Obstruction-Free Authorization Enforcement

Aligning Security and Business Objectives

Author(s):
Basin, David A.; Burri, Samuel J.; Karjoth, Günter

Publication date:
2011

Permanent link:
https://doi.org/10.3929/ethz-a-006528705

Originally published in:
https://doi.org/10.1109/CSF.2011.14

To appear in:
Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF '11).
June 27–29, 2011, Domaine de l'Abbaye des Vaux de Cernay, France.

Obstruction-free Authorization Enforcement: Aligning Security and Business Objectives

David Basin*
* *ETH Zurich*
Information Security Group
E-Mail: basin@inf.ethz.ch

Samuel J. Burri*† and Günter Karjoth†
† *IBM Research – Zurich*
Security Group
E-Mail: {sbu, gka}@zurich.ibm.com

***Abstract*—Access control is fundamental in protecting information systems but it also poses an obstacle to achieving business objectives. We analyze this tradeoff and its avoidance in the context of systems modeled as workflows restricted by authorization constraints including those specifying Separation of Duty (SoD) and Binding of Duty (BoD).**

To begin with, we present a novel approach to scoping authorization constraints within workflows with loops and conditional execution. Afterwards, we consider enforcement's effects on business objectives. We identify the notion of *obstruction*, which generalizes deadlock within a system where access control is enforced, and we formulate the existence of an obstruction-free enforcement mechanism as a decision problem. We present lower and upper bounds for the complexity of this problem and also give an approximation algorithm that performs well when authorizations are equally distributed among users.

I. Introduction

As is well known, security often conflicts with other system-design objectives. Take the case of a business system where business objectives are modeled by workflows defining the tasks executed by users. Adding access control to this system prevents unauthorized task executions, but may also have unintended consequences. For example, the resulting system may deadlock or be obstructed in that fewer options are available to achieve the workflow's business objectives than were originally designed. A fundamental problem is how this conflict can be resolved. Can authorizations be enforced without obstructing system objectives?

In this paper, we investigate this question by modeling workflow-based systems at two levels of abstraction. At the *control-flow level*, a workflow models the temporal ordering and causal dependencies of a set of tasks that together implement a business objective. The *task-execution level* refines the control-flow level and also models who executes which task. The above question can be formalized as whether authorizations are enforceable at the task execution level without changing the workflow at the control-flow level.

Consider as an example a simple workflow with three tasks t_1, t_2, and t_3, illustrated as a labelled transition system W in the top half of Figure 1. At the control-flow level, a successful workflow execution specifies the business objective of executing t_1 and afterwards either t_2 or t_3. Now consider an authorization policy stating that user u_1 may execute all three tasks and u_2 may execute only t_1. Furthermore, t_1 and t_2 must not be executed by the same user. The bottom half of Figure 1 shows two refinements, W_1 and W_2, of W that respect this authorization policy, where we write $t.u$ to indicate that u executes t. In W_1, u_1 may execute t_1 but afterwards only t_3 is executable without violating the authorization policy. This, however, corresponds to a restriction of the workflow at the control flow level (indicated by the jagged arrow). We call this situation an *obstruction*. In contrast, W_2 avoids obstructions by being more restrictive than W_1 and not allowing u_1 to execute t_1.

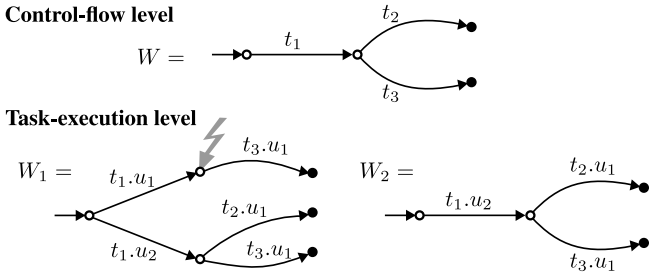

Figure 1. Enforcement with and without obstruction

This simple example illustrates the tension between security and business objectives and suggests that authorization enforcement should be designed in a way that aligns both objectives. Our underlying assumption is that for achieving business objectives, it does not matter who is executing a task as long as every task can be executed by an authorized user. As illustrated by the example, we thereby give the preservation of a workflow at the control-flow level priority over the choice of who can execute a task.

Concretely, we consider three classes of authorization constraints: We call the workflow-independent permissions of users to execute tasks, *e.g.* modeled in RBAC [4], *static authorizations*. We augment them with dynamic *Separation of Duty (SoD)* and dynamic *Binding of Duty (BoD)* con-

straints that are workflow-specific and depend on who has executed previous tasks. SoD, also known as the four-eyes-principle, aims at reducing fraud and errors by preventing a user from executing tasks with conflicting interests. BoD is dual to SoD and aims at reusing existing knowledge and preventing widespread dissemination of sensitive information by restricting the execution of two related tasks to a single user. These classes of constraints are recommended as best practice by frameworks like Cobit [9] and regulations like SOX [1] and are commonplace in regulated business environments, such as the financial industry.

We proceed as follows. First, we formalize workflows and authorization constraints as CSP processes [14] and model authorization enforcement as parallel, synchronized composition. All our results are formulated in terms of CSP. However, to bridge the gap to higher-level workflow languages, we visualize workflows using the *Business Process Modeling Notation (BPMN)* [11], a well-established workflow modeling language, and we propose an extension to BPMN to visualize our SoD and BoD constraints. We use a running example and our BPMN visualization to illustrate the applicability of our approach to realistic business cases. Second, we formulate the existence of an obstruction-free enforcement mechanism for a given set of authorization constraints and a workflow as a decision problem, which we call the *enforcement process existence* (**EPE**) problem. Finally, we present algorithms both to solve and approximate **EPE** and we analyze their runtime complexity.

Our first contribution is the formalization and analysis of obstruction-free authorization enforcement in workflow systems. We thereby generalize the notion of deadlock-freedom of a process to also includes cases where progress of the workflow execution is possible although with fewer options than are specified at the control-flow level. We prove that **EPE** is decidable, however **NP**-hard. Furthermore, we show that our approximation algorithm has a polynomial runtime complexity and provides good approximation results when the set of users is large and the static authorizations are equally distributed among them.

Our second contribution is a novel approach to modeling SoD and BoD constraints that are scoped to subsets of task instances. Our formalism imposes no restrictions on the expressiveness of the underlying workflow modeling language. In particular, workflows may contain loops and conditional executions, which are usually omitted in existing formalisms. The visualization of our constraints paves the way to integrating our modeling approach in a graphical workflow modeling tool. This enables business experts to extend workflow models with authorization constraints that are enforceable without introducing obstructions.

The remainder of this paper is organized as follows. In Section II we provide background on both CSP and BPMN. In Section III, we model workflows and authorization constraints using CSP and visualize them in BPMN. We define obstruction-free authorization enforcement in Section IV. Based on this definition, we introduce **EPE** and analyze its complexity. In Section V, we present approximation algorithms for **EPE**. We review related work in Section VI and draw conclusions in Section VII. The appendix provides proofs and additional background on CSP and graph coloring, which we use in our reductions.

II. Background

A. CSP

We use a subset of Hoare's process algebra CSP [14] to model the specification and enforcement of authorization constraints on workflows. CSP describes a system as a set of communicating *processes*. A process is referred to by a *name*; let $\mathcal{N}$ be the set of all process names. Processes communicate with each other by concurrently engaging in *events*. Σ is the set of all regular events. In addition, there are two special events: τ, a process-internal, hidden event, and $\checkmark$ that communicates successful termination. Let $C \subseteq \Sigma$. We write C^τ for $C \cup \{\tau\}$, $C^\checkmark$ for $C \cup \{\checkmark\}$, and $C^{\tau,\checkmark}$ for $C \cup \{\checkmark, \tau\}$. In particular, $\Sigma^{\tau,\checkmark}$ is the set of all events.

A *trace* is a sequence of regular events, possibly ending with the special event $\checkmark$. $\langle\rangle$ is the empty trace and $\langle\sigma_1, \ldots, \sigma_n\rangle$ is the trace containing the events σ_1 to σ_n, for $n \geq 1$. For two traces i_1 and i_2, their concatenation is denoted $i_1 \hat{} i_2$. C^* is the set of all finite traces over C and its superset $C^{*\checkmark} = C^* \cup \{i \hat{} \langle\checkmark\rangle \mid i \in C^*\}$ includes all traces ending with $\checkmark$. We abuse the set-membership operator $\in$ for traces and write $\sigma \in i$ for an event σ and a trace i, if there exist two traces i_1 and i_2 such that $i = i_1 \hat{} \langle\sigma\rangle \hat{} i_2$.

For an event $\sigma \in \Sigma$ and a name $n \in \mathcal{N}$, the set of processes $\mathcal{P}$ is inductively defined by the grammar $\mathcal{P} ::= \sigma \to \mathcal{P} \mid SKIP \mid STOP \mid n \mid \mathcal{P} \,\Box\, \mathcal{P} \mid \mathcal{P} \sqcap \mathcal{P} \mid \mathcal{P} \parallel \mathcal{P} \mid \mathcal{P} \,|||\, \mathcal{P} \mid \mathcal{P} \,;\, \mathcal{P}$.

There are different approaches to formally describing the behavior of a process. CSP's denotational semantics describes a process P as a prefix-closed set of traces $\mathsf{T}(P) \subseteq \Sigma^{*\checkmark}$, called the *traces model*. The operational semantics describes P as a *labelled transition system (LTS)*. We call a process *finite* if it corresponds to an LTS with finitely many states and input symbols. The two semantics are compatible. Because we mainly use the traces model, we describe in the following the process composition operators, introduced above, in terms of the denotational semantics. We review the operational semantics, which we use in some proofs, in Appendix A.

