Modular Synthesis of Enforcement Mechanisms for the Workflow Satisfiability Problem: Scalability and Reusability[*]

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

Serena Elisa Ponta
SAP Labs France
serena.ponta@sap.com

Silvio Ranise
Fondazione Bruno Kessler
ranise@fbk.eu

ABSTRACT

Modularity is an important concept in the design and enactment of workflows. However, supporting the specification and enforcement of authorization in this setting is not straightforward. In this paper, we introduce a notion of component and a combination mechanism for security-sensitive workflows. These are business processes in which execution constraints on the tasks are complemented with authorization constraints (e.g., Separation of Duty) and authorization policies (specifying which users can execute which tasks). We show how authorization constraints can also be imposed across components and demonstrate the usefulness of our notion of component by showing (i) the scalability of a technique for the synthesis of run-time monitors for security-sensitive workflows; and (ii) the design of a plug-in for the reuse of workflows and related run-time monitors inside an editor for security-sensitive workflows.

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

Business Process, Modularity, Workflow Satisfiability

1. INTRODUCTION

Business process designers constantly strive to adapt to rapidly evolving markets under continuous pressure of regulatory and technological changes. In this respect, a frequent problem faced by companies is the lack of automation when trying to incorporate new requirements into existing processes. A traditional approach to business process modeling frequently results in large models that are difficult to change and maintain. This makes it critical that business process models be modular and flexible, not only for increased modeling agility at design-time but also for greater robustness and flexibility of enacting at run-time (see, e.g., [18] for a discussion about this and related problems).

The situation is further complicated when considering the class of security-sensitive workflows [2], i.e. when tasks in processes are executed under the responsibility of humans or software agents acting on their behalf. This means that, besides the usual execution constraints (specified by causal relations among tasks), there are authorization policies and constraints, i.e. the conditions under which users can execute tasks. Authorization policies are usually specified by using some variant of the Role Based Access Control (RBAC) model (see, e.g., [30]), while authorization constraints restrict which users can execute some set of tasks in a given workflow instance; an example is the Separation of Duty (SoD) constraint requiring two tasks to be executed by distinct users.

Since authorization policies and constraints may prevent the successful termination of the workflow, it is crucial to be able to establish if all tasks in the workflow can be executed satisfying the authorization policy without violating any authorization constraint, which is known as the Workflow Satisfiability Problem (WSP) [7]. In case of large and complex workflow specifications with expressive access control policies, detecting user assignments that may prevent the termination of a workflow becomes a computationally heavy task; the WSP is known to be NP-hard already in presence of one SoD constraint [31]. At run-time, the situation poses even more constraints on performance since at each new user request to execute a task, it is necessary to solve a new instance of the WSP by taking into account the history of the execution so far, i.e. which users have executed which tasks up to that instant (see, e.g., [3, 4]). Many of the available solutions to the WSP (such as [31, 3, 10, 17, 9]) do not provide practical tools capable of enabling designers of business processes to compose satisfiable workflows with authorization requirements. This ultimately prevents the development of efficient enactment mechanisms for security-sensitive workflows.

The modular design of business processes has been advocated for a long time in academia because of its support to reuse at design-time and scalability at run-time [21, 22]. In industry, it is more and more common to find solutions allowing

[*]This work has been partly supported by the EU under grant 317387 SECENTIS (FP7-PEOPLE-2012-ITN).

SACMAT '16, June 5–8, 2016, Shanghai, China.
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3802-8/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2914642.2914649

the reuse of (parts of) workflows to realize complex business processes. For instance, SAP Operational Process Intelligence[1] supports the creation of end-to-end business processes spanning multiple workflows. Such (template) workflows can be created once and stored to be then operated in different contexts. As an example, a *Purchase Order* workflow with tasks *Create Purchase Order* and *Create Invoice* would be part of any end-to-end business process selling goods, whereas a *Warehouse Management* workflow composed of tasks *Locate Product* and *Send Product* would be included only in cases where physical goods are involved.

Although techniques for modular specification and enactment of workflows and their impact have been extensively studied in the literature (see, e.g., [21, 22, 15]), the same is not true for security-sensitive workflows. In this special class of workflows, not only the control-flow spans several modules, but even authorization constraints may be defined across different components. Given the difficulties in specifying and enforcing execution and authorization constraints in this context, it is not surprising that vulnerabilities can be exploited by malicious users. For example, recently, the incorrect handling of authorization constraints between a *Purchase Order* and a *Warehouse Management* workflow allowed an Amazon employee to pay for cheap products and deliver expensive electronics to himself[2]. This kind of fraud could be avoided by specifying at design-time and enforcing at run-time a SoD constraint between tasks *Create Purchase Order* and *Send Product*.

To summarize, the modular specification and enactment of security-sensitive workflows is complicated by the lack of adequate answers to the following questions:

(i) how to specify authorization constraints that span multiple modules (inter-module constraints)?
(ii) how to enforce such constraints?
(iii) how to scale the enforcement mechanism and handle large workflows?
(iv) how to reuse already specified modules across processes?

In this paper, we introduce an approach capable of answering the questions above by making the following **contributions**:

- the definition of security-sensitive workflow components equipped with interfaces that allow to glue components together and define constraints between them (Section 3), to answer question (i);
- an automated technique, extending previous work [4], to synthesize run-time monitors from workflow components ensuring that all tasks can be executed without violating the policy or the constraints (Section 3.2), to answer question (ii);
- an experimental evaluation of our approach that clearly shows its viability and scalability (Section 4.1), to answer question (iii); and
- a description of a prototype implementing the reuse of workflow modules and monitors that can be integrated with industrial BPM systems (Section 4.2), to answer question (iv).

Section 2 presents the required background and an overview of the technique, while Section 5 discusses related work, a promising future direction of research, and concludes the paper.

[1]https://help.sap.com/hana-opint
[2]https://goo.gl/1bySZH

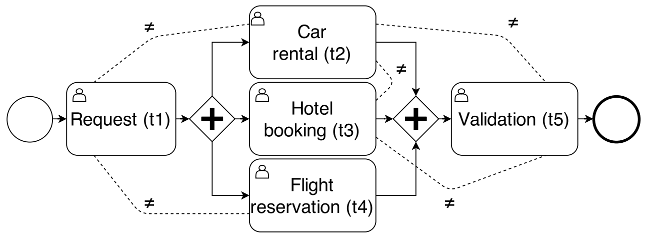


Figure 1: TRW in extended BPM notation

2. OVERVIEW

Our goal is to support the modular design and enactment of security-sensitive workflows and the synthesis of run-time monitors solving the WSP for these workflows. To introduce our approach, in this section we first recall the (non-modular) workflow specification and monitor synthesis technique from [4] (Section 2.1), and then we show the main intuitions and examples on how to extend the technique to support modular workflows (Section 2.2).

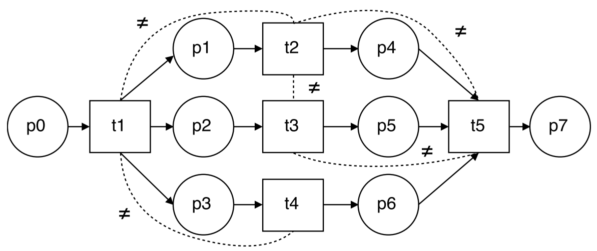
2.1 Workflow Specification and Monitor Synthesis

We illustrate our technique [4] for synthesizing a run-time monitor to solve the WSP by means of a simple example.

Example 2.1. The workflow shown in Figure 1 represents the Trip Request Workflow (TRW), whose goal is requesting trips for employees in an organization. It is composed of five tasks: Request ($t1$), Car rental ($t2$), Hotel booking ($t3$), Flight reservation ($t4$), and Validation ($t5$). Five SoD constraints must be enforced, i.e. the tasks in the pairs $(t1, t2)$, $(t1, t4)$, $(t2, t3)$, $(t2, t5)$, and $(t3, t5)$ must be executed by distinct users in any sequence of task executions of the TRW.

The workflow is specified in extended BPM Notation (BPMN) [20]. It contains two circles, the one on the left represents the start event (triggering the execution of the workflow), whereas that on the right the end event (terminating the execution of the workflow), tasks are depicted by labeled boxes, the constraints on the execution of tasks are shown as solid arrows (for sequence flows) and diamonds labeled by + (for parallel flows), the fact that a task must be executed under the responsibility of a user is indicated by the man icon inside a box, and SoD constraints as dashed lines labeled by $\neq$.

