has the right to execute $t2$ (i.e. $a_{t2}(u)$), we also need to require the SoD constraints with $t1$ and $t3$ (not that with $t5$ since this will be executed afterwards), i.e. that u has executed neither $t1$ (i.e. $\neg h_{t1}(u)$) nor $t3$ (i.e. $\neg h_{t3}(u)$). The authorization constraints on the other tasks are modeled in a similar way.

Table 1 shows the formalization of all transitions in the extended Petri net of Figure 2. The first column reports the name of the transition together with the fact that it is dependent on the user u taking the responsibility of its execution. The second column shows the enabling condition divided in two parts: CF, pertaining to the execution constraints, and Auth, to the authorization constraints. The third and last column list the effects of the execution of the transition again divided in two parts: CF, for the workflow, and Auth, for the authorization.

The initial state of the security-sensitive workflow is described by the *initial* formula

$$p0 \wedge \bigwedge_{i=1,\ldots,7} \neg pi \wedge \bigwedge_{i=1,\ldots,5} \neg d_{ti} \wedge \bigwedge_{i=1,\ldots,5} \forall u. \neg h_{ti}(u) \qquad (1)$$

saying that there is just one token in $p0$, no task has been executed, and indeed no user has yet executed any of the tasks, whereas a state of a terminating execution of the workflow by the *goal* or *final* formula

$$p7 \wedge \bigwedge_{i=0,\ldots,6} \neg pi \wedge \bigwedge_{i=1,\ldots,5} d_{ti} \qquad (2)$$

saying that there is just one token in $p7$ and all the tasks have been executed.

Formally, the way in which we specify the transition systems corresponding to security-sensitive workflows can be seen as an extended version of the assertional framework proposed in [22]. We emphasize that obtaining, from the extended BPM notation of Figure 1, the symbolic representation S of the initial and goal formulae with that of the transitions in Table 1 is a fully automated process.

Exploring the search space. After obtaining the symbolic representation of the initial and goal states together with the transitions of the security-sensitive workflow, we invoke a symbolic model checker in order to compute the symbolic representation R of the set of (reachable) states visited while executing all possible sequences of transitions leading from an initial to a goal state. A crucial assumption of our approach is that the model checker is able to compute

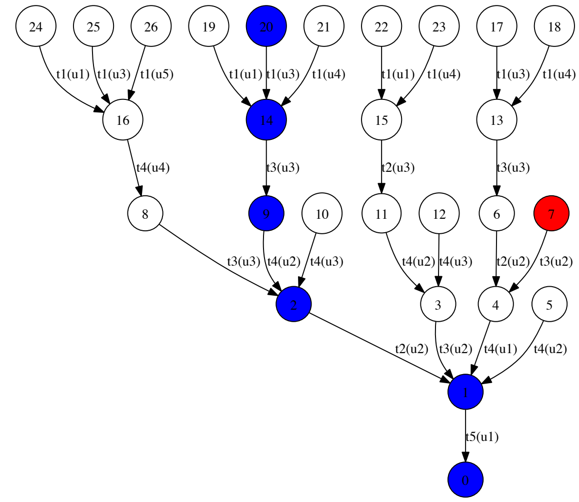

Figure 3: Graph-like representation of the set of reachable states for the workflow in Figure 1

R for any finite number of users. By doing this, the interface functions a_t's can be instantiated with any policy P, i.e. containing any number of users. As a consequence, changes in the authorization policy do not imply to re-run the off-line phase. In summary, our goal is to compute a parametric—in the number n of users—representation of the set of states visited while executing all possible terminating sequences of transitions. From now on, we write R_n to emphasize this fact.

Although the computation of R_n seems to be a daunting task, there exist techniques available in the literature about parameterized model checking (see the seminal paper [1]) that allow us to do this. Among those available, we have chosen the Model Checking Modulo Theories approach proposed in [16] for it uses first-order formulae as the symbolic representation of transition systems and the availability of tools, such as MCMT [17], which are capable of returning the set of reachable states as a first-order formula.

For instance, Figure 3 shows a graph-like representation of the formula R_n for the security-sensitive workflow described by the symbolic transition system derived from Figure 1. Each node is associated to a first-order formula: node 0 (bottom of the figure) is labeled by the goal formula (2), nodes 17–26 (top of the figure) are labeled by formulae describing sets of states that have non-empty intersection with the set of initial states characterized by the initial formula (1), all other nodes (namely, those from 1 to 16) are labeled with formulae describing sets of states that are visited by executing transitions (labeling the arcs of the graph) belonging to a terminating sequence of executions of the workflow. For instance, node 1 is labeled by the formula

$$\begin{aligned} \neg p0 \wedge \neg p1 \wedge \neg p2 \wedge \neg p3 \wedge p4 \wedge p5 \wedge p6 \quad &\wedge \\ d_{t1} \wedge d_{t2} \wedge d_{t3} \wedge d_{t4} \wedge \neg d_{t5} \quad &\wedge \\ (a_{t5}(u1) \wedge \neg h_{t2}(u1) \wedge \neg h_{t3}(u1)) \quad & \end{aligned}$$

describing the set of states from which it is possible to reach a goal state when some user $u1$ takes the responsibility to execute task $t5$. The first two lines in the formula above require that there is a token in places $p4$, $p5$, $p6$ (thereby enabling transition $t5$), tasks $t1$, $t2$, $t3$, $t4$ have been executed, and $t5$ has not yet been performed. The last line requires that user $u1$ has the right to execute $t5$ and that he/she has performed neither $t2$ nor $t3$ (because of the SoD constraints between $t5$ and $t2$ or $t3$). In general, let us consider an arc $\nu \xrightarrow{t(u)} \nu'$ in the graph of Figure 3: the formula labeling node ν describes the set of states from which it is possible to reach the set of states described by the formula labeling node ν' when user u executes task t. Thus, the paths starting from one of the nodes 17–26 (labeled by formulae representing states with non-empty intersection with the set of initial states) and ending in node 0 (labeled by the goal formula) describe all possible terminating executions of the workflow in Figure 1 (although nodes 5, 7, 10 and 12 seem to be exceptions, this is not the case: explaining their role requires a more precise description of how the graph is built and will be discussed in the next section). For instance, the sequence of blue nodes describes the terminating sequence $t1, t3, t4, t2, t5$ of task executions by the users $u3$, $u3$, $u2$, $u2$, and $u1$, respectively. It is easy to check that this sequence satisfies both the execution and the authorization constraints required by the workflow in BPM notation of Figure 1. In fact, $t1$ is executed first, $t5$ is executed last, and $t2, t3, t4$ are executed in between; there are three *distinct* users $u1, u2, u3$ that can execute the five tasks without violating any of the SoD constraints. By considering all possible paths in the graph of Figure 3, it is easy to see that there should be at least three distinct users to be able to terminate the security-sensitive workflow in Figure 1. From what we said above, the formula R_n representing the set of states visited during terminating sequences of task executions of the security-sensitive workflow in Figure 1 can be obtained by taking the disjunction of the formulae labeling the nodes in the graph of Figure 3 except for the one labeling node 0 since, by construction, no task is enabled in the set of states represented by that formula. Let r_ν be the formula labeling node ν, then

$$R_n := \bigvee_{\nu \in N} r_\nu \qquad (3)$$

where N is the set of nodes in the graph (in the case of Figure 3, we have $N = \{1, ..., 26\}$).

2.2 On-line phase

Once MCMT has returned the first-order formula R_n describing the set of states visited during any terminating executions for a (finite but unknown) number n of users, we can derive a Datalog [10] program which constitutes the run-time monitor of the security-sensitive workflow formalized by the symbolic transition system used to compute R_n. Then, we can add the specification of the interface functions a_{t1}, ..., a_{t5} for a given value of n.

We have chosen Datalog as the programming paradigm in which to encode monitors for three main reasons. First, it is well-known [18] that a wide variety of access control policies can be easily expressed in Datalog. Second, Datalog permits efficient computations: the class of Datalog programs resulting from translating formulae R_n permits to answer queries in *LogSpace* (see below for more details). Third, it is possible to further translate the class of Datalog programs we produce to SQL statements so that run-time monitors can be easily implemented as database-backed applications. In the rest of this section, we describe how it is possible to derive Datalog programs from formulae describing the set of reachable states computed by the model checker and then how to add the definitions of the interface functions a_{t1}, ..., a_{t5}.