Let $P, P_1, P_2 \in \mathcal{P}$ be processes. The process $\sigma \to P$ *engages* in the event σ first and behaves like P afterward. Formally, $\mathsf{T}(\sigma \to P) = \{\langle\sigma\rangle \hat{} i \mid i \in \mathsf{T}(P)\} \cup \{\langle\rangle\}$ This notation can be extended. For $C \subseteq \Sigma$, the expression $\sigma : C \to P$ represents a process that engages in a $\sigma \in C$ first and behaves like P afterward. $SKIP$ engages in $\checkmark$ and no further event afterward; $\mathsf{T}(SKIP) = \{\langle\rangle, \langle\checkmark\rangle\}$. $STOP$ represents the process that does not engage in any event;

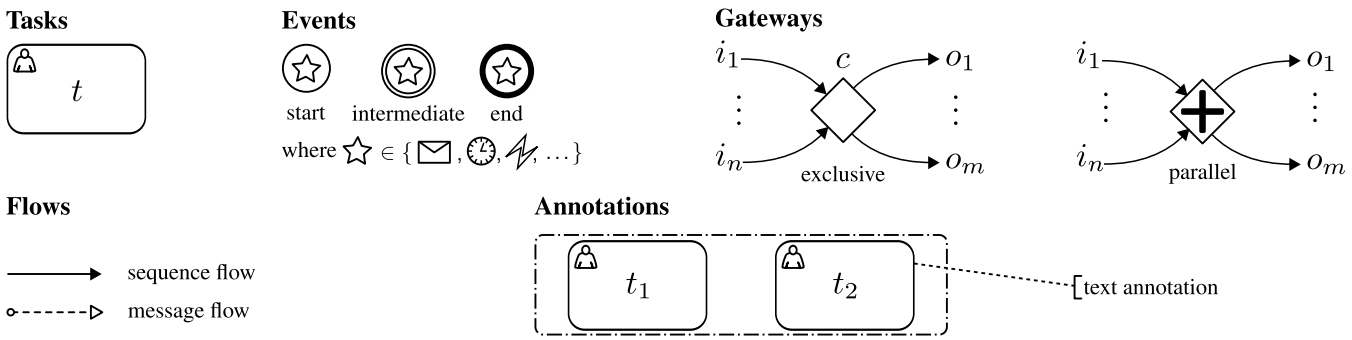

Figure 2. BPMN artifacts

$\mathsf{T}(STOP) = \{\langle\rangle\}$. In other words, $SKIP$ represents successful termination and $STOP$ a deadlock. We write $n = P$ to assign P to the name n; the process n behaves like P. The process $P_1 \,\Box\, P_2$ represents the *external* choice and $P_1 \sqcap P_2$ the *internal* choice between P_1 and P_2. With respect to the traces model, $P_1 \,\Box\, P_2$ and $P_1 \sqcap P_2$ are indistinguishable, namely $\mathsf{T}(P_1 \,\Box\, P_2) = \mathsf{T}(P_1 \sqcap P_2) = \mathsf{T}(P_1) \cup \mathsf{T}(P_2)$. The failures model explained below distinguishes between the two processes. The process $P_1 \parallel P_2$ represents the parallel and (fully-)synchronized composition of P_1 and P_2. It engages in an event σ if both P_1 and P_2 synchronously engage in σ; $\mathsf{T}(P_1 \parallel P_2) = \mathsf{T}(P_1) \cap \mathsf{T}(P_2)$. Similarly, the process $P_1 \,|||\, P_2$ is the parallel, unsynchronized composition of P_1 and P_2. It engages in σ if either P_1 or P_2 engage in σ; $\mathsf{T}(P_1 \,|||\, P_2)$ is the set of all interleavings of i_1 and i_2 for $i_1 \in \mathsf{T}(P_1)$ and $i_2 \in \mathsf{T}(P_2)$. The process $P_1 \,;\, P_2$ denotes the sequential composition of P_1 and P_2. It first behaves like P_1. Upon successful termination of P_1, the event $\checkmark$ is hidden, which is denoted by the invisible event τ. Afterwards, the process behaves like P_2. Formally, $\mathsf{T}(P_1 \,;\, P_2) = (\mathsf{T}(P_1) \cap \Sigma^*) \cup \{i_1 \hat{} i_2 \mid i_1 \hat{} \langle\checkmark\rangle \in \mathsf{T}(P_1), i_2 \in \mathsf{T}(P_2)\}$. Note that the invisible event τ does not appear in traces, similar to ε-transitions in nondeterministic automata. For a trace i, $P \setminus i$ represents the process P after engaging in all events in i. If $\mathsf{T}(P_1) \subseteq \mathsf{T}(P_2)$, then P_1 is a *trace refinement* of P_2, denoted $P_2 \sqsubseteq_{\mathsf{T}} P_1$. If $P_2 \sqsubseteq_{\mathsf{T}} P_1$ and $P_1 \sqsubseteq_{\mathsf{T}} P_2$, then P_1 and P_2 are *trace equivalent*, denoted $P_1 =_{\mathsf{T}} P_2$.

The traces model is insensitive to nondeterminism. It describes what a process *can* do but not what it *may refuse* to do. The *failures model* F is a refinement of the traces model that overcomes this shortcoming. Let P be a process. P's *refusal set* is a set of events all of which P can refuse to engage in and $rs(P) \subseteq 2^{\Sigma^\checkmark}$ is the set of all refusal sets of P. The set of *failures* of P is then $\mathsf{F}(P) = \{(i, C) \mid i \in \mathsf{T}(P), C \in rs(P \setminus i)\}$. For two processes P_1 and P_2, P_1 is a *failure refinement* of P_2, written $P_2 \sqsubseteq_{\mathsf{F}} P_1$, if $\mathsf{F}(P_1) \subseteq \mathsf{F}(P_2)$. Furthermore, P_1 is *failure equivalent* to P_2, written $P_1 =_{\mathsf{F}} P_2$, if $P_1 \sqsubseteq_{\mathsf{F}} P_2$ and $P_2 \sqsubseteq_{\mathsf{F}} P_1$. A more detailed definition of refusal sets and failures is given in Appendix A.

For a relation $R \subseteq \Sigma \times \Sigma$ and a process P, $P[R]$ denotes P *renamed* by R. For every tuple $(\sigma_1, \sigma_2) \in R$, $P[R]$ engages in σ_2 if P engages in σ_1.

B. BPMN

We introduce a subset of the *Business Process Modeling Notation (BPMN)* [11] that we later extend to model authorization constraints for workflows. BPMN describes workflows at a high level of abstraction using a graph-like notation. We distinguish five kinds of BPMN artifacts, illustrated in Figure 2. *Tasks* are modeled by rectangles with rounded corners, labelled with the name of the task. A small icon in the upper left corner may specify the task's type. In this paper, we consider only tasks executed by humans, called *user task* in BPMN and denoted by an icon depicting a person.

An *event* models the occurrence of a condition or an interaction with the environment. Events are circle-shaped. Their exterior boundary indicates whether their occurrence triggers a workflow instantiation, called a *start event*, whether they occur during the workflow's execution, called an *intermediate event*, or whether their occurrence terminates a workflow instance, called an *end event*. Furthermore, an event's interior may contain an icon, which determines the event's type. Examples are the arrival of a message or the expiration of a deadline, illustrated by an envelope and a clock, respectively.

Flows describe a workflow's control-flow. A *sequence flow*, illustrated by a solid line with an arrow, defines the order in which tasks are executed and events occur. BPMN has other flow artifacts, such as message flows, but we only make use of sequence flows.

Merging and branching of the control-flow is modeled by *gateways*. A gateway has $n \geq 1$ incoming and $m \geq 1$ outgoing sequence flows. *Exclusive gateways* are depicted by an empty (or with an x labeled) diamond. Whenever the control-flow reaches an exclusive gateway on an incoming sequence flow, it passes the control-flow immediately on to exactly one of the m outgoing sequence flows, based

on the evaluation of the condition c associated with the gateway. *Parallel gateways* are illustrated by a diamond labeled with the symbol "+". They synchronize the control flow on the n incoming sequence flows and spawn the concurrent execution on the m outgoing sequence flows.

BPMN models can be annotated. For example, tasks may have textual annotations as illustrated in Figure 2. Sets of tasks are defined by placing them in a dot-dashed box.

III. Authorization-constrained Workflows

We use CSP to formalize workflows, authorization constraints, and their interplay. CSP's notion of renaming facilitates a mapping between the control-flow and the task-execution level. Furthermore, its notion of parallel, synchronized process execution enables a concise description of workflow systems that are composed from multiple subprocesses, each modeling a separate system aspect. Concretely, we first describe workflows and our three classes of authorization constraints as individual processes. Afterwards, we describe the overall workflow system as the parallel, synchronized composition of these processes. In addition, we describe the visualization of our constraints using an extension of BPMN.

A. Workflows

There are numerous translations from BPMN and similar workflow modeling languages to process calculi such as CSP [21] or the π-calculus [12]. The technical differences are unimportant for our work here and we use a straightforward translation to CSP, illustrated in our running example.

For the reminder of this paper, assume a set of *tasks* $\mathcal{T}$ and a set of *points* $\mathcal{O}$. Points are used to model BPMN events. We now describe workflows at the control-flow level using CSP.

Definition 1 (Workflow process) *A* workflow process *is a process* W *such that* $\mathsf{T}(W) \subseteq (\mathcal{T} \cup \mathcal{O})^{*\checkmark}$.

In other words, a workflow process may engage in tasks, points, and finally the event $\checkmark$. We give below an example workflow, modeled in BPMN, and a corresponding workflow process. We will use this workflow as a running example to illustrate the concepts presented in this paper.

Example 1 (Collateral Evaluation Workflow) The financial industry distinguishes between secured and unsecured loans. In a secured loan, the borrower pledges some asset, such as a house or a car, as collateral for his debt. If the borrower defaults, the creditor takes possession of the asset to mitigate his financial loss.

Figure 3 shows a BPMN model of the *collateral evaluation workflow*, which we adopted from IBM's Information FrameWork [8]. Ignore the grey BPMN elements for the moment. This workflow is executed by a financial institution to evaluate, accept, and prepare the safeguarding of the collateral that a borrower pledges in return for a secured loan.

For this example, let $\mathcal{T} = \{t_1, \ldots, t_5\}$ where t_1 refers to Compute Market Value, t_2 to Control Computation, etc., and $\mathcal{O} = \{o_1, o_2, o_3\}$, as shown in Figure 3. The workflow process W models the collateral evaluation workflow in CSP.

$$W = (P_1 \,|||\, P_2)\ ;\ (t_5 \to ((o_2 \to W) \sqcap SKIP))$$
$$P_1 = t_1 \to t_2 \to ((o_1 \to P_1) \sqcap SKIP)$$
$$P_2 = o_3 \to t_3 \to ((t_4 \to SKIP) \sqcap SKIP)$$

We do not model data-flow in our example and therefore overapproximate gateway decisions with CSP's operator $\sqcap$ (internal choice). ◇

Next, we model the execution of tasks by users, workflow instances, and workflows at the task-execution level. For the reminder of this paper, let $\mathcal{U}$ be the set of *users*. For a task t and a user u, we call an event of the form $t.u$ a *(task) execution event* and denote by $\mathcal{X} = \{t.u \mid t \in \mathcal{T}, u \in \mathcal{U}\}$ the set of all execution events. We introduce the auxiliary relation $\pi = \{(t.u, t) \mid t \in \mathcal{T}, u \in \mathcal{U}\}$, which maps every execution event $t.u$ to the task t. The process $W[\pi^{-1}]$ then models the workflow process W at the task-execution level. It engages in the execution event $t.u$, for any $u \in \mathcal{U}$, if the workflow process W engages in t. The application of π^{-1},