A simple situation in which the TRW can be deployed is a tiny organization with a set $U = \{a, b, c\}$ of three users and the following authorization policy $TA = \{(a, t1), (b, t1), (a, t2), (b, t2), (c, t2), (a, t3), (b, t3), (c, t3), (a, t4), (a, t5), (b, t5), (c, t5)\}$, where $(u, t) \in TA$ means that user u is entitled to execute task t. The organization would then like to know if there is a concrete execution that allows the process to terminate. Indeed, this is possible as shown by the following sequence of task-user pairs: $\eta = t1(b), t3(c), t4(a), t2(a), t5(b)$ where $t(u)$ means that user u has executed task t and the position in the sequence corresponds to the order in which the tasks have been executed (i.e. $t1$ has been executed first, $t5$ last, $t3$ after $t1$ but before $t4$, $t2$, and $t5$, etc). It is easy to check that the tasks in η are executed so that the ordering constraints on task execution are satisfied, each user u in each pair $t(u)$ of η is authorized to execute t since $(u, t) \in TA$, and each SoD constraint is



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

Figure 2: TRW as an extended Petri net (top) and as a transition system (bottom)

satisfied (e.g., tasks $t1$ and $t2$ are executed by the distinct users b and a, respectively).

Workflows can be represented at different levels of abstraction: in a modeling language like BPMN, which is suited for process designers but abstracts details of the semantics of the systems; as transition systems, which are amenable to formal analysis, but require descriptions much more detailed than what is usually provided by designers; and as Petri nets, which are often not familiar to designers, but have a formal semantics and are more intuitive than transition systems. The translation between these levels of abstraction can be done automatically (see, e.g., [28]), i.e. workflow designers can work at the BPMN level (e.g., the TRW in Figure 1) and a procedure, that is transparent to end-users, can translate them to Petri nets (e.g., top of Figure 2 for the Petri net corresponding to the BPMN in Figure 1) and then to transition systems (bottom of Figure 2 for the transition system corresponding to the Petri net at the top of the same figure), which are amenable to formal analysis.

We now briefly recall the technique in [4] to synthesize run-time monitors solving the WSP (i.e. enforcement mechanisms capable of finding a solution to the WSP). It takes as input the specification of a security-sensitive workflow (e.g., the BPMN in Figure 1 for the TRW) with the specification of an authorization policy TA and consists of two steps.
Off-line step.

Let $S = (V, Tr)$ be the (symbolic) transition system [26] derived from a security-sensitive workflow composed of a finite set T of tasks, a finite set U of users, and a finite set C of authorization constraints where V is the (finite) set of state variables and Tr is the (finite) set of transitions. The authorization policy TA is not taken into consideration in this step as we are able to synthesize a monitor for the WSP which can accommodate any such policy.

Example 2.2. To illustrate, let us consider the transition system at the bottom of Figure 2. The set V of state variables contains the (Boolean) control-flow variables p_i and d_{ti}, where p_i represents the existence of a token in the place pi of the Petri net at the top of the same figure and d_{ti} represents the fact that transition ti of the Petri net at the top of the same figure has been executed. Additionally, V contains the authorization variables a_{ti} and h_{ti} that are (Boolean) arrays such that $a_{ti}(u)$ means that user u is entitled to execute task ti and $h_{ti}(u)$ means that user u has executed task ti.

The transitions in Tr are listed in the table at the bottom of Figure 2 and are composed of three parts: an id(entifier), an enabling condition, and an effect. To illustrate, consider the second line of the table: the id indicates that user u executes task $t2$, the enabling condition is composed of two parts CF, which stands for control-flow, and $Auth$, which stands for authorization. The enabling condition CF is the conjunction of predicates $p1$ and $\neg d_{t2}$ indicating that, for this event to be enabled, there must be a token in place $p1$ of the Petri net (at the top of the same figure) and task $t2$ has not been executed yet. The enabling condition $Auth$ is the conjunction of predicates $a_{t2}(u)$, indicating that the user requesting to execute this task must be authorized to do so by the authorization policy (i.e. $(u, t) \in TA$), $\neg h_{t3}(u)$, indicating that the user requesting to execute this task should not have executed task $t3$ (notice that $t2$ and $t3$ can be executed in parallel, due to the gateway), and $\neg h_{t1}(u)$, indicating that the user requesting to execute this task should not have executed task $t1$ (notice that the SoD constraint between $t2$ and $t5$ is not present in $t2(u)$ because $t5$ is always executed after $t2$). The effect is also divided in a CF and an $Auth$ part. The effect of executing $t2$ at the control flow level (CF) is to remove a token from place $p1$ and put a token in place $p4$ (formally, this is done by setting $p1$ to False and $p4$ to True, as well as recording the execution of $t2$ by setting d_{t2} to True). The effect of executing $t2$ at the authorization level ($Auth$) is to update the history function h_{t2} to record the fact that $t2$ has been executed by user u (formally, $h_{t2}(u) := T$).

The transition system S is used to compute a *(symbolic) reachability graph* RG, i.e. a directed graph whose edges are labeled by task-user pairs in which users are symbolically represented by variables (called *user variables*) and whose nodes are labeled by a symbolic representation (namely, a formula of first-order logic) of the set of states from which it is possible to reach a state in which the workflow successfully terminates (for the TRW, this is the set of states in which all five tasks have been executed). A sequence $\eta_s = t_1(\upsilon_{j_1}), ..., t_n(\upsilon_{j_n})$ of task-user pairs is a *symbolic execution* where υ_{j_i} is a user variable with $1 \le j_i \le n$ and $i = 1, ..., n$. A *well-formed* path in RG is a path starting with a node without an incoming edge and ending with a node without an outgoing edge. The crucial property of RG is that the symbolic execution $\eta_s = t_1(\upsilon_{j_1}), ..., t_n(\upsilon_{j_n})$ collected while traversing one of its well-formed paths corresponds to an eligible (i.e. not violating any constraint in C) concrete execution $\eta_c = t_1(\mu(\upsilon_{j_1})), ..., t_n(\mu(\upsilon_{j_n}))$ for μ an injective function from the set $\Upsilon = \{\upsilon_{j_1}, ..., \upsilon_{j_n}\}$ of user variables (also called *symbolic users*) to the given set U of users (since μ is injective, distinct user variables are never mapped to the same user).

Example 2.3. An excerpt of the symbolic reachability graph for the TRW is depicted in Figure 3. The formulae labeling the nodes are not shown in the figure for the sake of simplicity. As an example, we show formula β_3, attached to node 3 of

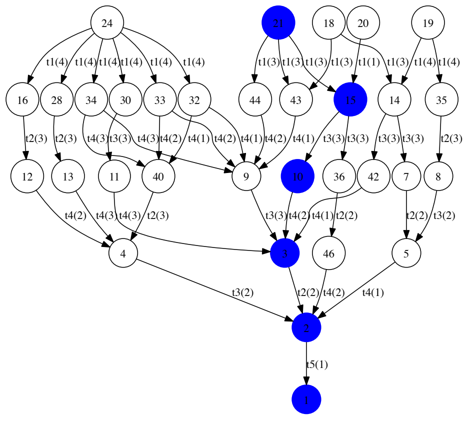


Figure 3: An excerpt of the symbolic reachability graph for the TRW

the graph, from which it is possible to execute $t2$:

$$\begin{array}{l}\neg p0 \wedge p1 \wedge \neg p2 \wedge \neg p3 \wedge \neg p4 \wedge p5 \wedge p6 \wedge \\ d_{t1} \wedge \neg d_{t2} \wedge d_{t3} \wedge d_{t4} \wedge \neg d_{t5} \wedge \\ a_{t2}(\upsilon 2) \wedge \neg h_{t1}(\upsilon 2) \wedge \neg h_{t3}(\upsilon 2) \wedge \\ a_{t5}(\upsilon 1) \wedge \neg h_{t3}(\upsilon 1) \wedge \neg h_{t2}(\upsilon 1) \wedge \upsilon 1 \neq \upsilon 2\end{array}$$

Formula β_3 encodes the fact that, in order for a user $\upsilon 2$ to be allowed to execute $t2$, the system must be in a state where there are tokens in places $p1$, $p5$, and $p6$ while there are no tokens in places $p0$, $p2$, $p3$, and $p4$ (first line), tasks $t1$, $t3$, and $t4$ have been already executed while tasks $t2$ and $t5$ have not been executed (second line), user $\upsilon 2$ should be authorized to perform $t2$ and should not have executed neither $t1$ nor $t3$ (third line), and there should exist a user $\upsilon 1$ (distinct from $\upsilon 2$) authorized to execute $t5$ who should have executed neither $t1$ nor $t3$ (last line).