From R_n to Datalog. Recall the form (3) of R_n. It is not difficult to see that each r_ν can be seen as the conjunction of a formula r_ν^{CF} containing the Boolean functions $p0, ..., p7$ for places and $d_{t1}, ..., d_{t5}$ keeping track of task execution with a formula r_ν^{Auth} of the form

$$a_t(u0) \wedge \rho_\nu^{\mathrm{Auth}}(u0, u1, ..., uk)$$

where $u0$ identifies the user taking the responsibility to execute task t, ρ_ν^{Auth} is a formula containing the variables $u0, u1, ..., uk$, the interface functions $a_{t1}, ..., a_{t5}$, the history functions $h_{t1}, ..., h_{t5}$, and all disequalities between pairwise distinct variables from $u0, u1, ..., uk$ (indeed, if there are no variables, there is no need to add such disequalities). For instance, formula r_1 labeling node 1 in Figure 3 is $r_1^{\mathrm{CF}} \wedge r_1^{\mathrm{Auth}}$ where

$$\begin{aligned} r_1^{\mathrm{CF}} &:= \neg p0 \wedge \neg p1 \wedge \neg p2 \wedge \neg p3 \wedge p4 \wedge p5 \wedge p6 \wedge \\ &\quad\; d_{t1} \wedge d_{t2} \wedge d_{t3} \wedge d_{t4} \wedge \neg d_{t5} \\ r_1^{\mathrm{Auth}} &:= \rho_1^{\mathrm{Auth}}(u1) \\ \rho_\nu^{\mathrm{Auth}}(u1) &:= a_{t5}(u1) \wedge \neg h_{t2}(u1) \wedge \neg h_{t3}(u1) \end{aligned}$$

with $u0$ renamed to $u1$.

In general, each r_ν in the expression (3) for the formula R_n can be written as

$$r_\nu^{\mathrm{CF}} \wedge a_t(u0) \wedge \rho_\nu^{\mathrm{Auth}}(u0, u1, ..., uk) \qquad (4)$$

and describes a set of states in which user $u0$ executes task t while guaranteeing that the workflow will terminate since ν is one of the nodes in the graph computed by the model checker while generating all terminating sequences of tasks. In other words, (4) implies that $u0$ can execute task t or, equivalently written as a Datalog clause: $\mathit{can\_do}(u0, t) \leftarrow$ (4), where *can_do* is a Boolean function returning true iff a user (first argument) is entitled to execute a task (second argument) while all execution and authorization constraints are satisfied and the workflow can terminate. Notice that $\mathit{can\_do}(u0, t) \leftarrow (4)$ is a Datalog clause. So, we generate the following Datalog clauses

$$\mathit{can\_do}(u0, t) \leftarrow r_\nu^{\mathrm{CF}} \wedge a_t(u0) \wedge \rho_\nu^{\mathrm{Auth}}(u0, u1, ..., uk) \qquad (5)$$

for each $\nu \in N$. In the following, let D_n be the Datalog program composed of all the clauses of the form (5). For instance, the Datalog clause corresponding to node 1 is

$$\begin{aligned} \mathit{can\_do}(u1, t5) \;\leftarrow\;& \neg p0 \wedge \neg p1 \wedge \neg p2 \wedge \neg p3 \wedge p4 \wedge p5 \wedge p6 \wedge \\ & d_{t1} \wedge d_{t2} \wedge d_{t3} \wedge d_{t4} \wedge \neg d_{t5} \wedge \\ & a_{t5}(u1) \wedge \neg h_{t2}(u1) \wedge \neg h_{t3}(u1)\,. \end{aligned}$$

It is not difficult to show that $\mathit{can\_do}(u, t)$ iff there exists a disjunct of the form (4) in R_n for a given number n of users. Finally, observe that clauses of the form (5) contain negations but are non-recursive.

Specifying the policy P. We are left with the problem of specifying the access control policy P for a given number n of users. As already observed above, there should be at least three distinct users in the system to be able to terminate the execution of the workflow in Figure 1. So, to illustrate, let $U = \{a, b, c\}$ be the set of users and use the RBAC model to express the policy. This means that we have a set $R = \{r_1, r_2, r_3\}$ of roles which are indirections between users and (permissions to execute) tasks. Let $UA = \{(a, r1), (a, r2), (a, r3), (b, r2), (b, r3), (c, r2)\}$ be the user-role assignments and $TA = \{(r_3, t1), (r_2, t2), (r_2, t3), (r_1, t4), (r_2, t5)\}$ be the role-task assignment. Then, a user u can execute task t iff there exists a role r such that $(u, r) \in UA$ and $(r, t) \in TA$. This can be formalized by the following Datalog clauses:

$$
\begin{array}{l}
ua(a, r1)\ ua(a, r2)\ ua(a, r3)\ ua(b, r2)\ ua(b, r3)\ ua(c, r2) \\
pa(r_3, t1)\ pa(r_2, t2)\ pa(r_2, t3)\ pa(r_1, t4)\ pa(r_2, t5) \\
a_t(u) \ \leftarrow\ ua(u, r) \wedge pa(r, t) \text{ for each } t \in \{t1, ..., t5\}
\end{array}
$$

and denoted by D_P. By taking the union of the clauses of D_n and D_P, we build a Datalog program $M_{n=3}$ allowing us to monitor the security-sensitive workflow of Figure 1. I.e. $M_{n=3}$ is capable of answering queries of the form $can\_do(u, t)$ in such a way that all execution and authorization constraints are satisfied and the workflow execution terminates. An example of a run of the monitor is in Table 2, where each line represents a state of the system; columns CF and Auth describe the values of the variables in that state ("Token in" shows which places have a token and the various h_{ti} hold the name of the user who executed task t_i); $can\_do(u, t)$ represents user u requesting to execute task t and 'Resp' is the corresponding response returned by the monitor (grant or deny the request). The execution in the table shows two denied requests, one in line 0 and one in line 2. In line 0, user a requests to execute task $t1$ but this is not possible since a is the only user authorized to execute $t4$, and if a executes $t1$, he/she will not be allowed to execute $t4$ because of the SoD constraint between $t1$ and $t4$ (see Figure 1). In line 2, user b requests to execute task $t2$ but again this is not possible since b has already executed task $t1$ and this would violate the SoD constraint between $t1$ and $t2$. All the other requests are granted, as they do not violate neither execution nor authorization constraints.

So far, we have described the key ideas underlying our technique while neglecting efficiency considerations related to the enumeration of all possible terminating execution sequences of the security-sensitive workflow. If we want our approach to scale up and handle real-world workflows, we have to design suitable heuristics as discussed in Section 4.

Table 2: A run of the monitor program $M_{n=3}$ for the security-sensitive workflow in Figure 1

	CF	Auth					*can_do*	
#	Token in	h_{t1}	h_{t2}	h_{t3}	h_{t4}	h_{t5}	(u, t)	Resp.
0	$p0$	-	-	-	-	-	$(a, t1)$	deny
1	$p0$	-	-	-	-	-	$(b, t1)$	grant
2	$p1, p2, p3$	b	-	-	-	-	$(b, t2)$	deny
3	$p1, p2, p3$	b	-	-	-	-	$(a, t2)$	grant
4	$p4, p2, p3$	b	a	-	-	-	$(c, t3)$	grant
5	$p4, p5, p3$	b	a	c	-	-	$(a, t4)$	grant
6	$p4, p5, p6$	b	a	c	a	-	$(b, t5)$	grant
7	$p7$	b	a	c	a	b	-	-

3. AUTOMATED SYNTHESIS OF RUN-TIME MONITORS

Considering the specification of workflows as transition systems presented in Section 2, we now describe how a symbolic model checker can compute a reachability graph that represents all terminating executions of the workflow (off-line phase) and how this is then translated to a Datalog program that implements the run-time monitor for the WSP (on-line phase).