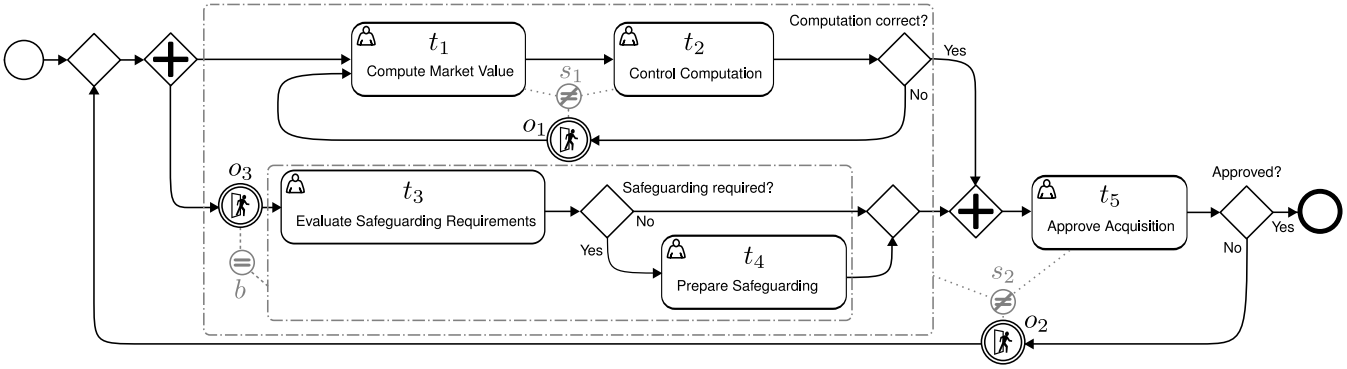

Figure 3. Collateral Evaluation Workflow

$$
\begin{aligned}
i_1 &= \langle t_1.\mathsf{Alice},\ t_2.\mathsf{Bob},\ t_4.\mathsf{Claire}\rangle \\
i_2 &= \langle t_1.\mathsf{Alice},\ o_3,\ t_3.\mathsf{Bob},\ t_2.\mathsf{Alice},\ o_1,\ t_1.\mathsf{Bob},\ t_2.\mathsf{Claire},\ t_5.\mathsf{Claire},\ \checkmark\rangle \\
i_3 &= \langle t_1.\mathsf{Alice},\ o_3,\ t_3.\mathsf{Bob},\ t_2.\mathsf{Bob},\ o_1,\ t_1.\mathsf{Alice},\ t_4.\mathsf{Dave},\ t_2.\mathsf{Claire},\ t_5.\mathsf{Claire},\ \checkmark\rangle \\
i_4 &= \langle t_1.\mathsf{Alice},\ o_3,\ t_3.\mathsf{Bob},\ t_2.\mathsf{Bob},\ o_1,\ t_1.\mathsf{Bob},\ t_4.\mathsf{Bob},\ t_2.\mathsf{Claire},\ t_5.\mathsf{Dave},\ \checkmark\rangle
\end{aligned}
$$

Figure 4. Examples of workflow traces

the inverse of π, to W has no effect on points and $\checkmark$; *i.e.* if W engages in a point or $\checkmark$, then so does $W[\pi^{-1}]$. Note that we will abuse the renaming notation to map a trace $i \in \mathsf{T}(W[\pi^{-1}])$ to a trace $i[\pi] \in \mathsf{T}(W)$.

Definition 2 (Workflow trace) *A* workflow trace *is a trace* $i \in (\mathcal{X} \cup \mathcal{O})^{*\checkmark}$.

A workflow trace models a workflow instance. In particular, if $i \in \mathsf{T}(W[\pi^{-1}])$, then i models an instance of the workflow modeled by W. We say the workflow instance modeled by i has *successfully terminated* if $\checkmark \in i$.

Example 2 (Workflow traces) Let $\mathcal{U} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, \mathsf{Dave}\}$ for the collateral evaluation workflow. Consider the workflow traces in Figure 4. The traces i_2, i_3, and i_4 model successfully terminated workflow instances of the collateral evaluation workflow, where the inner loop was executed twice, *i.e.* $i_2, i_3, i_4 \in \mathsf{T}(W[\pi^{-1}])$. We discuss the differences between these traces in later examples. The trace i_1, however, neither models a successfully terminated workflow instance nor is it a workflow instance trace of $W[\pi^{-1}]$ because t_4 can only be executed after t_3 has been executed. ◇

Example 2 illustrates that successfully terminated workflow instances may contain multiple instances of a task. For example, t_2 and t_4 are part of the collateral evaluation workflow and i_2 contains two execution events involving t_2 but none involving t_4.

B. Authorization processes

We now introduce a formalism to model authorization policies for workflows at the granularity of task instances. We support three classes of constraints.

- **Static authorizations:** The task execution is restricted to users with the necessary qualifications and responsibilities and does not change depending on the history of executed tasks.
- **Dynamic Separation of Duties:** Authorizations to execute tasks are restricted based on who has executed previous tasks to ensure that tasks with conflicting interests are not executed by the same user. For example, consider two tasks t_1 and t_2 with conflicting interests. A dynamic SoD constraint is used to prevent a user from executing an instance of t_2 after having executed an instance of t_1 and vice versa.
- **Dynamic Binding of Duties:** Authorizations to execute tasks are restricted based on who has executed previous tasks to limit the exposure of sensitive data and to reuse knowledge that users have gained from previous task executions. For example, consider two tasks t_1 and t_2, both revealing the same sensitive information. A dynamic BoD constraint forces a user to execute all instance of t_2 (and further instances of t_1) after having executed an instance of t_1 and vice versa.

It is standard to distinguish between static and dynamic authorization constraints [6]. In this paper, static SoD and BoD constraints are subsumed by static authorizations and are not discussed explicitly. We will therefore use SoD as synonym for dynamic SoD and BoD for dynamic BoD, respectively.

For each class of constraints, we now describe its visualization in BPMN and define its semantics in terms of CSP. For each constraint c, we define a process A_c and say that a workflow trace i *satisfies* c if $i \in \mathsf{T}(A_c)$.

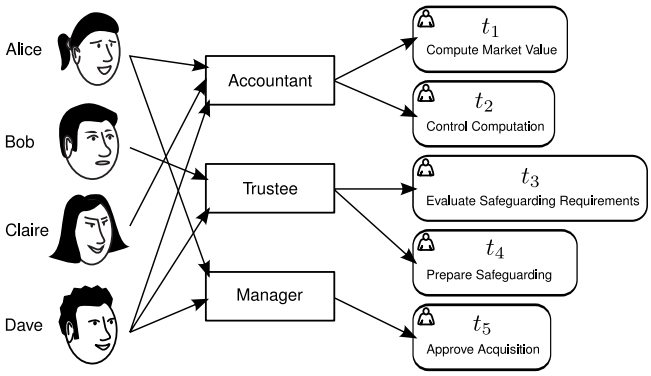

Figure 5. Role-based static authorizations

1) Static Authorization Constraints: A policy for static authorizations is basically a standard access control policy, describing the assignment of permissions to users. In the context of this paper, a permission is the right to execute a task. There exists a wealth of models and formalisms for describing static authorizations. For example, Figure 5 shows a role-based model [4] describing the static authorizations of Alice, Bob, Claire, and Dave with respect to the tasks of the collateral evaluation workflow.

Static authorizations are not the main focus of this paper. Often, they are not defined within the workflow model but on the system on which the workflow is executed,

e.g. see [7]. In the interest of ecumenical neutrality and supporting numerous access control languages, we model static authorizations abstractly by a relation $UT \subseteq \mathcal{U} \times \mathcal{T}$, called a *user-task assignment*. Given a user-task assignment UT, a user u, and a task t, we say u is *statically authorized* to execute t with respect to UT if $(u,t) \in UT$.

Definition 3 (Static Authorization Process) *For a user-task assignment UT, a* static authorization process *for UT is the process*

$$\begin{aligned} A_{UT} \quad = \quad & (t.u) : \{t'.u' \mid (u',t') \in UT\} \to A_{UT} \\ & \Box\, o : \mathcal{O} \to A_{UT} \\ & \Box\, SKIP\ . \end{aligned}$$

The process A_{UT} engages in every execution event $t.u$ if the user u is authorized to execute the task t with respect to UT. Furthermore, A_{UT} engages in every point o and can terminate at any time. The static nature of UT is reflected by the fact that A_{UT} behaves again like A_{UT} after engaging in every event (except the final event ✓).

As previously mentioned, static authorizations are often defined outside the workflow model. We therefore do not discuss the graphical description of static authorizations as part of workflow models in detail. BPMN has the concept of *lanes*, which subdivide workflows and group tasks [11]. Lanes are often associated with a role or a business unit and may therefore be interpreted as the role-permission assignment of a role-based access control model. For example, we could add a lane labeled Accountant, containing the tasks t_1 and t_2, a lane Trustee, containing t_3 and t_4, and a lane Manager for t_5 to the BPMN model of the collateral evaluation workflow (cf. Figure 5). However, the specification of users and their assignment to lanes is outside the scope of BPMN. Furthermore, if tasks can be executed by multiple roles, a workflow may have exponentially many lanes in the number of roles.

2) SoD Constraints: To separate duties between two tasks, we must keep track of the users who execute them in order to block users from executing both tasks. Thus, we associate a user with the tasks he executes and determine his authorizations to execute further tasks based on his association to previously executed tasks. We now introduce the concept of *releasing*, which removes associations between users and their previously executed tasks and thereby scopes SoD constraints to instances of tasks.

For two non-empty, disjoint sets of tasks T_1 and T_2, *i.e.* $|T_1| \geq 1$, $|T_2| \geq 1$, and $T_1 \cap T_2 = \varnothing$, and a set of points O, an *SoD constraint* is a triple (T_1, T_2, O).

Definition 4 (SoD Process) *For an SoD constraint $s = (T_1, T_2, O)$, the* SoD process *for s is the process $A_s(\mathcal{U},\mathcal{U})$ where*

$$\begin{aligned} A_s(U_{T_1}, U_{T_2}) = \\ & t : T_1.u : U_{T_1} \to A_s(U_{T_1}, U_{T_2} \setminus \{u\}) \\ & \Box\, t : T_2.u : U_{T_2} \to A_s(U_{T_1} \setminus \{u\}, U_{T_2}) \\ & \Box\, o : O \to A_s(\mathcal{U},\mathcal{U}) \\ & \Box\, t : \mathcal{T} \setminus (T_1 \cup T_2).u : \mathcal{U} \to A_s(U_{T_1}, U_{T_2}) \\ & \Box\, o : \mathcal{O} \setminus O \to A_s(U_{T_1}, U_{T_2}) \\ & \Box\, SKIP\ . \end{aligned}$$

An SoD process A_s offers the external choice between six kinds of events. (1) For a user u, A_s engages in the execution event $t_1.u$, for $t_1 \in T_1$, if u is not associated with a task $t_2 \in T_2$, *i.e.* u has not executed a task in T_2 that has conflicting duties with t_1. Afterward, A_s associates u with T_1 to block u from executing tasks in T_2. (2) Symmetrically, A_s blocks u from executing a task in T_1 after executing a task in T_2. (3) By engaging in a point $o \in O$, A_s releases all users from their associations with T_1 and T_2. We therefore call a point used in an SoD (or BoD) constraint a *release point*. (4) A_s engages also in every execution event involving tasks other than T_1 and T_2 and (5) points other than O without changing its behavior. (6) Finally, A_s may behave like $SKIP$ and terminate at any time.