Concerning the labels on the edges of the symbolic reachability graph, in Figure 3, a task-user pair $t(\upsilon_k)$ labeling an edge is abbreviated by $t(k)$ for the sake of compactness. So, for instance, the symbolic execution $\eta_s = t1(\upsilon_3), t3(\upsilon_3), t4(\upsilon_2), t2(\upsilon_2), t5(\upsilon_1)$ (cf. the well-formed path identified by the blue nodes in Figure 3) represents all those executions in which a symbolic user identified by υ_3 first performs task $t1$ followed by $t3$, then a symbolic user identified by υ_2 performs $t4$ and $t2$ in this order, and finally a symbolic user identified by υ_1 executes $t5$. If we apply an injective function μ from the set $\Upsilon = \{\upsilon_1, \upsilon_2, \upsilon_3\}$ of user variables to any finite set U of users (of cardinality at least three), the corresponding execution $\eta_c = \mu(\eta_s)$ is eligible according to the set C of SoD constraints shown in Figure 1.

The final action of the off-line step is to derive a non-recursive Datalog program M (with negation) from the symbolic reachability graph RG by generating a clause of the form $can\_do(\upsilon, t) \leftarrow \beta_\upsilon$ for each node υ in the graph RG.

Example 2.4. To illustrate, consider again node 3 in the graph as done in Example 2.3. Then, the Datalog program M will contain the following clause:

$$\begin{array}{ll} can\_do(t2, \upsilon 2) \leftarrow & \neg p0, p1, \neg p2, \neg p3, \neg p4, p5, p6, \\ & d_{t1}, \neg d_{t2}, d_{t3}, d_{t4}, \neg d_{t5}, \\ & a_{t2}(\upsilon 2), \neg h_{t1}(\upsilon 2), \neg h_{t3}(\upsilon 2), \\ & a_{t5}(\upsilon 1), \neg h_{t3}(\upsilon 1), \neg h_{t2}(\upsilon 1), \upsilon 1 \neq \upsilon 2. \end{array}$$

where the comma stands for logical conjunction.

On-line step. To build a run-time monitor for the WSP, we explain how to combine the Datalog program M obtained in the off-line step with the authorization policy TA. For this, the following observation is crucial. As shown in Example 2.4, the formula β_υ contains invocations to the binary predicates a_{ti} and h_{ti}. The former is the interface to the authorization policy and it is such that $a(u, t)$ holds iff $(u, t) \in TA$ while the latter keeps track of which user has executed which task, i.e. $h(t, u)$ means that t has been executed by u. Following an established tradition (see, e.g., [16]) claiming that (variants of) Datalog are adequate to express a wide range of access control policy idioms, we assume a to be defined by a Datalog program P. The predicate h is dynamic and defined by a set H of (ground) facts which is updated after each task execution. Thus, if the query $can\_do(u, t)$ can be derived from M, P, H (in symbols, $M, P, H \vdash can\_do(u, t)$), user u can execute task t and the workflow can terminate while satisfying the authorization policy and the authorization constraints.

Example 2.5. For the TRW, let us consider the relation TA presented after Example 2.1, which can be specified after the RBAC model [25] by the Datalog program P:

$$\begin{array}{l} ua(a, r1).\ ua(a, r2).\ ua(a, r3).\ ua(b, r2).\ ua(b, r3).\ ua(c, r2). \\ pa(r_3, t1).\ pa(r_2, t2).\ pa(r_2, t3).\ pa(r_1, t4).\ pa(r_2, t5). \\ a(\upsilon, \tau) \ \leftarrow \ ua(\upsilon, \rho),\ pa(\rho, \tau). \end{array}$$

where $r1$, $r2$, and $r3$ are roles, ua is the user-role assignment (cf. first line of facts), pa is the role-task assignment (cf. second line of facts), υ is a user variable, τ is a variable ranging over tasks, and a is defined as the join of the relations ua and pa (cf. Datalog clause in the last line). Notice that $P \vdash a(u, t)$ iff $(u, t) \in TA$ for user u and task t.

An example run of the monitor derived from the symbolic reachability graph in Figure 3 combined with the RBAC policy above is shown in Table 1: column 'History' shows which facts are added to the set H and column 'Answer' reports grant (deny, respectively) when the query in column 'Query' can (cannot, respectively) be derived from M, P, H. For instance, there are two denied requests: 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 no more be allowed to execute $t4$ because of the SoD constraint between $t1$ and $t4$ (see Figure 1); in line 4, 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 other requests are granted, as they violate neither task execution nor authorization constraints. The execution resulting from this run of the monitor is $t1(b), t3(c), t4(a), t2(a), t5(b)$, which is derived from the

Table 1: A run of the monitor program for the TRW

#	History	Query	Answer
0	$\emptyset$	$can\_do(a, t1)$	deny
1	-	$can\_do(b, t1)$	grant
2	$h(t1, b)$	$can\_do(c, t3)$	grant
3	$h(t3, c)$	$can\_do(a, t4)$	grant
4	$h(t4, a)$	$can\_do(b, t2)$	deny
5	-	$can\_do(a, t2)$	grant
6	$h(t2, a)$	$can\_do(b, t5)$	grant
7	$h(t5, b)$	-	-

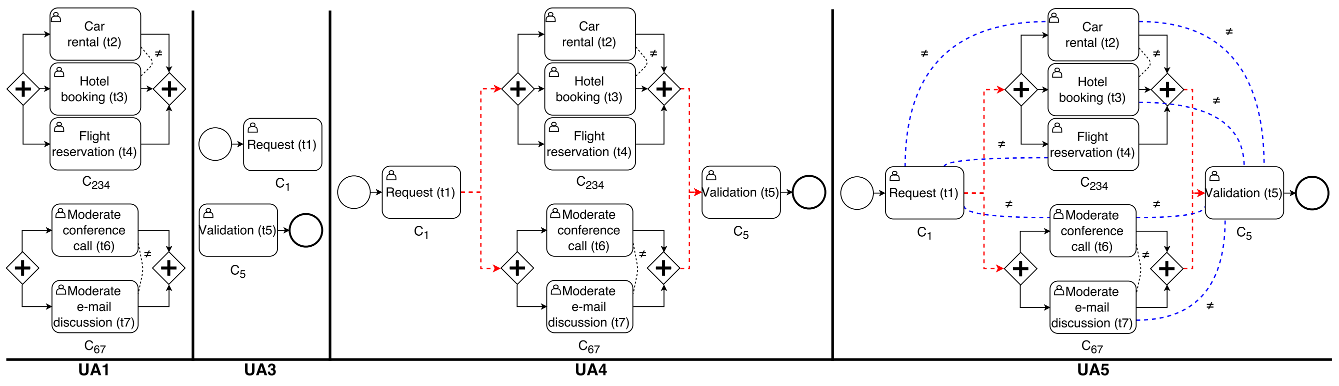
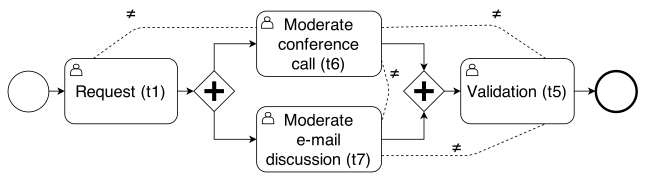


Figure 4: User actions necessary to specify and compose modules representing the TRW and MDW



Request (t1)
Moderate conference call (t6)
Moderate e-mail discussion (t7)
Validation (t5)
≠

Figure 5: MDW in extended BPM notation

symbolic execution $t1(v_1), t3(v_3), t4(v_2), t2(v_2), t5(v_1)$ in the graph of Figure 3 (cf. the path with the blue nodes; see also Example 2.3) by applying the injective function μ mapping v_1 to b, v_2 to a, and v_3 to c.

As described so far, our technique to synthesize run-time monitors for the WSP is based on representing each workflow with a single transition system. In [4], it is shown that monitor synthesis done this way is not scalable due to state space explosion. Additionally, the technique offers no support to the reuse and composition of (selected parts of) workflow specifications in larger workflows. We now explain how to extend this technique to handle a composition of (modular) workflows, making the approach scalable while fostering workflow reuse.

2.2 Modular Design and Enactment

We introduce our approach for the modular design and enactment of security-sensitive workflows by combining the previously introduced TRW with another example workflow.

Example 2.6. Figure 5 shows the Moderate Discussion Workflow (MDW) whose goal is to organize a discussion and voting process in an organization. It is composed of four tasks: Request ($t1$), Moderate Conference Call ($t6$), Moderate e-mail Discussion ($t7$), and Validation ($t5$). Four SoD constraints must be enforced: $(t1, t6)$, $(t6, t5)$, $(t6, t7)$, and $(t7, t5)$. Again, each task is executed under the responsibility of a user who is entitled to do so according to some authorization policy, which we leave unspecified for the sake of brevity and because the synthesis technique that we use generates a monitor that can accommodate any authorization policy (see Section 3.2 for details).