3.1 Off-line

As already observed in Section 2.1, it is standard to use (extensions of) Petri nets to give a formal semantics to workflows written in BPM notation [24]. In turn, it is well-known how to represent (extension of) Petri nets as state transition systems (see, e.g., [21]), that are composed of a set of *state variables* and a set of *events*, as proposed in [22]. A *state* of the system is defined by the values of the variables. A *predicate* (Boolean function) over the state variables implicitly defines a set of states, i.e. the one containing the values of the variables for which the predicate evaluates to true. A state *satisfies* a predicate iff it belongs to the set of states implicitly defined by the predicate. An event has an *enabling condition*, which is a predicate on the state variables, and an *action*, which updates the state variables. When the enabling condition of an event evaluates to true in a given state s, we say that the event is *enabled* at s. Executing an event enabled at state s results in a new state s' obtained by applying the update of the event to the values of the variables in s. A *behavior* is a sequence of the form $s_0 \overset{e_0}{\to} s_1 \overset{e_1}{\to} \cdots$ where s_i is a state, e_i is an event, and state s_{i+1} is obtained by executing event e_i in state s_i, for $i = 0, 1, ...$ We say that a state s_n is *reachable* from a state s_0 iff there exists a behavior $s_0 \overset{e_0}{\to} s_1 \overset{e_1}{\to} \cdots s_{n-1} \overset{e_{n-1}}{\longrightarrow} s_n$.

For the class of security-sensitive workflows considered in this paper, the set V of state variables is the union of a set V_{CF} and a set V_{Auth} where the former contains a Boolean variable p_i for each place in the Petri net (for $i = 0, 1, ...$) and a Boolean variable d_t for each transition t in the Petri net, whereas the latter contains two function variables a_t and h_t mapping the set U of users to Booleans for each transition t in the net. Intuitively, p_i is true iff there is a token in the corresponding place, d_t is true iff task t has been executed, $a_t(u)$ is true iff user u has the right to execute task t, and $h_t(u)$ is true iff user u has executed task t. The enabling condition and the action of an event t are of the following forms: $enabled_{\text{CF}} \wedge enabled_{\text{Auth}}$ and $act_{CF} || act_{Auth}$, respectively, where $enabled_{\text{CF}}$ is a predicate over V_{CF}, $enabled_{\text{Auth}}$ is a predicate over V_{Auth}, act_{CF} (act_{Auth}, resp.) is the parallel ($||$) updates of (some of) the variables in V_{CF} (V_{Auth}, resp.), which are written as $x_1, ..., x_k := v_1, ..., v_k$ for x_i a state variable and v_i is the value to which x_i should be updated to. An update of a function variable f from users to Booleans is written as $f(u) := b$ where u is a user, b is a Boolean value, and after the update the function is identical to the previous one except at u for which the value b is returned. An event is a tuple $(t(u), enabled_{\text{CF}} \wedge enabled_{\text{Auth}}, act_{CF} || act_{Auth})$ written as

$$t(u) : enabled_{\text{CF}} \wedge enabled_{\text{Auth}} \to act_{CF} || act_{Auth} \qquad (6)$$

where t is the name of the event (taken from a finite set) and u is a user. Notice that an event is parametric with respect to a user; thus, (6) specifies a collection of events, one for every u in the set U of users. A *security-sensitive (state) transition system over the finite set U of users* is a tuple $(V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Tr)$ where U is a finite set of users, $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}$ is the set of state variables as described above, and Tr is the set of events obtained by considering all users in U.

Let $\mathcal{U}$ be an unbounded set of users and $S = (V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Tr)$ be a security-sensitive workflow over a finite set $U \subseteq \mathcal{U}$, I and F be two predicates over $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}$ and V_{CF}, respectively, characterizing the set of *initial* and *final* states. (Intuitively, F describes the set of states in which the security-sensitive workflow terminates: to express this, the variables in V_{CF} are sufficient.) The goal of the off-line phase is to compute the set $B(S, I, F)$ of all behaviors $s_0 \overset{e_0}{\to} s_1 \overset{e_1}{\to} \cdots s_{n-1} \overset{e_{n-1}}{\longrightarrow} s_n$ such that s_0 is an initial state (i.e. satisfies I) and s_n is a final state (i.e. satisfies F), for every finite sub-set U of users in $\mathcal{U}$.

Symbolic behaviors. We solve the problem of enumerating all possible behaviors of a security-sensitive workflow $S = (V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Tr)$ for every sub-set U of users in $\mathcal{U}$ by using a symbolic representation for S and U. We use first-order logic formulae [15] to represent sets of states. A *state formula* is a first-order formula containing (at most) the state variables in $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}} \cup V_{\mathrm{User}}$ as free variables where V_{User} is a set of variables taking values over the set U of users. A state formula P evaluates to true (in symbols, $s, v \models P$) or false (in symbols, $s, v \not\models P$) in a state s of the system and for an assignment v of the user variables (i.e. a mapping from V_{User} to U): for each variable x in $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}} \cup V_{\mathrm{User}}$ that appears free in P, replace x by its value in s or v and then evaluate the resulting formula. In other words, state formulae define predicates or, equivalently, sets of states. Examples of state formulae are (1) and (2) describing the sets of initial and final states, respectively, of the security-sensitive workflow in Figure 2. A *symbolic event* is a tuple of the form (6) where, this time, u is a first-order variable in V_{User}, $enabled_{\mathrm{CF}}$ is a state formula over V_{CF}, and $enabled_{\mathrm{Auth}}$ is a state formula over $V_{\mathrm{Auth}} \cup V_{\mathrm{User}}$, act_{CF} is as before, and act_{Auth} is of the form $f(u) := b$ where b is a Boolean value and u is the same variable in the label $t(u)$. A *symbolic security-sensitive transition system* is a tuple $(V_{\mathrm{CF}} \cup V_{\mathrm{Auth}} \cup V_{\mathrm{user}}, Ev)$ where $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}$ is the set of state variables, V_{User} is the set of user variables, and Ev is a finite set of symbolic events. The semantics of a symbolic security-sensitive transition system $(V_{\mathrm{CF}} \cup V_{\mathrm{Auth}} \cup V_{\mathrm{User}}, Ev)$ is axiomatically defined by using the notion of weakest liberal precondition (wlp) [14]:

$$\mathsf{wlp}(Ev, P) \;:=\; \bigvee_{(t(u):en \to act) \in Ev} (en \wedge P[act]) \qquad (7)$$

where $P[act]$ denotes the formula obtained from P by substituting the state variable v with the value b when the assignment $v := b$ is in act_{CF} and substituting $v(x)$ with either $v(x) \vee x = u$ when $v(x) := true$ is in act_{Auth} or with $v(x) \wedge x \neq u$ when $v(x) := false$ is in act_{Auth} for x in V_{User} and $act := act_{\mathrm{CF}} \wedge act_{\mathrm{Auth}}$. When Ev is a singleton containing a single symbolic event ev, we write $\mathsf{wlp}(ev, P)$ instead of $\mathsf{wlp}(\{ev\}, P)$. Notice that $\mathsf{wlp}(Ev, P)$ is equivalent to $\bigvee_{ev \in Ev} \mathsf{wlp}(ev, P)$. To make expressions more compact, we also write $\mathsf{wlp}(t(u), P)$ instead of $\mathsf{wlp}(t(u) : en \to act, P)$.

To illustrate, we compute $\mathsf{wlp}(t5(u), (2))$ where the symbolic event $t5(u)$ is defined in Table 1 by using (7):

$$\left(\begin{array}{l} p4 \wedge p5 \wedge p6 \wedge \neg d_{t5} \wedge \\ a_{t5}(u) \wedge \neg h_{t3}(u) \wedge \neg h_{t2}(u) \end{array} \right) \wedge (\bigwedge_{i=0,\ldots,3} \neg pi \wedge \bigwedge_{i=0,\ldots,4} d_{ti})$$

which is equivalent to

$$(\bigwedge_{i=0,\ldots,3} pi \wedge \neg p4 \wedge \neg p5 \wedge \neg p6 \wedge \bigwedge_{i=1,\ldots,4} d_{ti} \wedge \neg d_{t5}) \wedge \\ a_{t5}(u1) \wedge \neg h_{t3}(u1) \wedge \neg h_{t2}(u1))$$

and it identifies those states in which there is a token in places $p4$, $p5$, and $p6$, task $t5$ has not yet been executed whereas tasks $t1$, ..., $t4$ have been executed, user $u1$ has the right to execute $t5$ and has executed neither $t2$ nor $t3$. This is exactly the formula labeling node 1 in Figure 3.