We may use the following shorthand notation to describe SoD constraints and to avoid cluttering graphical workflow models. Consider the SoD constraint (T_1, T_2, O). If T_1, T_2, or O are singleton sets, we simply use the respective element and omit the set notation. For example, if $T_1 = \{t_1\}$, $T_2 = \{t_2\}$, and $O = \{o\}$, we write (t_1, t_2, o).

To visualize SoD constraints in BPMN, we introduce a new class of internal (BPMN) events, called *release events*. This facilitates the description of releasing as part of a workflow's control-flow. The release event icon is a user who leaves a door, as shown in Figure 3 with o_1, o_2, and o_3. We use the dot-dashed BPMN notation for grouping tasks to specify sets of tasks. For example, Figure 3 contains a group denoting the set of tasks $\{t_1, t_2, t_3, t_4\}$. An SoD constraint is graphically described by linking two disjoint, non-empty sets of tasks and a set of release events with a dotted line, joined by a node labeled with the symbol "$\neq$". This notation is an adaptation of BPMN's textual annotation of tasks. If one of the sets of tasks is a singleton set, we may omit the BPMN grouping and directly link the respective task and the $\neq$-node. For example, Figure 3 contains the SoD constraint $s_2 = (\{t_1, t_2, t_3, t_4\}, t_5, o_1)$.

The effect of an SoD constraint is only fully defined with respect to a workflow process. The workflow process defines the order in which tasks are executed and release points are reached. We illustrate the effect of different placements of a release point with an example.

Example 3 (Release Point Placement) Figure 6 shows a workflow with two tasks and three SoD constraints, $s_i = (t_1, t_2, o_i)$ for $i \in \{1, 2, 3\}$. Successfully terminated instances of this workflow correspond to workflow instance traces of the form

$$\langle\, o_1, o_2, o_3, t_1.u_{1,1}, \ldots, o_3, t_1.u_{1,n_1}, t_2.u_{1,n_1+1},$$
$$o_2, o_3, t_1.u_{2,1}, \ldots, o_3, t_1.u_{2,n_2}, t_2.u_{1,n_2+1},$$
$$\ldots$$
$$o_2, o_3, t_1.u_{m,1}, \ldots, o_3, t_1.u_{m,n_m}, t_2.u_{m,n_m+1}, \checkmark \rangle$$

for $n_m, m \geq 1$. The only difference between s_1, s_2, and s_3 is the position of the respective release point within the workflow. SoD constraint s_1 is satisfied if $\{u_{1,1}, u_{1,2}, \ldots, u_{1,n_1}, u_{2,1}, \ \ldots, u_{m,n_m}\} \cap \{u_{1,n_1+1}, u_{2,n_2+1}, \ldots, u_{m,n_m+1}\} = \varnothing$. In other words, s_1 is satisfied if no user who executes t_1 executes t_2 and vice versa. Because o_1 is reached only once and before any constrained task is executed, effectively no releasing takes place. Reaching a release point that is placed at the very start or end of a workflow has no effect and, hence, the constraint separates duties over all instances of the respective tasks. This illustrates that our policies are more expressive than existing SoD formalisms that do not distinguish between different instances of the same task.

Let $k \in \{1, 2, \ldots, m\}$. The SoD constraint s_2 is satisfied if $u_{k,n_1+1} \notin \{u_{k,1}, u_{k,2}, \ldots, u_{k,n_1}\}$. That is, for every execution of the workflow's outer loop, s_2 separates the duties between users who execute t_1 and those who execute t_2. Finally, s_3 is satisfied if $u_{k,n_1} \neq u_{k,n_1+1}$. Thus, in every execution of the workflow's outer loop, only the user who executes the last instance of t_1 must be different from the user who executes t_2. It follows that a workflow instance that satisfies s_1 also satisfies s_2 and s_3. Moreover, an instance satisfying s_2 also satisfies s_3. $\diamond$

3) BoD Constraints: Assume we want to bind duties between a set of tasks T. At first, every user is authorized to execute all tasks. Once a user has executed an instance of a task in T, no other user is authorized to execute instances tasks in T anymore. Again we use release points to scope BoD constraints to subsets of task instances.

For a non-empty set of tasks T, $|T| \geq 1$, and a set of points O, a *BoD constraint* is a tuple (T, O).

Definition 5 (BoD Process) *For a BoD constraint $b = (T, O)$, the* BoD process *for b is the process $A_b(\mathcal{U})$ where*

$$\begin{aligned} A_b(U) \quad = \quad & t : T.u : U \to A_b(\{u\}) \\ & \Box\, o : O \to A_b(\mathcal{U}) \\ & \Box\, t : \mathcal{T} \setminus T.u : \mathcal{U} \to A_b(U) \\ & \Box\, o : \mathcal{O} \setminus O \to A_b(U) \\ & \Box\, SKIP \ . \end{aligned}$$

The BoD process $A_b(U)$ offers the external choice between five kinds of events. (1) It engages in every execution event $t.u$ for $t \in T$ and $u \in U$. Initially $U = \mathcal{U}$. Once a user u executes a task in T, U is updated to $\{u\}$. Only after engaging in one of the release points in O are users other than u authorized to execute tasks in T again. Thus, for $t \in T$, executing $t.u$ "binds" u to T until (2) an $o \in O$ is reached and u is released. In particular, for $|T| = 1$ the respective BoD constraint binds the duties of all instances of a single task. Similar to SoD processes, $A_b(U)$ engages (3) in every execution event involving tasks other than those in T, (4) points other than the ones in O, and (5) may behave like $SKIP$ and terminate at any time.

As with SoD constraints, we visualize BoD constraints in BPMN by linking a non-empty set of tasks and a set of release event with a dotted line, joined by a node labeled with the symbol "=". We may also use the shorthand notation introduced for SoD constraints. For example, Figure 3 contains the BoD constraint $b = (\{t_3, t_4\}, o_3)$. Similar to SoD processes, the placement of release points with respect to a workflow process effects the semantics of a BoD constraint.

4) Constraint Composition: For a user-task assignment UT, a set of SoD constraints S, and a set of BoD constraints B, we call the triple (UT, S, B) an *authorization policy*. We define the semantics of authorization policies by composing the respective static authorization process and the sets of SoD and BoD processes.

Definition 6 (Authorization Process) *For an authorization policy $\phi = (UT, S, B)$, the* authorization process *for ϕ is the process*

$$A_\phi = A_{UT} \parallel \Big(\, \Vert_{s \in S} A_s \Big) \parallel \Big(\, \Vert_{b \in B} A_b \Big) \ .$$

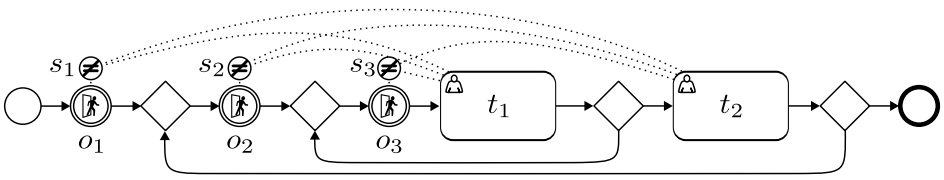

Figure 6. Location matters: The placement of a release point effects the semantics of the respective SoD constraint

Given a workflow trace i and an authorization policy $\phi = (UT, S, B)$. We say i *satisfies* ϕ if $i \in \mathsf{T}(A_\phi)$. By the trace semantics of CSP, i satisfies ϕ if and only if i satisfies UT, all SoD constraints in S, and all BoD constraints in B. We say ϕ is an authorization policy *for* a workflow process W if all tasks and points in ϕ appear in W. Furthermore, we call a workflow whose execution is constrained by an authorization policy an *authorization-constrained workflow*. In the following example, we provide an authorization policy for the collateral evaluation workflow.

Example 4 (Authorization Policy) Consider the authorization policy $\phi = (UT, S, B)$, where UT is illustrated in Figure 5 and $S = \{s_1, s_2\}$ and $B = \{b\}$ are illustrated in Figure 3. Furthermore, consider the traces i_2, i_3, and i_4 of Example 2, which model successfully terminated instances of the collateral evaluation workflow. Trace i_2 does not satisfy ϕ because Alice executed t_1 and t_2 before reaching o_1, thereby violating s_1. Trace i_3 does not satisfy ϕ for several reasons: s_2 is violated because Claire executed t_2 and t_5, b is violated because t_3 and t_4 are not executed by the same user, and UT is violated because Claire is statically not authorized to execute t_5. However, i_4 satisfies ϕ. ◇

IV. Enforcing authorization policies

We now explain how to enforce authorization policies at the task-execution level without changing the workflow at the control-flow level.

A. Obstruction

We link the control-flow and task-execution level by the notion of obstruction.

Definition 7 (Obstruction) *Let W be a workflow process, ϕ an authorization policy, and $i \in \mathsf{T}(W[\pi^{-1}])$ a workflow trace of W. We say that i is* obstructed *if there exists a task t such that $i[\pi]\hat{}t \in \mathsf{T}(W)$ but there does not exist a user u such that $i\hat{}\langle t.u\rangle$ satisfies ϕ.*

An obstruction describes a state of a workflow instance where the enforcement of the authorization policy conflicts with the business objectives. At the control-flow level, the business objectives can be achieved by executing a task t but at the task-execution level there is no user who is authorized to execute t without violating the authorization policy ϕ.

Example 5 (Obstructed Workflow Trace) Consider the workflow process W and the authorization policy ϕ introduced in Examples 1 and 4, respectively. Furthermore, consider the workflow trace $i = \langle t_1.\mathsf{Alice}, t_2.\mathsf{Claire}, t_3.\mathsf{Dave}, t_4.\mathsf{Dave}\rangle$, modeling an instance of the collateral evaluation workflow, *i.e.* $i \in \mathsf{T}(W[\pi^{-1}])$. After executing the workflow instance corresponding to i, task t_5 can be executed according to the collateral evaluation workflow, *i.e.* $i[\pi]\hat{}t_5 \in \mathsf{T}(W)$. However, the only users who are statically authorized to execute t_5 with respect to UT are Alice and Dave, but neither $i\hat{}\langle t_5.\mathsf{Alice}\rangle$ nor $i\hat{}\langle t_5.\mathsf{Dave}\rangle$ satisfy ϕ. Hence, i is obstructed. In this example, the workflow instance cannot even successfully terminate without violating ϕ. ◇

B. Enforcement Processes

We describe the enforcement of an authorization policy on a workflow process W in terms of a process E that executes in parallel to $W[\pi^{-1}]$, formally $W[\pi^{-1}] \parallel E$.