Notice that tasks $t1$ and $t5$ in Figures 1 and 5 are the same in both TRW and MDW. The notion of security-sensitive component introduced in this paper allows to reuse the specification of tasks $t1$ and $t5$ in different systems so that only the specification of the parallel execution of tasks $t2$, $t3$, and $t4$ for the TRW and $t6$ and $t7$ for the MDW must be developed from scratch.

By using the approach in this paper, a process designer can model both TRW and MDW by executing the following user actions, that are also depicted in Figure 4 (where the elements in black represent the internal specification of components, the red dashed arrows represent inter-component execution (control-flow) constraints and the blue dashed lines represent inter-component authorization constraints):

UA1 specify the parallel execution of tasks $t2$, $t3$, and $t4$ as a new component $\mathcal{C}_{234}$ for the TRW and of $t6$ and $t7$ as $\mathcal{C}_{67}$ for the MDW together with their authorization constraints, i.e. SoD between $t2$ and $t3$ for TRW and between $t6$ and $t7$ for MDW;

UA2 synthesize run-time monitors for the new components $\mathcal{C}_{234}$ and $\mathcal{C}_{67}$ to be stored (together with the monitors) in a repository for future use;

UA3 import, from the available workflow repository, the security-sensitive components containing tasks $t1$ and $t5$ in Figures 1 and 5, called $\mathcal{C}_1$ and $\mathcal{C}_5$, respectively;

UA4 define the control-flow among components; and

UA5 define inter-component authorization constraints.

As we will see, together with security-sensitive component specifications, in the workflow repository it is also possible to store the associated run-time monitors solving the run-time version of the WSP. To enact the modularly designed business processes TRW and MDW, the designer can simply add an authorization policy and deploy the process to the run-time environment. Behind the scenes, the monitors of the various components are automatically combined to build one for the composed processes, namely TRW and MDW. This combination is done by using a set G of "gluing assertions," which are logical assertions connecting the components, i.e. transferring control-flow and constraining the execution of tasks in the next components.

The main result of this paper (Theorem 3.1) shows that the combination of monitors M_1, M_{234}, and M_5 synthesized for components $\mathcal{C}_1$, $\mathcal{C}_{234}$ and $\mathcal{C}_5$, respectively, with their Datalog authorization policies P_1, P_{234}, P_5 and their execution histories H_1, H_{234}, H_5, and using the assertions in G, answers to user requests in the same way as a monitor M computed for the TRW as a single component. Formally, $M_1, M_{234}, M_5, G, P_1, P_{234}, P_5, H_1, H_{234}, H_5 \vdash \mathit{can\_do}(u, t)$ iff $M, P, H \vdash \mathit{can\_do}(u, t)$. Therefore, a similar run as the one shown in Table 1 for M can be obtained with M_1, M_{234}, M_5.

Indeed, the simplicity of the TRW and MDW spoils the advantages of a modular approach; the small dimension of the workflows allows us to keep the paper to a reasonable size. However, for large workflows—as we will see in Section 4—the advantages are substantial. To give an intuition of this, imagine replacing the tasks reused in both workflows, i.e. $t1$ and $t5$, with complex workflows: reusing their specifications and synthesized run-time monitors in larger workflows in which they are plugged, becomes much more interesting.

Each of the aforementioned user actions is based on the approach and notions introduced in the rest of the paper: **UA1** and **UA3** in Section 3; **UA4** and **UA5** in Section 3.1; and **UA2** in Section 3.2.

3. SECURITY-SENSITIVE WORKFLOW COMPONENTS

The goal of this Section is to identify a refinement of the notion of security-sensitive workflow, introduced in Section 2.1, that can be modularly composed with others through an appropriate interface. Technically, this is done by extending and partitioning the state variables of the transition system representing a security-sensitive workflow and then adding an appropriate notion of interface to support composition. The resulting notion is called a security-sensitive component. Below, we provide the key ideas underlying our techniques while omitting some of the formal details, which are available in a technical report [13].

Example 3.1. Preliminarily, we give some intuitions about the notion of security-sensitive component by considering the modular specification of both the TRW and MDW. Figure 6 shows the four components $\mathcal{C}_1$, $\mathcal{C}_{234}$, $\mathcal{C}_{67}$, and $\mathcal{C}_5$ whose composition gives both TRW and MDW. Each component is represented as a Petri net that is automatically derived from the BPMN model of Figure 4 (in a way similar to the one discussed in Section 2.1 to derive the Petri net at the top of Figure 2 from the BPMN model in Figure 1). The left side of the figure shows the extended Petri nets representing the four components: circles represent places, rectangles with a man icon transitions to be executed under the responsibility of users, rectangles without the icon transitions not needing human intervention, (black) dashed lines represent SoD constraints between tasks belonging to the same component, and (black) solid arrows the control flow in the same component. The right side of the figure shows how to connect these components in order to obtain the TRW and the MDW: (blue) dashed lines represent SoD constraints between tasks belonging to distinct components and (red) dashed arrows the control flow between two components.

The control flow between two components is outside of the semantics of extended Petri nets. For example, a token in place $p0$ of $\mathcal{C}_1$ goes to $p1$ of $\mathcal{C}_1$ after the execution of $t1$ and, at the same time a token is put in place $p1$ of $\mathcal{C}_{234}$ because of the (red) dashed arrow from $p1$ in $\mathcal{C}_1$ to $p0$ in $\mathcal{C}_{234}$ representing an inter-component execution constraint. When the token is in $p0$, the system executes the split transition s in C_{234} that removes the token from $p0$ and puts one in $p1$, $p2$, and $p3$ so that $t2$, $t3$, and $t4$ in $\mathcal{C}_{234}$ become enabled. Notice that the execution of $t2$ is constrained by a SoD constraint from task $t1$ in component C_1 (dashed arrow between $t1$ in C_1 and $t2$ in $\mathcal{C}_{234}$): this means that the user who has executed $t1$ in $\mathcal{C}_1$ cannot execute also $t2$ in $\mathcal{C}_{234}$.

Refined transition systems. Recall the description of the transition system $S = (V, Tr)$ associated to the TRW and derived from the Petri net at the top of Figure 2 given in Example 2.2. The state variables V can be partitioned in the following (disjoint) sets: P containing the Boolean variables pi's encoding the fact that a token is in place pi of the Petri net or not, D containing the Boolean variables d_{ti}'s encoding the fact that the task ti has been executed or not, A containing the Boolean arrays a_{ti}'s encoding the fact that a certain user is entitled to execute task ti or not, and H containing the Boolean arrays h_{ti}'s encoding the fact that a certain user has executed or not task ti. To support the definition of authorization constraints across components, we add a set C of Boolean arrays c_{ti}'s to the state variables of the transition system in order to represent SoD or BoD constraints (involving task ti) together with a set B of (so-called) *always constraints* fixing the values of the variables in C as Boolean combinations of the (history) variables in H. Formally, we assume B to contain a formula of the form $\forall u.v(u) \Leftrightarrow hst$, where v is in C, u is a variable ranging over users, and hst is a Boolean combination of atoms of the form $w(u)$ with $w \in H$. A *(refined) transition system* is a tuple of the form $((P, D, A, H, C), Tr, B)$ where the P, D, A, H, and C are sets of state variables, Tr is the set of transitions, and B is the set of always constraints.

Example 3.2. We refine the transition system $S = (V, Tr)$ in Example 2.2 as the tuple $((P, D, A, H, C), Tr', B)$ introduced above. The sets of state variables are defined as $P = \{p0, ..., p7\}$, $D = \{d_{t1}, ..., d_{t5}\}$, $A = \{a_{t1}, ..., a_{t5}\}$, $H = \{h_{t1}, ..., h_{t5}\}$, and $C = \{c_{t1}, ..., c_{t5}\}$. The set B of always constraints contains the formulae: $\forall v.c_{t1}(v) \Leftrightarrow T$, $\forall v.c_{t2}(v) \Leftrightarrow \neg h_{t1}(v) \wedge \neg h_{t3}(v)$, $\forall v.c_{t3}(v) \Leftrightarrow \neg h_{t2}(v)$, $\forall v.c_{t4}(v) \Leftrightarrow \neg h_{t1}(v)$, and $\forall v.c_{t5}(v) \Leftrightarrow \neg h_{t1}(v) \wedge \neg h_{t3}(v)$. The set Tr' contains the transitions shown in Table 2. The table at the bottom of Figure 2 can be derived from Table 2 by simply replacing each occurrence of the c_{ti}'s with the formula hst in the corresponding always constraint in B. While in this case the c_{ti}'s play the simple role of abbreviations, they are crucial to support the specification of authorization constraints spanning across components. This will be clear in Example 3.3 and Section 3.1 below, when considering the specification of the TRW as a composition of sub-modules.