A *symbolic behavior* is a sequence of the form $P_0 \overset{e_0}{\longrightarrow} P_1 \overset{e_1}{\longrightarrow} \cdots \overset{e_{n-1}}{\longrightarrow} P_n$ where P_i is a state formula and e_i is a symbolic event such that (a) $P_0 \wedge I$ is satisfiable, (b) P_i is logically equivalent to $\mathsf{wlp}(e_i, P_{i+1})$ for $i = 0, ..., n-1$, and (c) P_n is F for I and F formulae characterizing the initial and final states, respectively. The crucial advantage of symbolic events is the use of variables to represent users instead of enumerating them. To illustrate, consider a simple security-sensitive workflow with just two tasks t_1, t_2 such that t_1 should be executed before t_2 and there is a SoD constraint between them. If the cardinality of the set U of users is n, then the cardinality of the set of all possible behaviors is $n^2 - n$. By using symbolic events, we can represent all such behaviors by a single symbolic behavior $P_0 \overset{t_1(u1)}{\longrightarrow} P_1 \overset{t_2(u2)}{\longrightarrow} P_2$ with the proviso that $u1 \neq u2$ where $u1, u2$ are variables. Before stating formally this result, we need to introduce the notion of security-sensitive transition system $T = (V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Ev_T)$ *associated to a symbolic security-sensitive transition system $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ and a finite set $U \subseteq \mathcal{U}$ of users*: if the symbolic event $t(ui) : en_S \to act_S$ is in Ev_S, then Ev_T contains an event $t(u_i) : en \to act$ where u is a user in U, en is the predicate interpreting the formula obtained from en_S by substituting the variable ui with u_i and all other user variables with users in U (in all possible ways), and act is obtained from act_S by substituting ui with u_i.

THEOREM 3.1. *Let $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ be a symbolic security-sensitive transition system and $T = (V_{CF} \cup V_{Auth}, Ev_T)$ be the associated security-sensitive transition system for the set $U \subseteq \mathcal{U}$ of users. If $s_0 \overset{t_0(u_0)}{\to} s_1 \overset{t_1(u_1)}{\to} \cdots s_{n-1} \overset{t_{n-1}(u_{n-1})}{\longrightarrow} s_n$ is a behavior of T for $u_0, ..., u_{n-1}$ in U, then there exists a symbolic behavior $P_0 \overset{t_0(u0)}{\longrightarrow} P_1 \overset{t_1(u1)}{\longrightarrow} \cdots \overset{t_{n-1}(un-1)}{\longrightarrow} P_n$ such that $s_i, v_i \models P_i$ with $v_i(ui) = u_i$ for $i = 0, ..., n-1$ and $s_n, v_{n-1} \models P_n$.*

This result tells us that a symbolic behavior is an adequate (and hopefully compact) representation of a set of behaviors. The proof is by a standard induction on the length of the behaviors and exploits the fact that the enforcement of authorization constraints depends only on two aspects: the identity of users (via the state variables a_t's modeling the interface to the concrete authorization policy establishing if a user has the right to execute a task) and the history of the computation (via the state variables h_t's keeping track of who has executed which tasks so that SoD and BoD constraints can be guaranteed to hold).

Computation of symbolic behaviors. Algorithm 1 computes the set of all possible symbolic behaviors of a symbolic security-sensitive workflow. It takes as input the symbolic security-sensitive workflow S together with the state formula F defining the set of final states and returns a labeled graph RG, called *reachability graph*, whose set of labeled paths is the set of all symbolic behaviors of S ending with F. The procedure incrementally builds the reachability graph RG by updating the set N of nodes, the set E of edges, and the labeling function λ from N to state formulae. Initially (line 1), a new node i is created (by invoking the auxiliary function `new`, which returns a "fresh" node—i.e. distinct from any other node already in N—at each invocation), N is assigned to the singleton containing node i, which is also labeled (via λ) by the final formula F. The algorithm also maintains the set TBV of nodes to be visited, which is made equal to N. Then, the main loop (lines 2–14) is entered by checking if there are some nodes to be visited (line 2). At each iteration, it is first (line 3) checked whether the set of states identified by the wlp of the formula $\lambda[i]$ with respect to the set Ev_S of symbolic events is included in the union of the sets of states that have been already generated. This is done by invoking `subsumed`(i,N,N') which returns true iff, for each symbolic event $ev \in Ev_S$, there exists a sub-set N' of $N-\{i\}$ and $wlp(ev,\lambda[i])$ implies the formula $\bigvee_{j\in N'}\lambda[j]$ (notice that the third argument N' is passed by reference). If this is the case, we can avoid to add a new node ν to N labeled by $wlp(ev,\lambda[i])$ as the symbolic behaviors arriving in ν have already been generated when visiting the nodes in N'. Thus, we can delete node i from TBV, add a new node j labeled by $wlp(ev,\lambda[i])$ together with an edge from j to i labeled by $\overline{ev}$ and—by invoking the auxiliary function `connect`—duplicate the initial part of each path passing through a node n' in N' by replacing n' with j provided that the newly created path is a symbolic behavior of the symbolic transition system. To illustrate, consider node 7 in Figure 3 (colored in red): $wlp(ti(u),\lambda[7])$ is unsatisfiable for $i=1,3,4,5$ (and can thus be ignored) whereas $wlp(t2(u3),\lambda[7])$ is satisfiable and implies $\lambda[13]$; this is checked by invoking `subsumed`

Algorithm 1 Building a symbolic reachability graph

Input: $S=(V_{\text{CF}}\cup V_{\text{Auth}}\cup V_{\text{User}}, Ev_S)$ and F
Output: $RG=(N,\lambda,E)$
1: $i\leftarrow$ new(); $N\leftarrow\{i\}$; $E\leftarrow\emptyset$; $\lambda[i]\leftarrow F$; $TBV\leftarrow\{i\}$;
2: **while** $TBV\neq\emptyset$ **do**
3: **if** `subsumed`(i,N,N') **then**
4: `connect`(N',i); $TBV\leftarrow TBV-\{i\}$;
5: **end if**
6: **for all** $ev\in Ev_S$ **do**
7: $P\leftarrow \mathsf{wlp}(ev,\lambda[i])$;
8: **if** P is satisfiable **then**
9: $j\leftarrow$ new(); $N\leftarrow N\cup\{j\}$; $E\leftarrow E\cup\{(i,\overline{ev},j)\}$;
10: $\lambda[j]\leftarrow P$; $TBV\leftarrow TBV\cup\{j\}$;
11: **end if**
12: **end for**
13: $i\leftarrow$ `pickOne`(TBV); $TBV\leftarrow TBV-\{i\}$;
14: **end while**
15: **return** (N,λ,E);

with $N'=\{13\}$. Thus, we create a new node (say) 29, with $\lambda[29]$ equal to $wlp(t2(u3),\lambda[7])$, draw an edge from 29 to 7 with label $t2(u3)$, duplicate the initial parts of the paths passing through node 13 (namely $\lambda[17]\xrightarrow{t1(u3)}\lambda[13]$ and $\lambda[18]\xrightarrow{t1(u4)}\lambda[13]$) while replacing 13 with 29 (thus obtaining $\lambda[17]\xrightarrow{t1(u3)}\lambda[29]$ and $\lambda[18]\xrightarrow{t1(u4)}\lambda[29]$), and then check that the newly created paths, namely

$$\lambda[17]\xrightarrow{t1(u3)}\lambda[29]\xrightarrow{t2(u3)}\lambda[7]\xrightarrow{t3(u2)}\lambda[4]\xrightarrow{t4(u1)}\lambda[4]\xrightarrow{t5(u1)}\lambda[0] \text{ and}$$
$$\lambda[18]\xrightarrow{t1(u4)}\lambda[29]\xrightarrow{t2(u3)}\lambda[7]\xrightarrow{t3(u2)}\lambda[4]\xrightarrow{t4(u1)}\lambda[4]\xrightarrow{t5(u1)}\lambda[0]\,,$$

are symbolic behaviors. It turns out that only the latter is so, since the former violates the SoD constraint between $t1$ and $t2$. We thus add only the path $\lambda[18]\xrightarrow{t1(u4)}\lambda[29]\xrightarrow{t2(u3)}\lambda[7]$ to the graph in Figure 3. Nodes 5, 10, and 12 are handled similarly. These extensions to the graph in Figure 3 are omitted to keep it readable.