Definition 8 (Enforcement Process) *For a workflow process W and an authorization policy ϕ for W, an* enforcement process *for ϕ on W, written $E_{\phi,W}$, is a process that satisfies the conditions*

1) $A_\phi \sqsubseteq_{\mathsf{T}} E_{\phi,W}$ *and*
2) $(W[\pi^{-1}] \parallel E_{\phi,W})[\pi] =_{\mathsf{F}} W$.

Unlike the authorization process, the enforcement process not only implements the authorization policy ϕ but also takes W into account. Condition 1 states that $E_{\phi,W}$ is at least as restrictive as A_ϕ. The failure equivalence used in Condition 2 states that at the control-flow level the processes W and W constrained by $E_{\phi,W}$ are indistinguishable.

Suppose $E_{\phi,W}$ is an enforcement process for ϕ on W. By CSP's trace semantics, if $i \in \mathsf{T}(W[\pi^{-1}] \parallel E_{\phi,W})$ then $i \in \mathsf{T}(W[\pi^{-1}])$ and $i \in \mathsf{T}(E_{\phi,W})$. It follows that i satisfies ϕ by Condition 1. Due to the failure equivalence of $(W[\pi^{-1}] \parallel E_{\phi,W})[\pi]$ and W, *i.e.* Condition 2, i is not obstructed. Hence, $E_{\phi,W}$ is an obstruction-free enforcement of ϕ on W.

We now give an example of an enforcement process for the authorization-constrained collateral evaluation workflow.

Example 6 (Enforcement Process) Consider W and ϕ from the previous examples and the following processes.

$$E = (E_1 \,|||\, E_2)\ ;\ (t_5.\mathsf{Dave} \to ((o_2 \to E) \sqcap SKIP))$$
$$E_1 = t_1.\mathsf{Alice} \to t_2.\mathsf{Claire} \to ((o_1 \to E_1) \sqcap SKIP)$$
$$E_2 = o_3 \to t_3.\mathsf{Bob} \to ((t_4.\mathsf{Bob} \to SKIP) \sqcap SKIP)$$

All traces of E satisfy ϕ and therefore Condition 1 of Definition 8 holds. By the laws of CSP and the structure of E, $(W[\pi^{-1}] \parallel E)[\pi] = W[\pi^{-1}][\pi] \parallel E[\pi] = W \parallel W = W$ and therefore Condition 2 holds too. Therefore, E is an enforcement process for ϕ on W. ◇

For illustration purposes, this example is rather simple in that all instances of the same task must be executed by the same user, for example Alice is the only user who executes instances of t_1. Enforcement processes can, of course, be much more complex and also authorize multiple users to execute instances of the same task.

According to Definition 8, an authorization policy is only enforceable if a workflow remains unchanged at the control-flow level. This is a design decision and other options are possible. For example, one could choose to give authorizations precedence over an obstruction-free enforcement.

However, even if the policy *must* be enforced and obstructed workflow instances are tolerated, our approach is helpful because it reveals tasks that may not be executed. The workflow can consequently be simplified without reducing the set of possible workflow instances.

C. The Enforcement Process Existence Problem

We now formulate the existence of an enforcement process as a decision problem and present complexity bounds.

Definition 9 (Enforcement Process Existence Problem **EPE**) *Given: A workflow process W and an authorization policy ϕ. Output:* YES *if there exists an enforcement process for ϕ on W and* NO *otherwise.*

We first show that **EPE** is **NP**-hard by reducing the **NP**-hard graph-coloring problem k-**COLORING**, summarized in Appendix B, to **EPE**.

Lemma 1 **EPE** *is* **NP***-hard.*

Proof: Given a k-**COLORING** instance consisting of a graph $G = (V, E)$ and an integer k, we describe a polynomial reduction to **EPE**. We construct a workflow process W and an authorization policy $\phi = (UT, S, B)$ and show that there exists a k-coloring for G if there exists an enforcement process for ϕ on W. Let $\mathcal{T} = V$, for $V = \{v_1, v_2, \ldots, v_n\}$, and $\mathcal{U} = \{1, 2, \ldots, k\}$. Now consider $W = v_1 \to v_2 \to \ldots \to v_n \to SKIP$, $UT = \mathcal{U} \times \mathcal{T}$, $B = \varnothing$, and for every edge $(v_l, v_m) \in E$ we construct an SoD constraint $(v_l, v_m, \varnothing)$. Figure 7 illustrates this construction for a graph with $n = 5$ and $k = 4$.

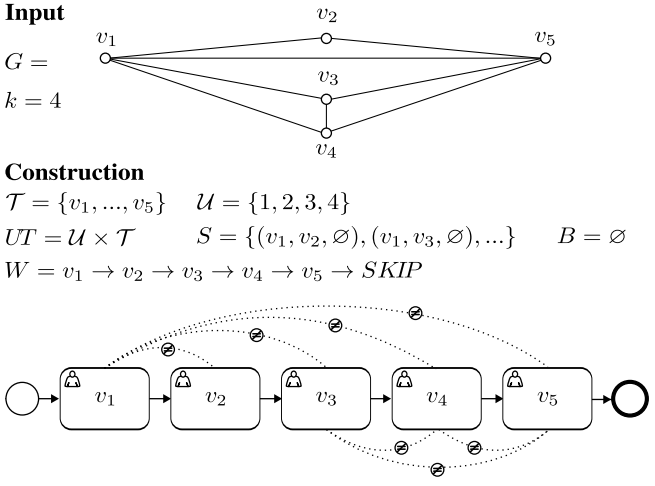

Figure 7. Illustration of polynomial reduction from k-**COLORING** to **EPE**

By the construction of W, $h = \langle v_1, v_2, \ldots, v_n, \checkmark \rangle \in \mathsf{T}(W)$. If an algorithm for **EPE** returns YES, then an enforcement process $E_{\phi,W}$ exists by Definition 9 and $h \in \mathsf{T}((W[\pi^{-1}] \parallel E_{\phi,W})[\pi])$ by Definition 8. It follows that there exists a workflow trace $i = \langle v_1.u_1, v_2.u_2, \ldots, v_n.u_n, \checkmark \rangle \in \mathsf{T}(W[\pi^{-1}] \parallel E_{\phi,W})$. By our construction, $u_j \in \{1, \ldots, k\}$, for $j \in \{1, \ldots, n\}$. Therefore, every task (*i.e.* node) is executed exactly once and thus associated with one of k users (*i.e.* colors). By Definition 8, $i \in \mathsf{T}(A_\phi)$ and i satisfies every constraint in ϕ. Therefore, for every SoD constraint $(v_l, v_m, \varnothing)$ in S, the user u_l who executes v_l is different from the user u_m who executes v_m. Hence, i describes a k-coloring for G. Because this reduction is in polynomial time, it follows that **EPE** is **NP**-hard. ■

We do not know whether **EPE** is in **NP**. However, it is decidable when $\mathcal{U}$ and W are finite.

Theorem 1 **EPE** *is decidable if $\mathcal{U}$ and W are finite.*

We sketch a proof here and give full details in Appendix C.

If $\mathcal{U}$ and W are finite, it follows that A_ϕ is finite too by Definitions 3–5 and the operational semantics of CSP. If there is an enforcement process $E_{\phi,W}$, it must satisfy the two conditions of Definition 8. Because A_ϕ is finite, for every process C, such that $A_\phi \sqsubseteq_{\mathsf{T}} C$, there is a finite labelled transition system that corresponds to C. We can therefore construct all processes C that are candidates to be $E_{\phi,W}$ with respect to Condition 1. Let C be one of them. Because W and $\mathcal{U}$ are finite, so is $W[\pi^{-1}]$. Furthermore, $(W[\pi^{-1}] \parallel C)[\pi]$ is finite because π and C are finite. Because failure-refinement is decidable for finite processes [13], we can check if C satisfies Condition 2, *i.e.* if $(W[\pi^{-1}] \parallel C)[\pi] =_{\mathsf{F}} W$. If C satisfies Condition 2, then C is an enforcement process for ϕ on W. If none of the finitely many candidate processes C satisfies Condition 2, then there exists no enforcement process for ϕ on W. □

The runtime complexity of solving **EPE** as sketched above is as follows. For an SoD constraint s, consider the SoD process A_s. The number of states of a transition system that corresponds to A_s is in $O(2^{|\mathcal{U}|})$ because A_s is parametrized by two subsets of $\mathcal{U}$ and there is a state for every possible subset. The number of states of a transition system corresponding to A_b, for a BoD constraint b, is linear in the size of $\mathcal{U}$. The number of states of a transition system corresponding to A_{UT}, for an user-task assignment UT, is constant. Let $\phi = (UT, S, B)$. By Definition 6 and the operational semantics for the parallel, synchronized composition of two processes (see Definition 12 in Appendix A), it follows that the number of states of a transition system corresponding to A_ϕ is in $O(|\mathcal{U}|^{|B|} 2^{|S||\mathcal{U}|})$. The set of input symbols of a transition system corresponding to A_ϕ is $(\mathcal{X} \cup \mathcal{O})^\checkmark$. Therefore, the number of transitions is in $O((|\mathcal{O}| + |\mathcal{T}||\mathcal{U}|)|\mathcal{U}|^{2|B|} 2^{2|S||\mathcal{U}|})$.

The above decision procedure checks for each transition system that has a subset of A_ϕ's transitions whether it satisfies Condition 2 of Definition 8. This requires deciding failure equivalence which is **PSPACE**-complete [13].

Thus, this approach has a runtime complexity that is double exponential in the number of users and constraints. Hence, it is not applicable to workflows with large sets of users. We therefore propose approximation algorithms for **EPE** in the following section.

V. Approximations

We first present an approximation algorithm for **EPE**, called EPEA, with an exponential runtime complexity. Afterwards, we show how to approximate **EPE** in polynomial time, using bounds from graph-coloring.

EPEA, illustrated in Algorithm 1, takes an instance of **EPE** as input and returns either a relation or No. EPEA is an approximation in that it may return No even though an enforcement process for the given input exists. However, if EPEA returns a relation, this relation can be transformed to an enforcement process for the given **EPE** instance.

Algorithm 1: EPEA(T, ϕ)

Input: T and $\phi = (UT, S, B)$
Output: returns a relation $R \subseteq \pi^{-1}$ or No

if $T = \varnothing$ **then**
  return $\varnothing$
else
  $col_L \leftarrow$ LCol(CGraph(T, ϕ))
  if CGraph *or* LCol *return* No **then**
    return No
  else
    return $\{(t, t.u) \mid (T \mapsto u) \in col_L, t \in T\}$

In more detail, EPEA composes CGraph and LCol. CGraph, given in Algorithm 2, transforms the tasks of a workflow process W, *i.e.* $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$, and an authorization policy $\phi = (UT, S, B)$ to an instance of the **ListColoring** problem. **ListColoring** is a generalization of the well-known k-**Coloring** problem. A review of these problems and graph-coloring terminology is given in Appendix B. CGraph returns either V, E, and L, where (V, E) is a graph and $L : V \to 2^{\mathcal{U}}$ is a *list-coloring function* for (V, E), or No. The vertices in V are sets of tasks of W. Every task of W is contained in one vertex. The BoD constraints B define which sets of tasks form vertices, UT defines L, and the edges correspond to the SoD constraints in S.