The last observation above indeed holds in general: given a refined transition system $((P, D, A, H, C), Tr', B)$, it is always possible to build a transition system (V, Tr) by taking V as the union of P, D, A, H, C, and Tr to contain the transitions obtained by replacing each c_{ti} in Tr' with the corresponding hst in B. In this way, the classical interleaving semantics

Table 2: Refined transitions

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

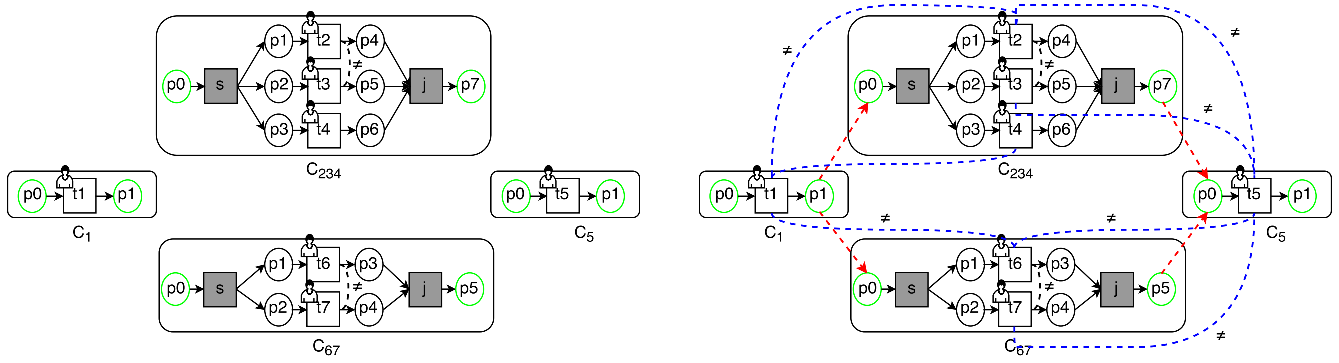


Figure 6: Security-sensitive components (left) and how to glue them together (right)

defined for (V, Tr) in [4] can also be adopted for the refined transition system $((P, D, A, H, C), Tr', B)$.

Adding the interface. A *security-sensitive (workflow) component* is a pair (S, Int) where $S = ((P, D, A, H, C), Tr, B)$ is a refined transition system and Int is its interface. Intuitively, Int identifies the variables of S whose values can be set by another component (called input variables, indicated by the super-script i) and those that are set by the component itself (called output variables, indicated by the super-script o). Formally, Int is a tuple of the form (A, P^i, P^o, H^o, C^i) where

- $P^i \subseteq P$ and each $p^i \in P^i$ is such that $p^i := T$ does not occur in the parallel assignments of an event in Tr,
- $P^o \subseteq P$ and each $p^o \in P^o$ is such that $p^o := T$ occurs in the parallel assignments of an event in Tr whereas $p^o := F$ does not,
- $H^o \subseteq H$, $C^i \subseteq C$, and
- only the variables in $(C \setminus C^i) \cup H^o$ can occur in a symbolic always constraint of B.

A is included in Int as the values of its variables are induced by the authorization policy TA specifying which users are entitled to perform which task. When P^i, P^o, H^o, and C^i are all empty, the security-sensitive component (S, Int) can only be interfaced with an authorization policy via the interface variables in A. The state variables in D are only used internally, to indicate that a task has been or has not been executed; thus, none of them is exposed in the interface Int. The variables in P, H, and C are local to S but some of them can be exposed in the interface in order to enable the combination of S with other components in a way which will be described in Section 3.1. The requirement that variables in P^i are not assigned the value T(rue) by any transition of the component allows their values to be determined by those in another component. Dually, the requirement that variables in P^o can only be assigned the value T(rue) by any transition of the component allows them to determine the values of variables in another component. Similarly to the values of the variables in P^i, those of the variables in C^i are fixed when combining the module with another; this is the reason for which only the variables in $C \setminus C^i$ can occur in the always constraints of the component.

Example 3.3. We now specify the interface of the components presented in Example 3.1. For components $\mathcal{C}_1$ and $\mathcal{C}_5$ (supporting **UA3**) we set $P^i_y := \{p0_y\}$, $P^o_y := \{p1_y\}$, $H^o_y := \{h_{ty}\}$ (for $y = 1, 5$), $C^i_1 := \emptyset$, and $C^i_5 := \{c^i_{t5}\}$ The interface of each component is the following: $p0_y$ is the input place, $p1_y$ is the output place, and the history variable h_{ty} can be used to constrain the execution of tasks in other components (for instance of $t2$ in the TRW as $t1$ and $t2$ are involved in a SoD). Notice that the execution of task $t1$ cannot be constrained by the execution of tasks in other components (thus $C^i_1 := \emptyset$) since $t1$ is always executed before all other tasks and cannot possibly be influenced by their execution. Contrarily, $C^i_5 := \{c^i_{t5}\}$, since $t5$ is always executed after all other tasks. In particular, c^i_{t5} will be defined so as to satisfy the SoD constraints between $t5$ and $t2$ or $t3$ for TRW and $t6$ or $t7$ for MDW. For component $\mathcal{C}_{234}$ (supporting **UA1**) we set $P^i_{234} := \{p0_{234}\}$, $P^o_{234} := \{p7_{234}\}$, $H^o_{234} := \{h_{t2}, h_{t3}\}$, and $C^i_{234} := \{c^i_{t2}, c^i_{t4}\}$. The interface of $\mathcal{C}_{67}$ is similar to that of $\mathcal{C}_{234}$. Section 3.1 below explains how components $\mathcal{C}_1$, $\mathcal{C}_{234}$, $\mathcal{C}_{67}$, and $\mathcal{C}_5$ can be "glued together" to build TRW and MDW.

3.1 Gluing Together Security-Sensitive Components

We now show how components can be combined together in order to build other, more complex, components. In the same way as the transition systems and interfaces presented above are derived from the internal elements of workflow components specified in BPMN, also the elements used to compose them (gluing assertions) are derived automatically from a high-level specification (e.g., the red arrows and blue lines in Figures 4 and 6).

Formally, for $l = 1, 2$, let (S_l, Int_l) be a security-sensitive component where $Int_l = (A, P^i_l, P^o_l, H^o_l, C^i_l)$ and $S_l = ((P_l, D_l, A_l, H_l, C_l), Tr_l, B_l)$ is such that P_1 and P_2, D_1 and D_2, A_1 and A_2, H_1 and H_2, C_1 and C_2 are pairwise disjoint sets. Furthermore, let $G = G_{\text{EC}} \cup G_{\text{Auth}}$ be a set of *gluing assertions over* Int_1 *and* Int_2, where G_{EC} is a set of formulae of the form $p^i \Leftrightarrow p^o$ for $p^i \in P^i_k$ and $p^o \in P^o_j$, called *inter execution constraints*; and G_{Auth} is a set of always constraints in which only the variables in $C^i_k \cup H^o_j$ may occur, for $k, j = 1, 2$ and $k \neq j$. Intuitively, the gluing assertions in G specify inter component constraints; those in G_{EC} specify how the control flow is passed from one component to another, whereas those in G_{Auth} specify authorization constraints across components, i.e. how the fact that a task in a component is executed by a certain user constrains the execution of a task in another component by a sub-set of the users entitled to do so.