If node i is not subsumed by those in N (i.e. `subsumed`(i,N) returns false), we compute the wlp with respect to all symbolic events (inner loop 6–11). I.e., for each ev in Ev_S, we compute $\mathsf{wlp}(ev,\lambda[i])$ labeling the node i being visited (line 6) and verify if it defines a set of states which is non-empty, by checking the satisfiability of the resulting formula (line 7). If this is the case, we add a fresh node j, labeled by the wlp just computed, to N, an edge from i to j labeled by the name $\overline{ev}$ of the symbolic event ev, and add the newly created node j to the set TBV (lines 8 and 9). For instance, when computing the wlp of the formula labeling node 0 in Figure 3, we found out that only the symbolic event named $t5(u1)$ generates a formula denoting a non-empty set of states and thus we added node 1 labeled by such a formula and an edge from 1 to 0 labeled by $t5(u1)$. After exiting the inner loop, if the set TBV of nodes to be visited is non-empty, we consider another node to be visited by invoking the auxiliary function `pickOne`(TBV) which non-deterministically selects an element from TBV (when this is empty, `pickOne` returns a distinguished element), which is then deleted, and we start the main loop again.

THEOREM 3.2. *Let I be the initial state formula. If Algorithm 1 returns the reachability graph RG when taking as input the symbolic security-sensitive transition system $S=(V_{CF}\cup V_{Auth}\cup V_{User},Ev_S)$ and the final state formula F, then the set of all symbolic behaviors of S is the set of labeled paths in RG starting with a node labeled by a formula whose conjunction with I is satisfiable and ending with a node labeled by F.*

The proof of this theorem uses the definition of wlp and the properties discussed above about the auxiliary functions `subsumed` and `connect`. It is possible to show that Algorithm 1 always terminates by adapting the results in [8].

3.2 On-line

Theorem 3.2 implies that starting from an initial state (i.e. one satisfying the initial formula I) in the reachability graph computed by Algorithm 1, it is always possible to reach a final state (i.e. one satisfying the final formula F). If no event can be enabled infinitely often without being executed—called *strong fairness*—then a final state is eventually reached. (As observed in [25], the assumption of strong fairness is reasonable in the context of workflow management since decisions to execute tasks are under the responsibility of applications or humans.) This is the key to prove the following result, underlying the correctness of the automated technique—to be described below—for extracting (part of) the monitor from the reachability graph computed by Algorithm 1.

THEOREM 3.3. *Let $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ be a symbolic security-sensitive transition system and $T = (V_{CF} \cup V_{Auth}, Ev_T)$ be the associated security-sensitive transition system for the finite set $U \subseteq \mathcal{U}$ of users. Furthermore, let $RG = (N, \lambda, E)$ be the symbolic reachability graph computed by Algorithm 1 when taking as input S and a final state formula F. If the state s satisfies a formula $\lambda[i]$ for some $i \in N$, then there exists a behavior $s_0 \stackrel{t_0(u_0)}{\rightarrow} s_1 \stackrel{t_1(u_1)}{\rightarrow} \cdots s_{n-1} \stackrel{t_{n-1}(u_{n-1})}{\longrightarrow} s_n$ of T such that (i) $s_0 = s$, (ii) s_n satisfies F, and (iii) $(i, t(x), j) \in E$ with $t_0 = t$ and $s_0(x) = u_0$.*

Thus, if T is in state s and we want to know if a certain user u_0 can execute task t_0 while guaranteeing that the authorization constraints are satisfied and the workflow terminates, it is sufficient to find a node of the reachability graph that is satisfied by the s and one of the outgoing edges is labeled by t_0. Indeed, this is exactly the task a monitor is supposed to perform! To make this operational, we observe that we can associate the Datalog clause [10]

$$can\_do(u,t) \quad \leftarrow \quad \Gamma \wedge C_k[i]$$

for each node $i \in N$ and edge $(i, t(u), j) \in E$, where Γ is the conjunction of atoms of the form $is\_user(x)$ for each variable x in $C_k[i]$ with $RG = (N, \lambda, E)$ and $\bigvee_{k=1}^{n_i} C_k[i]$ is the disjunctive normal form of $\lambda[i]$. Let $D(RG)$ be the set of Datalog clauses built in this way from the reachability graph RG. (It is straightforward to check that $D(RG)$ is non-recursive; see [10] for a precise definition). Formally, the addition of Γ is needed to make $D(RG)$ a safe Datalog program (see again [10] for a precise definition) so that answering queries always terminates.

After building the Datalog program $D(RG)$, it is straightforward to build a run-time monitor. Let U be a finite set of users, $A \subseteq V_{\mathrm{Auth}}$ be the sub-set of state variables a_t's modeling the interface to the concrete authorization policy establishing if a user has the right to execute a task, and P be a Datalog program formalizing an authorization policy (i.e. P contains a clause of the form $is\_user(u)$ for each $u \in U$ and clauses whose heads contain only the predicates in A). We call P a *Datalog authorization policy program over the interface variables in V_{Auth}*. (How to write authorization policies in Datalog is outside the scope of this paper, the interested reader is pointed to [18].) Any assignment over the states variables in $V_{\mathrm{CF}} \cup (V_{\mathrm{Auth}} - A)$ can be represented by a set Σ of Datalog facts of the forms p, $\neg p$, $h_t(u)$, or $\neg h_t(u)$ for $p \in V_{\mathrm{CF}}$ and $h_t \in (V_{\mathrm{Auth}} - A)$. We call Σ a *partial Datalog state over the state variables in $V_{CF} \cup (V_{Auth} - A)$*.

THEOREM 3.4. *Let $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ be a symbolic security-sensitive transition system, $T = (V_{CF} \cup V_{Auth}, Ev_T)$ be the associated security-sensitive transition system for the finite set $U \subseteq \mathcal{U}$ of users, and $RG = (N, \lambda, E)$ be the symbolic reachability graph computed by Algorithm 1 when taking as input S and a final state formula F. Additionally, let P be a Datalog authorization policy over the interface variables in V_{Auth} and Σ be a partial Datalog state. A user $u \in U$ can execute task t guaranteeing the satisfaction of all authorization constraints and the termination of the workflow iff the query $can\_do(u, t)$ is answered positively by the Datalog program $D(RG) \cup P \cup \Sigma$.*

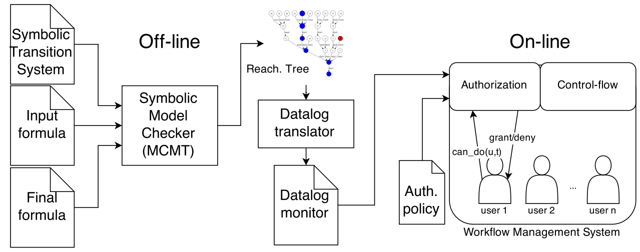

Figure 4: Architecture of the implementation

This is the main result of the paper and guarantees the correctness of our procedure to synthesize run-time monitors. It is a consequence of the definition of Datalog authorization policy program, partial Datalog state, and Theorem 3.3. Notice that when both $D(RG)$ and P are non-recursive (stratified) Datalog programs, queries can be answered very efficiently in *LogSpace* and can be translated to SQL without aggregate operators (such as AVG and COUNT).