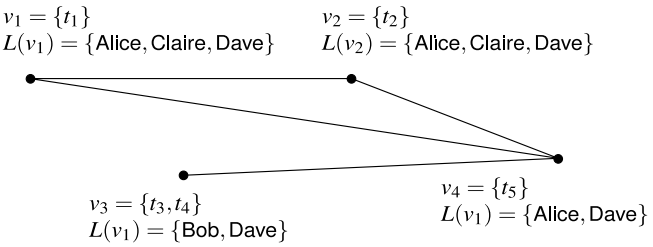

Figure 8. Constraint graph of the collateral evaluation workflow

CGraph returns No if W contains two tasks t_1 and t_2 whose execution is constrained by an SoD constraint in S and if there is a subset of BoD constraints in B that bind the duties between t_1 and t_2.

Algorithm 2: CGraph(T, ϕ)

Input: T and $\phi = (UT, S, B)$
Output: returns a graph (V, E) and a list coloring function $L : V \to 2^{\mathcal{U}}$ or No

$V, E, L \leftarrow \varnothing$
foreach $t \in T$ **do**
  $V \leftarrow V \cup \{\{t\}\}$
  $L \leftarrow L \cup \{(\{t\} \mapsto \{u \mid (u, t) \in UT\})\}$
foreach $(T_1, O) \in B$ **do**
  pick a $t_1 \in T_1$
  let $v_1 \in V$ s.t. $t_1 \in v_1$
  foreach $t_2 \in T_1 \setminus \{t\}$ **do**
    let $v_2 \in V$ s.t. $t_2 \in v_2$
    $V \leftarrow (V \setminus \{v_1, v_2\}) \cup \{v_1 \cup v_2\}$
    $L \leftarrow (L \setminus \{(v_1 \mapsto L(v_1)), (v_2 \mapsto L(v_2))\}) \cup \{(v_1 \cup v_2 \mapsto L(v_1) \cap L(v_2))\}$
foreach $(T_1, T_2, O) \in S$ **do**
  foreach $t_1 \in T_1$ **do**
    let $v_1 \in V$ s.t. $t_1 \in v_1$
    foreach $t_2 \in T_2$ **do**
      let $v_2 \in V$ s.t. $t_2 \in v_2$
      if $v_1 \neq v_2$ **then**
        $E \leftarrow E \cup \{(v_1, v_2)\}$
      else
        return No
return V, E, L

Example 7 (Graph Returned by CGraph) Figure 8 illustrates the graph and the list coloring function L returned by CGraph for the tasks of the collateral evaluation workflow and our example authorization policy ϕ. ◇

LCol is a standard algorithm for solving **ListColoring**. In Appendix B, we described LCol in detail and prove its correctness and completeness. EPEA first transforms its input to a **ListColoring** instance using CGraph. Afterwards, it solves the instance using LCol. Finally, it transforms the coloring returned by LCol to a relation between tasks and execution events and returns this relation. If CGraph does not succeed in building a graph or LCol does not find a coloring, then EPEA returns No.

Lemma 2 *Let W be a workflow process, $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$, and ϕ an authorization policy. If* $\text{EPEA}(T, \phi)$ *returns a relation R, then $W[R]$ is an enforcement process for ϕ on W.*

Proof: Assume a workflow process W, let $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$, and $\phi = (UT, S, B)$ be an authorization policy. Assume $\text{EPEA}(T, \phi)$ returns a relation R. We refer to a line i of CGRAPH as CGi and to line i of EPEA as EAi.

If $T = \varnothing$, then W does not engage in any task and $R = \varnothing$ by $EA2$. Because ϕ is an authorization policy for W and W contains no tasks, $UT = \varnothing$, $S = \varnothing$, and $B = \varnothing$. It follows that $A_\phi = A_{UT}$. Therefore, A_ϕ engages in every point and $\checkmark$, by Definition 3. It follows that $A_{UT} \sqsubseteq_{\mathsf{T}} W$, *i.e.* Condition 1 of Definition 8 holds. By the trace semantics of CSP and because W does not engage in tasks, $(W[\pi^{-1}] \parallel W[\varnothing])[\pi] =_{\mathsf{F}} (W \parallel W)[\pi] =_{\mathsf{F}} W[\pi] =_{\mathsf{F}} W$, *i.e.* Condition 2 of Definition 8 holds.

Assume $T \neq \varnothing$. Because EPEA returns a relation and $T \neq \varnothing$, $\text{CGRAPH}(T, \phi)$ returns a graph (V, E) and a function L by EA1, EA4, and EA5. Furthermore, $\text{LCOL}(V, E, L)$ returns a coloring col_L by EA4 and EA5. Because $T \neq \varnothing$ and by CG2, CG3, and CG10, $V \geq 1$. It follows from Lemma 4 in Appendix 4 that col_L is an L-coloring for (V, E). Let $t \in T$. By CG2, CG3, and CG10, there is exactly one vertex $v \in V$ such that $t \in v$. Therefore, there is exactly one tuple $(t, t.u) \in R$ by EA8, for a user u.

Let $i \in \mathsf{T}(W[R])$. In the following, we show for every constraint $c \in (\{UT\} \cup S \cup B)$ that $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_c)$. By Definitions 3–5, also $i \,\hat{}\, \langle o \rangle \in \mathsf{T}(A_c)$, for $o \in \mathcal{O}$, and $i \,\hat{}\, \langle \checkmark \rangle \in \mathsf{T}(A_c)$. It follows that $A_\phi \sqsubseteq_{\mathsf{T}} W[R]$, *i.e.* Condition 1 of Definition 8 holds.

Case UT: Let $v \in V$ such that $t \in v$. By EA8, $u = col_L(v)$. By the definition of L-coloring, $col_L(v) \in L(v)$. By CG4 and CG11, $L(v) \subseteq \{u' \mid (u', t) \in UT\}$. Hence, $(u, t) \in UT$ and $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_{UT})$ by Definition 3.

Case $s \in S$: Let $s = (T_1, T_2, O)$. If $t \notin (T_1 \cup T_2)$ then $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_s)$ by Definition 4. Consider the case $t \in (T_1 \cup T_2)$. Because $(T_1 \cap T_2) = \varnothing$ by the definition of SoD constraints, assume without loss of generality that $t \in T_1$. Let $t_2 \in T_2$ and $(t_2, t_2.u_2) \in R$, for a user u_2. Furthermore, let $v_1, v_2 \in V$ such that $t \in v_1$ and $t_2 \in v_2$. By CG12–CG18, $(v_1, v_2) \in E$. By the definition of L-coloring, $col_L(v_1) \neq col_L(v_2)$ and therefore $u \neq u_2$ by EA8. Because there is only one execution event in R for every task, $t_2.u \notin i$ and therefore $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_s)$ by Definition 4.

Case $b \in B$: Let $b = (T_1, O)$. If $t \notin T_1$ then $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_b)$ by Definition 5. Consider the case $t \in T_1$. Let $t_2 \in T_1$ and $(t_2, t_2.u_2) \in R$ for a user u_2. Let $v \in V$ such that $t \in v$. By CG5–CG11 it holds that $t_2 \in v$. By EA8 it follows that $u = u_2$. Therefore, no matter whether $t_2.u_2 \in i$ or $t_2.u_2 \notin i$, $i \,\hat{}\, \langle t.u \rangle \in \mathsf{T}(A_b)$ by Definition 5.

It remains to be shown that $W[R]$ satisfies Condition 2 of Definition 8. By CSP's semantics and because $R \subseteq \pi^{-1}$, $(W[\pi^{-1}] \parallel W[R])[\pi] =_{\mathsf{F}} W[R][\pi] =_{\mathsf{F}} W$. ∎

A. Polynomial Approximation

By applying graph-coloring bounds to the graph returned by CGRAPH, we can approximate **EPE** in polynomial time.

Corollary 1 *For a workflow process W and an authorization policy ϕ, let $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$ and $(V, E, L) = \text{CGRAPH}(T, \phi)$. If*

$$\max_{v \in V} |\{v' \mid (v, v') \in E\}| < \min_{v \in V} |L(v)|$$

then there exists an enforcement process for ϕ on W.

Proof: Let W be a workflow process, ϕ an authorization policy, $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$, and $(V, E, L) = \text{CGRAPH}(T, \phi)$. Then $\max_{v \in V} |\{v' \mid (v, v') \in E\}|$ is the maximal degree $\Delta(V, E)$ of (V, E). Furthermore, let $k = \min_{v \in V} |L(v)|$, *i.e.* L is a k-color-list function for (V, E). Assume that $\Delta(V, E) < k$. By Lemma 3 in Appendix B it follows that $\chi_l(V, E) \leq k$. Therefore, there exists an L-coloring for (V, E). Hence, $\text{EPEA}(T, \phi)$ returns a relation R and, by Lemma 2, $W[R]$ is an enforcement process for ϕ on W. ∎

Informally, Corollary 1 tells us the following. If the maximal number of SoD constraints under which a task is constrained is less than the minimal number of users who are authorized both statically and with respect to the BoD constraints to execute a task, then there exists an enforcement process. Said more simply, there exists an enforcement process if the set of users is large and their static authorizations are well-distributed.

Assume a workflow process W and an authorization policy ϕ. The algorithm CGRAPH computes (V, E, L) in polynomial time or returns NO. We can then check if the condition of Corollary 1 holds for V, E, and L. If it holds, we only know that an enforcement process for ϕ on W exists but $E_{\phi,W}$ is not constructed yet. However, by Lemma 3 and because the condition of Corollary 1 holds, a greedy algorithm with polynomial runtime complexity finds an L-coloring for (V, E). We can therefore replace the call to LCOL in EPEA by a call to the greedy algorithm. It follows that we can approximate **EPE** in polynomial time.

VI. RELATED WORK

Schneider formalized the concept of a *security automaton*, which is an enforcement monitor that is composed with an insecure system and checks whether commands are authorized prior to their execution [16]. Security automata, however, are limited in that preventing unauthorized commands either causes the target system to terminate or requires exception handling to be part of the security automaton as well as the target system. To overcome this limitation, several extensions to security automata, such as *edit automata* [10], have been proposed. We follow another

direction by incorporating knowledge about the system's control-flow, given by a workflow, into the enforcement monitor. Our approach uses this additional information to enforce authorization policies while preserving all of the target system's options as defined by the workflow.