The *security-sensitive component* (S, Int) *obtained by* *gluing* (S_1, Int_1) *and* (S_2, Int_2) *together with* G, in symbols $(S, Int) = (S_1, Int_1) \oplus_G (S_2, Int_2)$, is defined as $S = ((P, D, A, H, C), Tr, B)$ and $Int = (A, P^i, P^o, H^o, C^i)$, where

- $P = P_1 \cup P_2$, $D = D_1 \cup D_2$, $A = A_1 \cup A_2$, $H = H_1 \cup H_2$, $C = C_1 \cup C_2$,
- $Tr := [Tr_1]_{G_{\rm EC}} \cup [Tr_2]_{G_{\rm EC}}$ where $[Tr_j]_{G_{\rm EC}} := \{[tr_j]_{G_{\rm EC}} | tr_j \in Tr_j\}$,
- $B = B_1 \cup B_2 \cup G_{\rm Auth}$,
- $P^i = \{p \in (P_1^i \cup P_2^i) | p$ does not occur in $G_{\rm EC}\}$,
- $P^o = \{p \in (P_1^o \cup P_2^o) | p$ does not occur in $G_{\rm EC}\}$,
- $H^o = H_1^o \cup H_2^o$,
- $C^i = \{c \in (C_1^i \cup C_2^i) | c$ does not occur in $G_{\rm Auth}\}$,

and $[tr_j]_{G_{\rm EC}}$ is obtained from tr_j by adding the assignment $p^i := b$ if p^i is in P_j^i, there exists an inter execution constraint of the form $p^i \Leftrightarrow p^o$ in $G_{\rm EC}$, p^o is in P_k^o, and $p^o := b$ is among the parallel assignments of tr_j; otherwise, tr_j is returned unchanged, for $j, k = 1, 2$ and $j \neq k$.

The definition is well formed since S is obviously a security-sensitive transition system and *Int* satisfies all the structural constraints defined above.

Example 3.4. Let us consider components $\mathcal{C}_1$ and $\mathcal{C}_{234}$ of previous examples. We glue them together by using the following set $G = G_{\rm EC} \cup G_{\rm Auth}$ of gluing assertions where $G_{\rm EC} := \{p1_1 \Leftrightarrow p0_{234}\}$ and $G_{\rm Auth} := \{\forall u. c^i_{t2}(u) \Leftrightarrow \neg h_{t1}(u), \forall u. c^i_{t4}(u) \Leftrightarrow \neg h_{t1}(u)\}$. $G_{\rm EC}$ and $G_{\rm Auth}$ support **UA4** and **UA5**, respectively (see Figure 4). The inter execution constraint in $G_{\rm EC}$ corresponds to the dashed arrow connecting $p1$ in component $\mathcal{C}_1$ ($p1_1$) to $p0$ in component $\mathcal{C}_{234}$ ($p0_{234}$) in Figure 6. The always constraints in $G_{\rm Auth}$ formalize the dashed lines linking task $t1$ of component $\mathcal{C}_1$ to tasks $t2$ and $t4$ of component $\mathcal{C}_{234}$. Notice that $\mathcal{C}_1 \oplus_G \mathcal{C}_{234}$ can be combined with $\mathcal{C}_5$ to form a component corresponding to the TRW in Figure 1. This is possible by considering the following set $G' = G'_{\rm EC} \cup G'_{\rm Auth}$ of gluing assertions where $G'_{\rm EC} := \{p7_{234} \Leftrightarrow p0_5\}$ and $G'_{\rm Auth} := \{\forall u. c^i_{t5}(u) \Leftrightarrow \neg h_{t2}(u) \wedge \neg h_{t3}(u)\}$.

The operator $\oplus_G$ is both commutative and associative. This implies the desirable property that the result of combining components does not depend on the order in which these are imported (provided the execution and authorization constraints are unchanged).

3.2 Modular Synthesis of Run-time Monitors

Section 2.1 illustrates the methodology to automatically derive a monitor capable of solving the run-time version of the WSP for security-sensitive workflow systems. As argued above, the notions of security-sensitive workflow component and that of security-sensitive workflow system are equivalent. Below, let $\mathcal{M}$ be the function taking as input a security-sensitive component $S_C = ((P, D, A, H, C), Tr', B)$ and returning a security-sensitive transition system $S = (V, Tr)$ as explained at the end of Example 3.2. Based on this observation, we show how to turn the monolithic synthesis methodology of Section 2.1 into a modular one.

Let $\mathcal{RM}$ be the function taking as input a security-sensitive transition system $S = (V, Tr)$ and returning a Datalog program $\mathcal{RM}(S)$ defining a predicate $can\_do(u, t)$ such that user u can execute task t and the workflow S can successfully terminate. We define the function $\mathcal{RM}_c$ that takes as input a security-sensitive component $S_C = ((P, D, A, H, C), Tr, B)$ as follows: $\mathcal{RM}_c(S_C) = \mathcal{RM}(\mathcal{M}(S_c))$, i.e. first it transforms the security-sensitive component into a security-sensitive transition system and then applies the synthesis procedure of Section 2.1.

We now show how to reuse $\mathcal{RM}_c$ for the modular construction of run-time monitors for the WSP, i.e. we build a monitor for a composite component by combining those for their constituent components. Let $G = G_{\rm EC} \cup G_{\rm Auth}$ be a set of gluing assertions where $G_{\rm EC}$ is a set of inter execution constraints and $G_{\rm Auth}$ a set of always constraints over an interface (A, P^i, P^o, H^o, C^i), then $\langle G \rangle := \langle G_{\rm EC} \rangle \cup \langle G_{\rm Auth} \rangle$, where $\langle G_{\rm EC} \rangle := \{p^i \leftarrow p^o | p^i \Leftrightarrow p^o \in G_{\rm EC}\}$ and $\langle G_{\rm Auth} \rangle := \{c^i(u) \leftarrow hst^i(u) | \forall u. c^i(u) \Leftrightarrow hst^i(u) \in G_{\rm Auth}\}$. Intuitively, the shape of the Datalog clauses in $\langle G_{\rm EC} \rangle$ models how the execution flow is transferred from a component (that with an output place) to the other (that with an input place), while $\langle G_{\rm Auth} \rangle$ models how the execution of tasks in one component constrains the set of users who can execute tasks in another. Recall that, in Figures 4 and 6, the Datalog clauses in $\langle G_{\rm EC} \rangle$ are shown as dashed red arrows and those in $\langle G_{\rm Auth} \rangle$ as dashed blue lines.

Theorem 3.1. *Let (S_k, Int_k) be a security-sensitive component, $S_k = ((P_k, D_k, A_k, H_k, C_k), Tr_k, B_k)$, $\mathcal{H}_k$ is a set of (history) facts over H_k, and $\mathcal{P}_k$ a Datalog program (for the authorization policy) over A_k, for $k = 1, 2$. If G is a set of gluing assertions over Int_1 and Int_2, then $\mathcal{RM}(S), \mathcal{H}_1, \mathcal{H}_2, \mathcal{P}_1, \mathcal{P}_2 \vdash can\_do(u, t)$ iff $\mathcal{RM}(S_1), \mathcal{H}_1, \mathcal{P}_1, \langle G \rangle, \mathcal{RM}(S_2), \mathcal{H}_2, \mathcal{P}_2 \vdash can\_do(u, t)$, where $(S, Int) = (S_1, Int_1) \oplus_G (S_2, Int_2)$.*

The idea underlying the proof is that the monitors for the components are computed by considering all possible values for the variables in their interfaces. The additional constraints in the gluing assertions simply consider a sub-set of all these values by specifying how the execution flow goes from one component to the other and how the authorization constraints across components further constrain the possible executions of a component depending on which users have executed certain tasks in the other.

Theorem 3.1 supports action **UA2** in Section 2; additional applications are investigated in the section below.

4. APPLICATIONS

We have performed experiments focusing on two aspects: the scalability of the technique for the synthesis of run-time monitors for security-sensitive workflows (Section 4.1); and the design of a plug-in for the reuse of workflows and related run-time monitors inside an editor for security-sensitive workflows (Section 4.2).

4.1 Scalability

To show the practical scalability of our approach, we have performed a set of experiments with the random workflow generator from [4], which is capable of generating random security-sensitive workflows with an arbitrary number of tasks and composing them sequentially. For the experiments, we have generated components with a fixed size of 5 tasks and a varying number of constraints. The number of constraints is specified as a percentage (5%, 10% and 20%) of the number of tasks in each component for intra-component constraints and as a percentage of the total number of tasks for inter-component constraints[3]. Thus, in the configurations

[3]These configurations are taken from the experimental setup in [11], since the random generator was also based on the same work.