4. EXPERIMENTS

We have implemented the technique for the automated synthesis of monitors in a tool whose architecture is depicted in Figure 4. On the left (Off-line), we have a symbolic model checker (MCMT [17]) that, given a symbolic transition system representing a workflow together with initial and final formulae, computes the reachability graph according to Algorithm 1. The graph is then passed to the Datalog translator—implemented in Python (v2.7.5)—which creates a Datalog program as explained in Section 3.2. On the right of Figure 4 (On-line), we use pyDatalog (v0.14.5) as our Datalog engine to answer authorization queries of the form "*can user u execute task t and guarantee the successful termination of the workflow?*".

For a first evaluation of the scalability of our technique, we focus on an important class of workflows encountered in practice, namely those designed according to a hierarchical decomposition principle. The idea is to split a complex workflow into subflows which are again decomposed into smaller subflows up to a desired level of detail. Several workflows management systems support this style of workflow specification following an established line of works in both academy (see, e.g., [23]) and industry (see, e.g., the "SAP Modeling Handbook," available on-line at `http://wiki.scn.sap.com/wiki/display/ModHandbook/Process+Hierarchy`). Hierarchic workflows are structured according to the notion of *task refinement*, i.e. a task can be refined into a subflow. To illustrate, the workflow on the left of Figure 5 contains the task Bookings, which can be refined by the workflow on the right of the same picture. This means that by replacing task Bookings with the workflow on the right of the figure, we derive the workflow in Figure 1.

Besides fostering reuse and simplifying maintenance of complex workflows, hierarchic specifications allow for the development of a divide-and-conquer strategy when applying

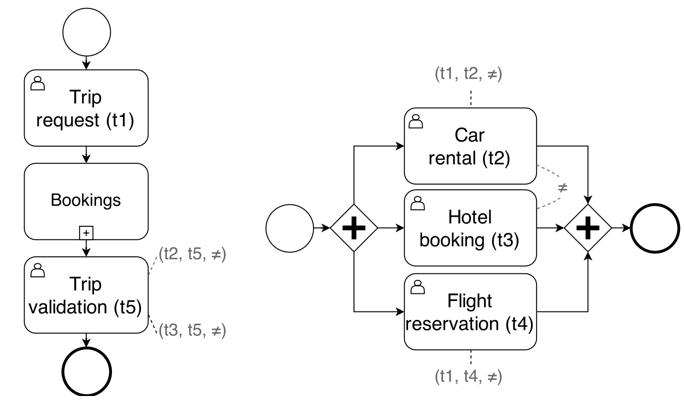

Figure 5: Hierarchic specification of the trip request example workflow (cf. Figure 1)

Algorithm 1. I.e., given a hierarchic workflow, it is possible to compute its monitor for the WSP by first computing the monitors for each of the subflows separately and then "gluing" them together. To understand what we mean by "gluing," let us consider the hierarchic workflow in Figure 5. We run the Algorithm 1 first on the workflow on the left, on that on the right, and we run the Datalog translator on the resulting reachability graphs to obtain the Datalog programs $D(RG_l)$ and $D(RG_r)$, respectively. Let p_i^l, p_f^l and p_i^r, p_f^r be the predicates corresponding to the initial and final places of the Petri nets representing the workflows on the left and on the right of Figure 5. The Datalog program for the hierarchic workflow (equivalent to the workflow in Figure 1) can be obtained by adding the clauses $p_i^r \leftarrow p_f^l$ (to transfer the control flow from the workflow on the left to that on the right) and $p_i^l \leftarrow p_f^r$ (to transfer back the control flow from the workflow on the right to that on the left) to $D(RG_l) \cup D(RG_r)$ and finally removing the clauses in which the identifier of the task Bookings occur. As can be seen in the example, it is possible to have authorization constraints that span different subflows. For each constraint in this case, a literal $\neg h_t(u)$ is added to the corresponding transition in the symbolic transition system. For instance, the SoD constraint $(t1, t2, \neq)$ adds the literal $\neg h_{t1}(u)$ to the enabling condition of $t2$. These literals are unconstrained when each subflow is taken separately, but after the "gluing" process, they act as any other constraint.

This modular approach to synthesizing monitors has been implemented in our tool and is key to scalability. In fact, without using hierarchic specifications, for a workflow with up to 5 tasks, running Algorithm 1 (the most expensive step of our technique) takes few seconds on a standard laptop; for 6 tasks, around a minute; and for 7 tasks, already two hours and a half! Since hierarchic specifications are so important for scalability, we have designed and implemented heuristics that, given monolithic workflows, are capable of deriving equivalent hierarchic specifications. For lack of space, we leave their description to future work and assume in the following that our tool is presented with hierarchic workflows. As observed above, hierarchic specifications of workflows are frequently available so that results from experiments on them already give significant indications about the efficiency of techniques for synthesizing run-time monitors for the WSP.

4.1 Real-world workflows

We have experimented with some real-world workflows taken from related works and present here two cases in details. These examples show the expressiveness of our approach and illustrate the use of all the basic control-flow patterns (such as sequence and exclusive choice) besides advanced patterns for arbitrary loops [26]. Figure 6 shows the two examples in extended BPM notation, where—following [3]—the circles marked by the depiction of a user leaving a door represent *release points* and the gray lines show the authorization constraints. A release point is a special event whereby the history of executions of the workflow is erased, so that authorization constraints can be handled in loops.

Drug dispensation process [4] (left of Figure 6). The execution of an instance of this workflow starts with a Patient requesting drugs to a Nurse ($t1$). The Nurse consults the Patient's record and sends it to a PrivacyAdvocate ($t2$), who decides if this data should be anonymized ($t3$ and $t4$). If the drug prescription has therapeutic notes, they must be reviewed by a Therapist ($t5$) and in parallel, a Researcher can add data related to experimental drugs ($t6$). In the end a Pharmacist either approves or denies the process ($t7$) and a Nurse carries out the related tasks: collect and dispense the drugs ($t9$ and $t10$) or notify the Patient ($t8$). A SoD constraint for this workflow, not shown in the Figure, is $(t1, t7, \neq)$: the same user cannot act as Patient and Pharmacist, so that a Pharmacist cannot dispense drugs to himself.

A workflow of this size (10 tasks) would be intractable for our tool. Thus, we come up with a hierarchic specification consisting of two subflows to be executed one after the other; the former is refined to the subflow containing tasks t_1, ..., t_4 and the latter the subflow with tasks t_5, ..., t_{10}. According to some control flow operators, not all tasks must be executed in the workflow for its successful termination. In fact, tasks $t4$, $t5$ and $t6$ may or may not be executed depending on certain conditions (e.g., "*anonymize?*") while tasks $t8$ and $t9$ are mutually exclusive. To represent the decisions that have to be taken to complete the workflow, we create transitions for the various branches whose enabling conditions depend on additional variables—called *environment variables*—modeling non-deterministic choices of the environment. For instance, the fact that task $t7$ is followed by the decision point *approved?* can be represented by the following two transitions:

$$\begin{aligned} t_7^{true}(u) = p6 \wedge p7 \wedge \neg d_{t7} \wedge app \wedge a_{t7} \wedge \neg h_{t1}(u) \;\rightarrow \\ p6, p7, p10, d_{t7}, h_{t7}(u) := F, F, T, T, T \\ t_7^{false}(u) = p6 \wedge p7 \wedge \neg d_{t7} \wedge \neg app \wedge a_{t7} \wedge \neg h_{t1}(u) \;\rightarrow \\ p6, p7, p8, d_{t7}, h_{t7}(u) := F, F, T, T, T\,. \end{aligned}$$

When the environment variable *app* is true (cf. t_7^{true}), tasks $t9$ and $t10$ must be executed; when it is false (t_7^{false}), only task $t8$ is executed. Besides permitting the precise representation of the control flow, environment variables allow for writing final formulae differentiating between the alternative execution. For example, assuming a Petri net representation of the drug dispensation process that has a place $p4$ after $t4$ and before $t5$, we run the model checker on the first subflow with the final formula

$$\begin{aligned} (\neg p0 \wedge \neg p1 \wedge ... \wedge p4 \wedge d_{t1} \wedge d_{t2} \wedge d_{t3} \wedge d_{t4}) \quad \vee \\ (\neg p0 \wedge \neg p1 \wedge ... \wedge \neg p4 \wedge d_{t1} \wedge d_{t2} \wedge d_{t3} \wedge \neg d_{t4}) \quad . \end{aligned}$$