An authorization policy is sometimes, *e.g.* [2,18], called *satisfiable* with respect to a workflow if there exists an assignment of users to tasks that does not violate the policy. Our approximation algorithm determines such an assignment for the enforcement process. In general, however, enforcement processes are more expressive in terms of the authorization policies they support than a static assignment of users to tasks. This is also reflected in Solworth's notion of "unscheduled approvability" requiring that every workflow instance can be extended to a final state no matter which path is taken [17]. The business process community has gone one step further in defining what constitutes a well-formed workflow model. Van der Aalst [19] calls a workflow *sound* if it has no dead transitions and it does not deadlock before completing its final task. Obstruction-free authorization enforcement on sound workflows guarantees that the workflow will always successfully terminate and thus achieves its business objectives.

Early work on authorization constraints, such as the *Transaction Control Expressions* proposed by Sandhu [15], model workflows only as part of the constraints, for example by stating how often a task must be executed. Bertino, Ferrari and Atluri were the first to model workflows explicitly, defining workflows as sequences of tasks. In their model, constraints on task executions are given by clauses in a logic program [2]. Later Tan, Crampton and Gunter refine a workflow to be a partially ordered set of tasks and explicitly define workflow and task instances [18]. Authorization constraints are given for pairs of tasks in terms of relations over users that must be satisfied when executed.

The above and most other work on the enforcement of constraints ignore conditions, loops and parallelism in workflows. A notable exception is Solworth [17], who models a workflow as a directed graph. However, constraints in the presence of loops are restricted such that the first task ("consuming" a user) must always be executed by the same person. Given a sufficient number of users per task, these restriction ensure that a workflow can always be completed when there are no conflicts between SoD and BoD constraints. The graph transformation used in CGRAPH is inspired by Solworth's conflict graph [17].

We do not impose restrictions, either on workflows or on constraints. Furthermore, to our knowledge, there is no other work related to our concept of release. By introducing release points into workflows, we support the fine-grained control of constraints in the presence of loops, scoping authorization constraints to subsets of task instances. Constraints with release points extend previous work on security-annotated graphical workflow models [20]. In addition, we give a formal semantics such that no workflow created by parallel and conditional task execution introduces ambiguity.

VII. CONCLUSIONS

We have presented a new approach to aligning security and business objectives for information systems. Using CSP, we modeled a system at two levels of abstraction: the control-flow level modeling the system's business objectives, and the task-execution level modeling who executes which task. We bridged these levels by the notion of obstruction which generalizes deadlocks. Furthermore, we presented a novel approach to scope SoD and BoD constraints to subsets of task instances using release points. Our formalism thereby generalizes existing SoD and BoD specification languages that separate and bind duties between all instances of constrained tasks. We showed how to visualize our constraints by extending a well-established workflow modeling language. We thus maintain the intuition and visual appeal of graphical modeling languages, making it easier for business process designers and security administrators to cooperate in specifying and aligning security and business objectives.

Our work gives rise to many interesting questions. For example, given a workflow process W and an authorization policy ϕ, many processes may meet the conditions of an enforcement process for ϕ on W as required by Definition 8. This raises the question of what constitutes a "good" enforcement process. One idea is to search for an enforcement process $E_{\phi,W}$ such that $\mathsf{T}(A_\phi) \setminus \mathsf{T}(E_{\phi,W})$ is minimal. In other words, one that maximizes the number of authorized execution events and thereby minimizes the restrictions enforced at the task-execution level.

With our enforcement process definition, we require obstruction-freedom and allow the enforcement to be more restrictive than specified by the respective authorization process. The preservation of a workflow at the control-flow level is therefore given priority over allowing every authorized task execution. Other designs are possible and remain to be investigated.

We would like to sharpen our complexity analysis for **EPE**, ideally finding upper-bounds that match the lower-bounds we have given.

Finally, we are currently building prototype tool support for our approach by extending the BPMN meta-model with our SoD and BoD constraint language and adapting an existing BPMN modeling tool to support this extension and our approximation algorithm. Through realistic case studies, we hope to better understand the performance of our approximation algorithm in practice.

Acknowledgments: We thank Vincent Jugé, Felix Klaedtke, Dominik Rüegger, Mohammad Torabi Dashti, and the anonymous reviewers for their helpful comments. The research leading to these results has received funding from the European Community's Seventh Framework Programme

(FP7/2007-2013) under grant agreement N° 216917. This work is partially supported by the EU FP7-ICT-2009.1.4 Project N° 256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

REFERENCES

[1] "Sarbanes-Oxley Act of 2002". Public Law 107-204 (116 Statute 745), United States, 2002.

[2] E. Bertino, E. Ferrari, and V. Atluri, "The specification and enforcement of authorization constraints in workflow management systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 1, pp. 65–104, 1999.

[3] G. Chartrand and P. Zhang, *Chromatic Graph Theory*, Ser. Discrete Mathematics and Its Applications. Chapman & Hall, 2008.

[4] D.F. Ferraiolo, R.S. Sandhu, S.I. Gavrila, D.R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[5] Formal Systems (Europe) Ltd, "Failures-Divergence Refinement - FDR2 User Manual," www.fsel.com, 2005.

[6] V.D. Gligor, S.I. Gavrila, and D. Ferraiolo, "On the Formal Definition of Separation-of-Duty Policies and their Composition," in *19th IEEE Symposium on Security and Privacy (S&P '98)*, 1998, pp. 172–183.

[7] IBM Corporation, "WebSphere Process Server v6.2," www.ibm.com/software/integration/wps/, 2009.

[8] ——, "IBM Information Framework (IFW)," www.ibm.com/software/industry/banking, 2010.

[9] IT Governance Institute, "Control objectives for information and related technology (Cobit) 4.1," 2005.

[10] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 2–16, 2005.

[11] Object Management Group (OMG), "Business Process Model and Notation (BPMN), version 2.0," OMG Standard, http://www.omg.org/spec/BPMN/2.0/PDF, 2011.

[12] F. Puhlmann and M. Weske, "Using the π-calculus for formalizing workflow patterns," in *3rd International Conference on Business Process Management (BPM '05)*, 2005, pp. 153–168.

[13] A.W. Roscoe, "Model-checking CSP," *A classical mind*. Prentice Hall, pp. 353–378, 1994.

[14] ——, *The theory and practice of concurrency*. Prentice Hall, 2005.

[15] R.S. Sandhu, "Transaction control expressions for separation of duties," in *4th IEEE Aerospace Computer Security Applications Conference*, 1988, pp. 282–286.

[16] F.B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 30–50, 2000.

[17] J.A. Solworth, "Approvability," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS '06)*, 2006, pp. 231–242.

[18] K. Tan, J. Crampton, and C.A. Gunter, "The consistency of task-based authorization constraints in workflow systems," in *17th IEEE Computer Security Foundations Workshop (CSFW '04)*, 2004, pp. 155–169.

[19] W.M.P. van der Aalst, "The application of Petri nets to workflow management," *Journal of Circuits, Systems, and Computers (JCSC)*, vol. 8, no. 1, pp. 21–66, 1998.

[20] C. Wolter, A. Schaad, and C. Meinel, "Task-based entailment constraints for basic workflow patterns," in *13th ACM Symposium on Access Control Models and Technologies (SACMAT '08)*, 2008, pp. 51–60.

[21] P.Y.H. Wong and J. Gibbons, "A process-algebraic approach to workflow specification and refinement," in *6th International Symposium on Software Composition (SC '07)*, 2007, pp. 51–65.

APPENDIX

A. CSP

A labelled transition system (LTS) is a quadruple (Q, C, δ, q^0), where Q is a set of states, C is a set of input symbols, $\delta \subseteq Q \times C \times Q$ is a nondeterministic state transition relation, and $q^0 \in Q$ is a start state. For $n \geq 1$, $q_0, q_n \in Q$, and a sequence of events $\langle \sigma_1, \ldots, \sigma_n \rangle \in C^*$, we write $q_0 \xrightarrow{\langle \sigma_1, \ldots, \sigma_n \rangle} q_n$ if there exists a set of states $\{q_1, \ldots, q_{n-1}\}$ such that $(q_{k-1}, \sigma_k, q_k) \in \delta$ for all $k \in \{1, \ldots, n\}$.

CSP's operational semantics interprets a process as an LTS where the input symbols correspond to the events that the process engages in, *i.e.* $C \subseteq \Sigma^{\tau,\checkmark}$. Let $L = (Q, \Sigma^{\tau,\checkmark}, \delta, q^0)$. For $q_0, q_n \in Q$ and a trace $i \in \Sigma^{*\checkmark}$, we write $q_1 \stackrel{i}{\Longrightarrow} q_2$ if there exists a sequence of events $h \in (\Sigma^{\tau})^{*\checkmark}$ such that $q_1 \xrightarrow{h} q_2$ and i is equal to h without τ events.

Let $q_1 \in Q$ be a state and $C \subseteq \Sigma^{*\checkmark}$ a set of events. The set C is a *refusal set* of q_1, written $q_1 \; ref \; C$, if $C \subseteq \{\sigma \in \Sigma^{\checkmark} \mid \neg \exists q_2 \in Q, (q_1, \sigma, q_2) \in \delta\}$. We say an LTS $L = (Q, \Sigma^{\tau,\checkmark}, \delta, q^0)$ *corresponds*[1] to a process P if

$$\mathsf{F}(P) = \{(i, C) \mid \exists q_2 \in Q, q_1 \stackrel{i}{\Longrightarrow} q_2, q_2 \; ref \; C\} \cup \{(i, C) \mid \exists q_2 \in Q, q_1 \stackrel{i^\frown\langle\checkmark\rangle}{\Longrightarrow} q_2, C \subseteq \Sigma^{\checkmark}\}.$$

Note that there may be multiple LTSs that correspond to the same process. We write L_P for an LTS that corresponds to P.

[1]The CSP-versed reader may have realized that we omit a discussion of divergence. We implicitly assume that workflow processes are divergence free. Our renaming relations and authorization processes do not introduce divergence.

B. Graph Coloring

A *graph* G is a tuple (V, E) where V is a set of *vertices* and $E \subseteq V \times V$ is a set of (undirected) *edges*. The *maximal degree* of a graph G, denoted $\Delta(G)$, is $\max_{v \in V} |\{v' \in V \mid (v, v') \in E\}|$, *i.e.* the maximal number edges linking a vertex to other vertices.

Definition 10 (The k-**COLORING** problem)

Given: A graph $G = (V, E)$ and an integer $k \in \mathbb{N}$.

Output: YES *if there exists a function $col : V \to \{1, \ldots, k\}$ such that for every edge $(v_1, v_2) \in E$, $col(v_1) \neq col(v_2)$ and* NO *otherwise.*

Let a graph G and an integer k be given. We call a function col a *k-coloring for G* if col satisfies the condition described in the k-**COLORING** problem for G and k. The k-**COLORING** problem is **NP**-complete [3]. The following problem generalizes of k-**COLORING**.

Definition 11 (The **LISTCOLORING** problem)

Given: A graph $G = (V, E)$ and a function $L : V \to 2^C$, for a set C.