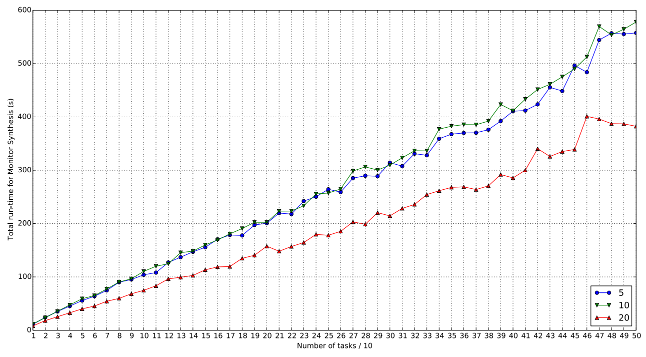


Figure 7: Time taken to synthesize a monitor varying with the number of tasks

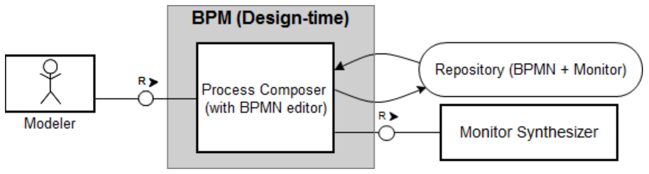
5% and 10% there are no intra-component constraints, while in the configuration 20% there is one for each component; e.g., for a workflow with 100 tasks, there are 5 inter-component constraints in the configuration 5%, 10 in the configuration 10% and 20 in the configuration 20%. The experiments have been conducted on a MacBook Air 2014 with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM running MAC OS X 10.10.2. The results are shown in Figure 7, in which the x-axis contains the total number of tasks in a workflow divided by 10 (the total number of components is the number in the x-axis times 2) and the y-axis shows the total time in seconds taken by the monitor synthesis procedure $\mathcal{RM}_c$ described above. Each data point is taken as the average of running $\mathcal{RM}_c$ 5 times for each configuration. Figure 7 suggests a linear (instead of the expected exponential!) behavior with respect to the number of tasks on this set of synthetic benchmarks. Indeed, without a modular approach the monitor synthesis technique is limited to only a few tasks, due to state space explosion.

We do not provide detailed timings for the on-line phase (i.e. how long does it take to answer authorization queries by the synthesized monitors) as these are under a second for workflows with (up to) 300 tasks and under 2 seconds for workflows (up to) 500 tasks. In case of multiple instances of the same workflows, we run a distinct (and independent) instance of the synthesized monitor; thus, the timings are similar to those of considering just one instance.

4.2 Reuse

It is possible to use the result of Theorem 3.1 to generate monitors for workflows specified as composed components and to reuse the monitor across different business processes (e.g., **UA3** in Section 2).

We have implemented the modular monitor synthesis technique as a tool composed of a single back-end developed in Python (ca. 3000 lines of code) and a series of front-ends which integrate with different BPM systems. The back-end takes as input specifications of modular workflows in the form of BPMN files (in XML), extended with support for authorization constraints. It derives a transition system for each component, the set of gluing assertions, and subsequently generates the corresponding run-time monitors, as described in Section 3.2. The front-end is responsible for calling the back-end and integrating with a workflow and monitor repository to store and retrieve the results of the back-end.



BPM (Design-time)
Modeler
R
Process Composer (with BPMN editor)
Repository (BPMN + Monitor)
R
Monitor Synthesizer

Figure 8: Architecture of a BPM design tool with a repository of models and monitors

To use the tool, process designers only have to input the graphical specification of the workflow and the constraints. The whole process of translating from BPMN to transition systems, applying the monitor synthesis procedure, storing, combining and retrieving monitors and workflows is automatic and happens behind the scenes. Thus, the tool does not require any knowledge besides BPMN modeling to be used.

The monitors that are stored in the repository are parametric wrt the user-task authorization relation, i.e. a synthesized monitor can accommodate several specifications of authorization policy (cf. Section 3.2). This allows secure enactment to be handled by simply composing together individual monitors.

An example front-end, integrated with the SAP HANA Workflow (HWF) engine was described in [6]. At design-time, execution constraints are modeled in BPMN and authorization constraints are input as part of the documentation of tasks, then HWF translates BPMN models into executable SQL procedures that are stored in the integrated HANA repository. The monitor synthesizer is invoked when a user calls the HWF compiler and it generates an SQL view[4] that is queried at run-time by the HWF execution engine. The monitor views are also stored in the HANA repository. Workflows and monitors can therefore be read from the repository and reused across deployments. At run-time, a synthesized monitor is invoked whenever a user tries to execute a task from the graphical user interface.

Our approach can be easily integrated into other existing BPM systems offering editors and repositories of business processes as outlined in the high-level architecture of Figure 8, where rectangles represent components, ovals represent storage systems, R-labeled links represent request/response communication between components and arrows represent access to storage. The *BPM* component represents existing solutions including a modeling environment for business processes, where the *Process Composer* sub-component contains a BPMN editor. The *Monitor Synthesizer* component implements the procedure described in Section 3.2 to compute (modular) monitors for workflow components and their composition modeled in the process composer. The *Repository* component represents a storage system for workflow models and synthesized monitors. Note that such repository may be part of the BPM solution or remotely located (e.g., Apromore [24]). The *modeler* interacts with the process composer with a request/response relation. The same relation exists between the process composer and the monitor synthesizer to request the synthesis of a run-time monitor for the BPMN model under specification. The process composer can store/retrieve BPMN models together with the synthesized

[4]Aggregation-free SQL and non-recursive Datalog are equivalent [1] and this translation is straightforward

monitors to/from the repository. The business process modeling and repository components in the proposed architecture are part of common reference architectures, e.g., [32].

5. CONCLUSIONS

The main contributions of the paper are (i) a modular approach to the synthesis of run-time monitors for (reusable) security-sensitive workflow components, (ii) the demonstration of the scalability of modular monitor synthesis by means of experiments, and (iii) a description of a tool integrating an editor with a repository of business processes (capable of storing run-time monitors) for business reuse. We regard these findings as the first significant step towards the development of efficient and practical enactment mechanisms for security-sensitive workflows, which go beyond the more theoretically oriented solutions to the WSP available in the literature.

5.1 Related and Future Work

Solutions to the WSP. Wang and Li [31] proposed a reduction of the WSP to SAT, which allows the use of off-the-shelf SAT solvers. The authors also showed that, with only equality and inequality relations, the WSP is fixed-parameter tractable (FPT) in the number of tasks. Later, Crampton et al. [10] improved the complexity bound for the WSP and extended the types of constraints for which it remains FPT. There are many works that take advantage of the FPT complexity result and design algorithms to solve the WSP with different kinds of constraints. The works in [8, 5] experimentally compare the results of FPT algorithms against those of a SAT solver on workflows of up to 30 tasks and conclude that FPT algorithms are better than those based on the SAT solver (the latter runs out of memory). [11] employs model checking on a fragment of LTL and experiments with workflows of up to 220 tasks.

On the practical side, our experiments show the scalability of our approach on workflows larger (up to 500 tasks) than those of the work above. On the theoretical side, the algorithms based on the use of SAT solving cannot cope with arbitrary authorization policies, while those based on FPT results must be invoked from scratch on "similar" instances of the WSP (a new instance is obtained from the previous one by asserting that a certain user has executed a given task at the previous step). Instead, our approach is capable of synthesizing a monitor for arbitrary authorization policies in the on-line phase and needs to keep track of which user has executed which tasks according to a given authorization policy at run-time, thereby significantly reducing the time to answer authorization requests, which—as pointed out in Section 4.1—remains under 2 seconds for workflows with 500 tasks. Even if algorithms based on SAT or FPT can be faster on smaller instances than our approach because the off-line phase requires to pre-compute all possible execution paths, our monitor synthesis approach offers the advantage of doing this just once and reuse the resulting monitors with arbitrary authorization policies while it is unclear (if possible at all) how this can be done with SAT- or FPT-based algorithms.
Resiliency. Workflow resiliency relates to the unavailability of users during the execution of a workflow instance. Mace et al. [17] discussed *quantitative* workflow resiliency, i.e. how likely a workflow is to terminate with an associated authorization policy and user availability model. Their solution uses Markov Decision Processes and exploits the off-line generation of an assignment tree, which shows all the possible executions of the workflow with different user assignments. The main difference between their assignment tree and our reachability graph is the fact that we use symbolic users, which allows us to accommodate different authorization policies at run-time. Crampton et al. [9] also considered a version of the problem called Valued WSP, where costs are associated to the violation of policies and constraints. A solution to the Valued WSP is an assignment of users to tasks with minimal cost and this problem was also shown to be FPT with user-independent constraints. It would be interesting to extend our approach to cope with resiliency or consider the costs of satisfying certain authorization constraints and synthesize risk-based monitors for the WSP. This is left as future work.
Modularity and reuse of workflow patterns. Reuse in Business Process Management has been an intense topic of research and industrial application; see, e.g., [14, 12]. Several works in the field of Petri nets have investigated modularity; see, e.g., [19]. None of these works addresses security issues as we do here. We observe that Theorem 3.1 may be used together with available techniques for decomposing large business processes; see, e.g., [23]. This would enable the synthesis of monitors that would be otherwise impossible to derive when considering monolithic processes because of the state-space explosion problem. Moreover, since workflows are built from basic control-flow patterns, see, e.g., [29, 15], a corollary of Theorem 3.1 is that it is possible to compute reachability graphs once for each basic security-sensitive workflow model, store the result, and modularly combine it with others along the lines of Section 3.2. In the following, we elaborate a bit on this idea according to which a workflow is seen as a combination of basic components (e.g., sequential, alternative and exclusive execution) that can be expressed by the gluing operator $\oplus$ introduced in Section 3.1.