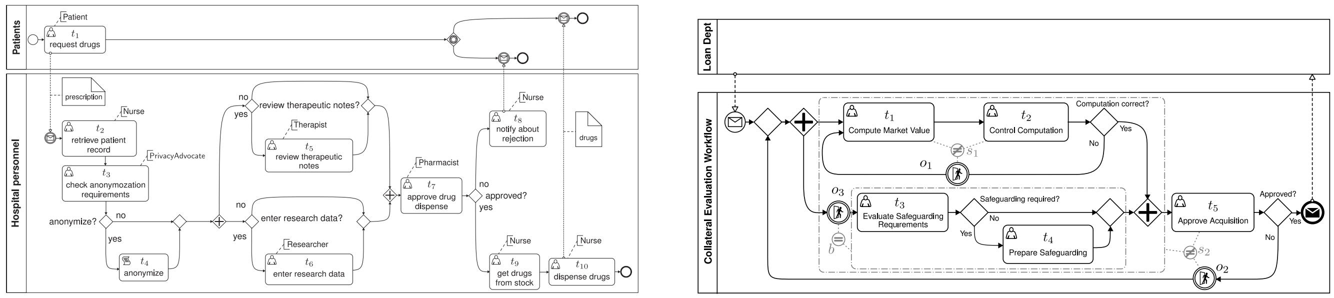

Figure 6: Left: Drug dispensation process from [4]. Right: Collateral evaluation workflow, from [3]

A similar final formula can be derived for the second subflow.

The reachability graph computed for the first subflow contains 200 nodes while that for second 231 nodes. Using a MacBook laptop (see below for a detailed description of its configuration), the time spent to compute the reachability graph and translate it to a Datalog program is around 15s (roughly, 3s for the first and 12s for the second). The time taken by the synthesized monitor to answer access requests is almost negligible.

Collateral evaluation workflow [3] (right of Figure 6). It is executed to evaluate a collateral pledge for a loan. The main difference with the previous workflow is the presence of loops, exemplified by the decision points *Computation correct?* and *Approved?*. This workflow has only 5 tasks, so we do not need to transform it to an equivalent hierarchic specification. Similarly to conditionals, we add environment variables to model decision points and suitable transitions for loops. For instance, task $t2$ is followed by the decision point—encoded by the environment variable $l1$—of the first loop which can be represented by transitions t_2^{true} when $l1$ is true and the loop is taken and t_2^{false} when $l1$ is false and the loop is not taken. The enabling condition of the transitions are analogous to what was shown above while the updates take into account the use of release points when updating the history variables h_t's in order to support the use of authorization constraints in loops as discussed in [3].

The reachability graph for the collateral evaluation process has 135 nodes and the time spent for computing it is around 4s. As before, answering authorization constraints in the on-line phase is immediate.

4.2 Synthetic benchmarks

To test the scalability of our approach, we have extended the generator of random workflows used in [12][1] to produce hierarchic workflows. Our generator has the following parameters: n_w, the number of subflows and n_{tw}, the number of tasks in each subflow ($n_t = n_w \cdot n_{tw}$ is the total number of tasks; n_u, the number of users; p_a, the authorization density which is the ratio, expressed as a percentage, between the cardinality of $\bigcup_t \{a_t(u) = true | u \text{ is a user}\}$ and $n_t \cdot n_u$ (where t ranges over the set of tasks); and p_c, the constraint density which is the ration between the number of SoD constraints in the set C and n_t.

The generator also produces random (finite) sequences $(r_0, r_1, ..., r_n)$ of authorization requests where $r_i = (t, u)$ for t a task and u a user, encoding the question "can u perform task t according to the authorization policy specified by the a_t's and the constraints in C while guaranteeing its termination?"

According to our experience with real-world workflows (cf. Section 4.1), we set n_{tw} to 5 and increase the number n_w of subflows so that the total number n_t of tasks in the generated workflows range from 10 to 500 (notice that [12] considers workflows with at most 150 tasks). More precisely, we let $n_t = 10, 20, ..., 150, 200, 250, ..., 500$ and, following [12], $n_u = n_t$, $p_a = 100\%, 50\%, 10\%$, $p_c = 5\%, 10\%, 20\%$.

Figure 7 shows the behavior of our prototype tool, for the off-line (left) and on-line (right) phases, on the hierarchic workflows produced by the random generator with the parameters described above. The x-axis shows the number n_t of task in the workflow, the y-axis the timings in seconds, each line corresponds to different values for the pair (p_a, p_c) of parameters (recall that $n_u = n_t$). The timings are obtained on a MacBook 2014 laptop with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM, running MAC OS X 10.9.4.

It is clear that the computation time of our tool in both the on-line and off-line phases is linear in the number of tasks in the workflows for any value of the pair (p_a, p_c) of parameters. For the off-line phase, this is so because of the divide-and-conquer strategy described above and supported by hierarchic workflows. For the on-line phase, the linear growth is due to the fact that the synthesized Datalog programs belong to a class whose requests can be answered in linear-time. Notice also that that for workflows with $n_t \leq 200$, the (median) time to answer a request is under 1 second while for workflows with $200 < n_t \leq 500$, it is around 1.6 seconds. This clearly demonstrates that the monitors synthesized by our tool are suitable to be used on-line.

To give an idea of the distribution of the answers given by the synthesized monitors to the randomly generated sequences of authorization requests, Figure 8 shows the number of denied (in red) and granted (in green) requests (y-axis) for workflows with $n_t = 10, ..., 400$ (x-axis) and $(p_a, p_c) = (10, 20)$ (the number shown on the x-axis must be multiplied by 10 to obtain the number of tasks in the workflow). We believe that these results clearly show the scalability and practical applicability of our approach on the important class of hierarchically specified workflows.

5. RELATED WORK

Verification of array-based systems. Model Checking Modulo Theories [16] is an approach for the verification of array-based systems based on the computation of pre-

[1] We would like to thank the authors for sharing their source code with us.

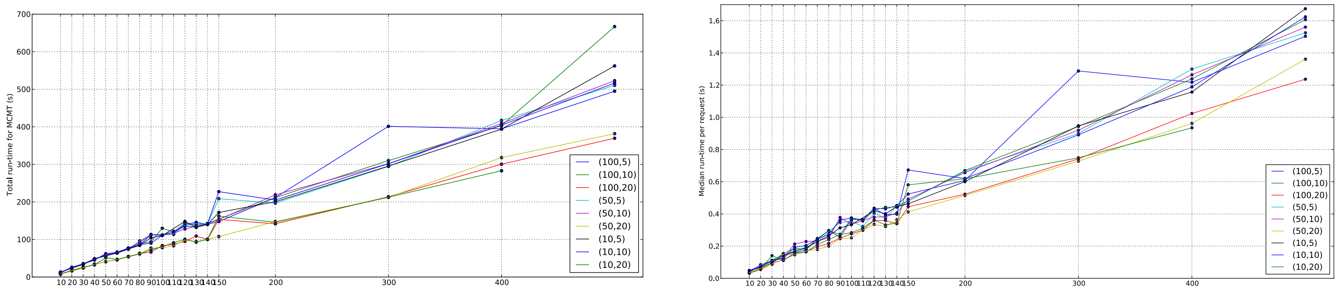

Figure 7: Total run-time of off-line (left) and online (right) phase by the number of tasks in all configurations

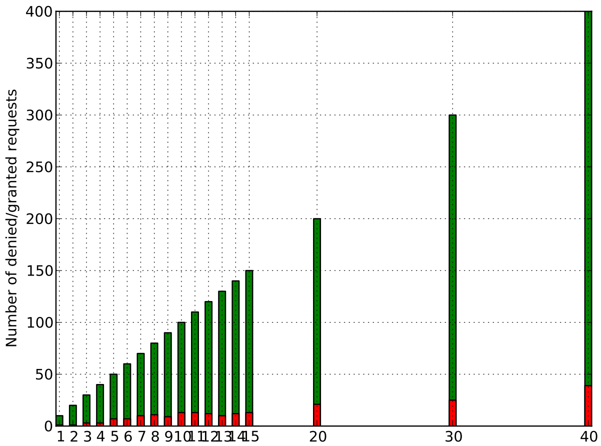

Figure 8: Number of granted (green) and denied (red) requests for increasing values of n_t with $(p_a, p_c) = (10, 20)$

images of a set of states using first-order formulae and on reducing fix-point checks to SMT solving. This approach is implemented by the model checker MCMT [17]. The link between array-based systems and security-sensitive workflows was given in [8], where composed array-based systems represent security-sensitive workflows, with a terminating procedure for the verification of reachability properties for this class of systems.

Workflow Satisfiability. Bertino et al. [6] described the specification and enforcement of authorization constraints in workflow management systems, presenting constraints as clauses in a logic program and an exponential algorithm for assigning users and roles to tasks without violating them, but considering only linear workflows. Crampton [11] showed another model for specifying constraints, considering workflows as DAGs, and an algorithm to determine whether there is an assignment of users to tasks that satisfies the constraints, showing that it can be incorporated into a reference monitor. [13] extended the previous work to consider the effects of delegation on satisfiability, showing algorithms to only allow delegations that can still satisfy a workflow. Crampton et al. [12] used model checking on an NP-complete fragment of linear temporal logic to decide the satisfiability of workflow instances. The authors presented three different encodings in LTL(F) that can compute a set of solutions, minimal user bases and a safe bound on resiliency. The synthesis of monitors was left as future work.

Wang and Li [27] proposed a role-and-relation based access control model that allows to describe the relationships between users and thus specify complex authorization constraints. The authors showed that the WSP is NP-complete in their model and that this intractability is inherent in authorization systems supporting simple constraints. They showed that with only equality and inequality relations, using the number of tasks as a parameter renders the WSP fixed-parameter tractable. They also reduced the problem to SAT. Yang et al. [20] studied the complexity of several formulations of the WSP, considering the possibility of different control-flow patterns, and showed that, in general, the problem is intractable.

Basin et al. [5] considered the problem of choosing authorization policies that allow a successful workflow execution and an optimal balance between system protection and user empowerment. They treat the problem as an optimization problem (finding the cost-minimizing authorization policy that allows a successful workflow execution) and show that, in the role-based case, it is NP-complete. They generalize the decision problem of whether a given authorization policy allows a successful workflow execution to the notion of an optimal authorization policy that satisfies this property. In a following work, Basin et al. [4] used the Separation of Duties Algebra (SoDA) to enforce SoD constraints in a dynamic, service-oriented enterprise environment. The authors generalized SoDA's semantics to workflow traces that satisfy a term and refined it for control-flow and role-based authorizations. Their formalization, based on CSP, is the base for provisioning SoD as a Service, with an implementation using a workflow engine and a SoD enforcement monitor. [4] is the closest to us in terms of monitor implementation, but their monitor only verifies if a trace of a workflow satisfies a SoDA term, being incapable of checking whether there is a future trace that can be concatenated in order to satisfy the workflow.

This work. This work extends the results in [7, 9] by describing a fully automated technique, arguing its correctness, providing an implementation, and a thorough experimental evaluation. The main advantages of our work with respect to the others discussed above is the specification of the security-sensitive workflows as array-based systems, the consideration of an off-line and an on-line phase and the composition of sub- workflows. These three characteristics allow us to efficiently compute all terminating executions of large instances of workflows for a finite but unbounded number of users and then translate it to a Datalog program that acts as an efficient run-time monitor.

6. CONCLUSIONS

We have introduced and implemented a precise technique to automatically synthesize run-time monitors capable of ensuring the successful termination of workflows while enforcing authorization policies and SoD constraints, thus solving the run-time version of the WSP. It consists of an off-line phase in which we compute a symbolic representation of all possible behaviors of a workflow and an on-line phase in which the monitor is derived from such a symbolic representation. An advantage of the technique is that changes in the policies can be taken into account without re-running the off-line phase since only an abstract interface to policies is required. The interface is refined to the concrete policy only in the on-line phase. We have also described the assumptions for the correctness of the technique (cf. Theorem 3.4). An extensive experimental evaluation with an implementation of the technique shows the scalability of our approach on the important class of hierarchic workflows.

As future work, we plan to present a detailed description of our heuristics to obtain equivalent hierarchic specification of monolithic workflows. This would allows us to enlarge the scope of applicability of our approach even further. We also intend to integrate our prototype in available workflow execution engines—e.g., the one available in the SAP HANA platform (`http://www.sap.com/hana`)—to collect data about the performances of our monitors on real workflows, and compare the results with those in this paper. This would be an important step towards the creation of a library of benchmarks to set a standard for the evaluation of workflow analysis techniques.

7. REFERENCES

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of LICS*, pages 313–321, 1996.

[2] A. Armando and S. E. Ponta. Model Checking of Security-sensitive Business Processes. In *6th Int. Ws. on Formal Aspects in Security and Trust (FAST)*, 2009.

[3] D. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *Proc. of CSF'11*, pages 99–113, Washington, DC., 2011. IEEE Computer Society.

[4] D. Basin, S. J. Burri, and G. Karjoth. Dynamic enforcement of abstract separation of duty constraints. *ACM TISSeC*, 15(3):13:1–13:30, Nov. 2012.

[5] D. Basin, S. J. Burri, and G. Karjoth. Optimal workflow-aware authorizations. In *Proc. of SACMAT '12*, pages 93–102, New York, NY, 2012. ACM.

[6] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *TISSeC*, 2:65–104, 1999.

[7] C. Bertolissi and S. Ranise. A Methodology to build run-time Monitors for Security-Aware Workflows. In *Proc. of ICITST'13*. IEEE, 2013.

[8] C. Bertolissi and S. Ranise. Verification of Composed Array-based Systems with Applications to Security-Aware Workflows. In *Proc. of FROCOS'13*. Springer, 2013.

[9] C. Bertolissi and S. Ranise. A smt–based methodology for monitoring of security–aware workflows. *Int. J. of ITST*, 5(3):275–290, 01 2014.

[10] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE TKDE*, 1(1):146–166, 1989.

[11] J. Crampton. A reference monitor for workflow systems with constrained task execution. In *10th ACM SACMAT*, pages 38–47. ACM, 2005.

[12] J. Crampton, M. Huth, and J.-P. Kuo. Authorized workflow schemas: deciding realizability through ltl(f) model checking. *STTT*, 16(1):31–48, 2014.

[13] J. Crampton and H. Khambhammettu. Delegation and satisfiability in workflow systems. In *SACMAT*, pages 31–40, New York, NY, USA, 2008. ACM.

[14] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[15] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York-London, 1972.

[16] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *In LMCS, Vol. 6, Issue 4*, 2010.

[17] S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *IJCAR*, volume 6173 of *LNCS*, pages 22–29, 2010.

[18] N. Li and J. C. Mitchell. Datalog with constraints: a foundation for trust management languages. In *PADL'03*, pages 58–73, 2003.

[19] T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[20] I. R. P. Yang, X. Xie and S. Lu. Satisfiability analysis of workflows with control-flow patterns and authorization constraints. *IEEE TSC*, 99, 2013.

[21] S. Sankaranarayanan, H. Sipma, and Z. Manna. Petri net analysis using invariant generation. In *In Verification: Theory and Practice, LNCS 2772*, pages 682–701. Springer Verlag, 2003.

[22] A. U. Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Comput. Surv.*, 25(3):225–262, Sept. 1993.

[23] W. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Business Process Management*, volume 1806 of *LNCS*, pages 161–183. Springer, 2000.

[24] W. van der Aalst and A. H. M. T. Hofstede. Yawl: Yet another workflow language. *Inf. Systems*, 30:245–275, 2003.

[25] W. van der Aalst, K. van Hee, A. ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Comp.*, 23(3):333–363, 2011.

[26] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.

[27] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *TISSeC*, 13:40:1–40:35, December 2010.