For the sake of simplicity, we consider the *sequential*, *parallel*, and *alternative* composition of just two security-sensitive workflow components (S_1, Int_1) and (S_2, Int_2) (the generalization to n components is straightforward). We also assume that there is just one input and just one output place in both components (this is satisfied for the important class of workflow nets; see, e.g., [27]).
Sequential composition. It is sufficient to consider a set $G = G_{\text{EC}} \cup G_{\text{Auth}}$ of gluing assertions over Int_1 and Int_2 such that $G_{\text{EC}} = \{p_2^i \Leftrightarrow p_1^o\}$. Notice that $(S_1, Int_1) \oplus_G (S_2, Int_2) = (S_2, Int_2) \oplus_G (S_1, Int_1)$ but because the gluing assertion in G_{EC} is $p_2^i \Leftrightarrow p_1^o$, and not $p_2^o \Leftrightarrow p_1^i$, the process specified by component (S_1, Int_1) will always be executed before that specified by (S_2, Int_2).
Parallel composition. We need to preliminarily introduce two other components, each containing a single transition, one for splitting (*a*nd *s*plit) and one for joining (*a*nd *j*oin) the execution flow. The transitions are as follows:

$$\begin{aligned} Tr_{as} &:= \{p0_{as} \wedge \neg d_{as} \rightarrow p0_{as}, p1_{as}, p2_s, d_{as} := F, T, T, T\} \\ Tr_{aj} &:= \{q0_{aj} \wedge q1_{aj} \wedge \neg d_{aj} \rightarrow \\ &\qquad q0_{aj}, q1_{aj}, q2_{aj}, d_{aj} := F, F, T, T\} \end{aligned}$$

Then, it is sufficient to consider a set $G = G_{\text{EC}} \cup G_{\text{Auth}}$ of gluing assertions over Int_1, Int_2, Int_{as}, and Int_{aj} (recall that the gluing operator is associative) such that $G_{\text{EC}} = \{p1_{as} \Leftrightarrow p_1^i, p2_{as} \Leftrightarrow p_2^i, p_1^o \Leftrightarrow q0_{aj}, p_2^o \Leftrightarrow q1_{aj}\}$.
Alternative composition. Similarly to parallel composition, we need to introduce two other components, each containing two non-deterministic transitions (*o*r *s*plit and *o*r

$j$oin) to route the execution flow in one of the two components (S_1, Int_1) or (S_2, Int_2). The transitions are

$$Tr_{os} := \left\{ \begin{array}{l} p0_{os} \wedge \neg d_{os} \rightarrow \\ \quad p0_{os}, p1_{os}, p2_s, d_{os} := F, T, F, T, \\ p0_{os} \wedge \neg d_{os} \rightarrow \\ \quad p0_{os}, p1_{os}, p2_s, d_{os} := F, F, T, T \end{array} \right\}$$

$$Tr_{oj} := \left\{ \begin{array}{l} q0_{oj} \wedge \neg d_{oj} \rightarrow q0_{oj}, q2_{oj}, d_{oj} := F, T, T, \\ q1_{oj} \wedge \neg d_{oj} \rightarrow q1_{oj}, q2_{oj}, d_{oj} := F, T, T \end{array} \right\}$$

Then, it is sufficient to consider a set $G = G_{\text{EC}} \cup G_{\text{Auth}}$ of gluing assertions over Int_1, Int_2, Int_{os}, and Int_{oj} such that $G_{\text{EC}} = \{p1_{os} \Leftrightarrow p_1^i, p2_{os} \Leftrightarrow p_2^i, p_1^o \Leftrightarrow q0_{oj}, p_2^o \Leftrightarrow q1_{oj}\}$.

Tool development. We plan to integrate our prototype with other front-ends and perform extensive experiments concerning modularity and reuse of business processes available in repositories and libraries, such as the SAP Business Process Repository.[5]

6. ACKNOWLEDGMENTS

The authors would like to thank the constructive criticisms of anonymous reviewers and Charles Morisset.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Boston, 1995.

[2] A. Armando and S. E. Ponta. Model Checking of Security-sensitive Business Processes. In *Proc. of FAST*, 2009.

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

[4] C. Bertolissi, D. R. dos Santos, and S. Ranise. Automated synthesis of run-time monitors to enforce authorization policies in business processes. In *Proc. of ASIACCS*, 2015.

[5] D. Cohen, J. Crampton, A. V. Gagarin, G. Gutin, and M. Jones. Algorithms for the workflow satisfiability problem engineered for counting constraints. *CoRR*, abs/1504.02420, 2015.

[6] L. Compagna, D. R. dos Santos, S. E. Ponta, and S. Ranise. Cerberus: Automated synthesis of monitors for security-sensitive business processes. In *Proc. of TACAS*, 2016.

[7] J. Crampton. A reference monitor for workflow systems with constrained task execution. In *Proc. of SACMAT*, 2005.

[8] J. Crampton, A. V. Gagarin, G. Gutin, and M. Jones. On the workflow satisfiability problem with class-independent constraints. In *Proc. of IPEC*, 2015.

[9] J. Crampton, G. Gutin, and D. Karapetyan. Valued workflow satisfiability problem. In *Proc. of SACMAT*, 2015.

[10] J. Crampton, G. Gutin, and A. Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *TISSEC*, 16(1):4:1–4:31, June 2013.

[11] J. Crampton, M. Huth, and J. Kuo. Authorized workflow schemas: deciding realizability through LTL(F) model checking. *STTT*, 16(1):31–48, 2014.

[12] J. de Freitas. Model business processes for flexibility and re-use: A component-oriented approach. Technical report, IBM, 2009.

[13] D. R. dos Santos, S. Ranise, and S. E. Ponta. Modularity for Security-Sensitive Workflows. Technical report, arXiv, 2015. Available at http://arxiv.org/abs/1507.07479.

[14] A. Koschmider, M. Fellmann, A. Schoknecht, and A. Oberweis. Analysis of process model reuse: Where are we now, where should we go from here? *Decision Support Systems*, 66(0):9–19, 2014.

[15] C. Leuxner, W. Sitou, and B. Spanfelner. A formal model for work flows. In *Proc. of SEFM*, 2010.

[16] N. Li and J. C. Mitchell. Datalog with constraints: a foundation for trust management languages. In *Proc. of PADL*, 2003.

[17] J. C. Mace, C. Morisset, and A. Moorsel. Quantitative workflow resiliency. In *Proc. of ESORICS*, 2014.

[18] I. Markovic and A. C. Pereira. Towards a formal framework for reuse in business process modeling. In *Proc. of BPM*, 2008.

[19] O. Oanea. *Verification of Soundness and Other Properties of Business Processes*. PhD thesis, TU Eindhoven, 2007.

[20] OMG. Business Process Model and Notation, v2.0. Technical report, Object Management Group, 2011.

[21] H. Reijers and J. Mendling. Modularity in process models: Review and effects. In *Proc. of BPM*, 2008.

[22] H. Reijers, J. Mendling, and R. Dijkman. On the usefulness of subprocesses in business process models. Technical report, BPM Center, 2010.

[23] H. Reijers, J. Mendling, and R. Dijkman. Human and automatic modularizations of process models to enhance their comprehension. *Inf. Syst.*, 36(5):881 – 897, 2011.

[24] M. L. Rosa, H. Reijers, W. van der Aalst, R. Dijkman, J. Mendling, M. Dumas, and L. Garca-Banuelos. Apromore: An advanced process model repository. *Expert Syst. Appl.*, 38(6):7029–7040, 2011.

[25] R. Sandhu, E. Coyne, H. Feinstein, and C. Youmann. Role-Based Access Control Models. *IEEE Computer*, 2(29):38–47, 1996.

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

[27] W. van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In *Proc. of BPM*, 2000.

[28] W. van der Aalst and A. ter Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30:245–275, 2003.

[29] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.

[30] J. Wainer, A. Kumar, and P. Barthelmess. Dw-rbac: A formal security model of delegation and revocation in workflow systems. *Inf. Syst.*, 32(3):365–384, May 2007.

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

[32] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, Secaucus, 2007.

[5]https://implementationcontent.sap.com/bpr