Output: YES *if there exists a function $col_L : V \to C$ such that for every vertex $v \in V$, $col_L(v) \in L(v)$ and for every edge $(v_1, v_2) \in E$, $col_L(v_1) \neq col_L(v_2)$ and* NO *otherwise.*

Unlike k-**COLORING**, **LISTCOLORING** does not offer the same set of colors for every vertex; for each vertex v, the colors must be chosen from a "list" of colors $L(v) \subseteq C$. Note, for historical reasons, what is called a *list* is actually a *set*. For consistency with the literature, we stick to the term list. Given a graph G and a color-list function L, we call a function col_L an *L-coloring* for G if col_L satisfies the condition described in Definition 11. We call L a *k-color-list function* if $|L(v)| \geq k$, for all $v \in V$. Given a graph G, the smallest integer k, such that G is L-colorable for all k-color-list functions L, is called G's *list-chromatic number* and is denoted $\chi_l(G)$. The maximal degree of a graph gives us an upper bound for the list-chromatic number.

Lemma 3 *For every graph G,*

$$\chi_l(G) \leq 1 + \Delta(G)$$

and a greedy algorithm for graph coloring with polynomial runtime finds an L-coloring for G for every $(1 + \Delta(G))$-color-list function L.

The definition of greedy algorithms for graph coloring is standard, *e.g.* see [3]. See also [3] for a proof of Lemma 3.

LISTCOLORING generalizes k-**COLORING** because a k-**COLORING** instance can be translated to a **LISTCOLORING** instance by setting $C = \{1, \ldots, k\}$, and $L(v) = C$, for every $v \in V$. Since a solution to the **LISTCOLORING** problem can be checked in polynomial time, **LISTCOLORING** is also **NP**-complete. Algorithm 3, called LCOL, solves **LISTCOLORING** in exponential time.

Algorithm 3: $\text{LCOL}(V, E, L)$

Input: $|V| \geq 1$, $E \subseteq V \times V$, and $L : V \to 2^C$, for a set C

Output: returns an L-coloring for (V, E) if it exists, NOotherwise

```
1  if V = {v} then
2      if |L(v)| ≥ 1 and (v, v) ∉ E then
3          let c ∈ L(v)
4          return {v ↦ c}
5      else
6          return NO
7  else
8      let v ∈ V
9      if (v, v) ∈ E then
10         return NO
11     foreach c ∈ L(v) do
12         V' ← V \ {v}
13         E' ← {(v1, v2) ∈ E | v1 ≠ v, v2 ≠ v}
14         L' ← ∅
15         foreach v' ∈ V' do
16             if (v, v') ∈ E then
17                 L' ← L' ∪ {(v' ↦ L(v') \ {c})}
18             else
19                 L' ← L' ∪ {(v' ↦ L(v'))}
20         return r ← LCOL(V', E', L')
21         if r ≠ NO then
22             r ∪ {(v ↦ c)}
23     return NO
```

Lemma 4 *Let a graph $G = (V, E)$, with $|V| \geq 1$, and a color-list function $L : V \to 2^C$, for a set C be given.*

- *Correctness: If* $\text{LCOL}(V, E, L)$ *returns a coloring col_L, then col_L is an L-coloring for G.*
- *Completeness: If there exists an L-coloring for G, then* $\text{LCOL}(V, E, L)$ *returns a coloring.*

Proof: Let $G = (V, E)$, with $|V| \geq 1$, and a color-list function $L : V \to 2^C$, for a set C, be given. We refer to a line i of Algorithm 3 as LCi. We first prove the correctness property and afterwards the completeness property of Lemma 4. We prove both cases by induction over V.

Correctness: Base case: Assume $V = \{v\}$ and let $col_L = \{v \mapsto c\} = \text{LCOL}(\{v\}, E, L)$. Therefore, $|L(v)| \geq 1$ and $(v, v) \notin E$ by $LC1$ and $LC2$. It follows that $E = \varnothing$ and $col_L(v) \in L(v)$ because of $LC3$. Hence, col_L is an L-coloring of G. Step case: Assume $|V| \geq 2$ and let $v \in V$. Let $G' = (V', E')$, for $V' = V \setminus \{v\}$ and $E' \subseteq V' \times V'$, and let $L' : V' \to 2^C$. Induction hypothesis: if $\text{LCOL}(V', E', L')$

returns a coloring $col_{L'}$, then $col_{L'}$ is an L'-coloring for G'. Assume $col_L = \text{LCOL}(V, E, L)$. Because $|V| \geq 2$, Algorithm 3 returns at $LC22$. Let $col_L = r \cup \{(v \mapsto c)\}$. By $LC20$, $LC21$, and the induction hypothesis, r is an L'-coloring for $G' = (V', E')$, for V', E', and L' as defined in $LC12$–$LC19$. Therefore, $col_L(v') \in L'(v')$ for all $v' \in V'$ and $col_L(v_1) \neq col_L(v_2)$ for all $(v_1, v_2) \in E'$. Let $E'' = E \setminus E'$. Because of $LC9$, $(v, v) \notin E''$. It follows by $LC13$ that for every $(v_1, v_2) \in E''$ either $v_1 = v$ or $v_2 = v$. Without loss of generality assume that $v_1 = v$. It follows that $v_2 \in V'$. By $LC17$, $col_L(v_2) \neq c$. Therefore, $col_L(v_1) \neq col_L(v_2)$. Furthermore, $col_L(v) \in L(v)$ by $LC11$. Hence, col_L is an L-coloring of G. Hence, the correctness property of Lemma 4 follows.

Completeness: Assume there exists an L-coloring col_L for G. Base case: Assume $V = \{v\}$. Because col_L is an L-coloring, $col_L(v) \in L(v)$. Furthermore, $col_L(v_1) \neq col_L(v_2)$ for all $(v_1, v_2) \in E$. Therefore, $|L(v)| \geq 1$ and $(v, v) \notin E$. It follows from $LC1$ and $LC2$ that $\text{LCOL}(\{v\}, E, L)$ returns at $LC4$ with a coloring. Step case: Assume $|V| \geq 2$ and let $v \in V$. Let $G' = (V', E')$ for $V' = V \setminus \{v\}$ and $E' \subseteq V' \times V'$, and let $L' : V' \to 2^C$. Induction hypothesis: if there exists an L'-coloring for G', then $\text{LCOL}(V', E', L')$ returns a coloring. Because $|V| \geq 2$, $\text{LCOL}(V, E, L)$ passes through $LC8$. Let v be the vertex chosen in $LC8$ and $c = col_L(v)$. Because col_L is an L-coloring for G, for all $(v_1, v_2) \in E$, $col_L(v_1) \neq col_L(v_2)$ and therefore $(v, v) \notin E$. Hence, $\text{LCOL}(V, E, L)$ executes the for-loop $LC11$–$LC22$. Algorithm 3 cannot return NO before c is chosen in $LC11$. Let V', E', and L' as defined in $LC12$–$LC19$. The coloring $col_{L'} = col_L \setminus \{(v \mapsto c)\}$ is an L'-coloring for (V', E'). Therefore, $\text{LCOL}(V', E', L')$ returns a coloring at $LC20$ by the induction hypothesis. Hence, $\text{LCOL}(V, E, L)$ returns a coloring at $LC22$. Hence, the completeness property of Lemma 4 follows. ■

C. Proofs

1) Theorem 1: The proof of Theorem 1 requires a formal definition of the parallel, (fully-)synchronized composition of two processes in terms of the operational semantics of CSP, which is a standard parallel composition of two non-deterministic LTSs. Without loss of generality, we assume now that the set of input symbols to an LTS that correspond to a process is the set of all events $\Sigma^{\tau,\checkmark}$.

Definition 12 (Operational Semantics of Parallel, Synchronized Composition) *Assume two processes P_1 and P_2. Let $L_{P_1} = (Q_{P_1}, \Sigma^{\tau,\checkmark}, \delta_{P_1}, q^0_{P_1})$ and $L_{P_2} = (Q_{P_2}, \Sigma^{\tau,\checkmark}, \delta_{P_2}, q^0_{P_2})$. An LTSs $L_{P_1 \parallel P_2} = (Q_{P_{12}}, \Sigma^{\tau,\checkmark}, \delta_{P_{12}}, q^0_{P_{12}})$ corresponding to the process $P_1 \parallel P_2$ can be constructed as follows:*

- $Q_{P_{12}} = Q_{P_1} \times Q_{P_2}$
- $\delta_{P_{12}} = \{((q_{P_1}, q_{P_2}), \sigma, (q'_{P_1}, q'_{P_2})) | (q_{P_1}, \sigma, q'_{P_1}) \in \delta_{P_1}, (q_{P_2}, \sigma, q'_{P_2}) \in \delta_{P_2}, \sigma \in \Sigma^{\checkmark}\} \cup \{((q_{P_1}, q_{P_2}), \tau, (q'_{P_1}, q_{P_2})) \mid (q_{P_1}, \tau, q'_{P_1}) \in \delta_{P_1}, q_{P_2} \in Q_{P_2}\} \cup \{((q_{P_1}, q_{P_2}), \tau, (q_{P_1}, q'_{P_2})) \mid (q_{P_2}, \tau, q'_{P_2}) \in \delta_{P_2}, q_{P_1} \in Q_{P_1}\}$
- $q^0_{P_{12}} = (q^0_{P_1}, q^0_{P_2})$

Proof of Theorem 1: Assume $\mathcal{U}$ is finite, let $\phi = (UT, S, B)$ be an authorization policy and W a finite workflow process. Let $L_W = (Q_W, \Sigma^{\tau,\checkmark}, \delta_W, q^0_W)$. Because $\mathcal{U}$ is finite, π^{-1} maps the finite number of tasks $\mathcal{T}$ of W to a finite number of execution events. We construct a finite LTS $L_{W[\pi^{-1}]} = (Q_{W[\pi^{-1}]}, \Sigma^{\tau,\checkmark}, \delta_{W[\pi^{-1}]}, q^0_{W[\pi^{-1}]})$ as follows: $Q_{W[\pi^{-1}]} = Q_W$, $\delta_{W[\pi^{-1}]} = \{(q_1, t.u, q_2) \mid (q_1, t, q_2) \in \delta_W, t \in \mathcal{T}, u \in \mathcal{U}\} \cup \{(q_1, \sigma, q_2) \mid (q_1, \sigma, q_2) \in \delta_W, \sigma \in (\Sigma^{\tau,\checkmark}_W \setminus \mathcal{T})\}$, and $q^0_{W[\pi^{-1}]} = q^0_W$. In other words, $L_{W[\pi^{-1}]}$ is the same LTS as L_W except for every transition $q_1 \xrightarrow{\langle t \rangle} q_2$ in L_W, for a task t, there is a set of transitions $q_1 \xrightarrow{\langle t.u \rangle} q_2$ in $\delta_{W[\pi^{-1}]}$, for every user $u \in \mathcal{U}$.

Consider $\phi = (UT, S, B)$. Because $\mathcal{T}$ and $\mathcal{U}$ are finite, the static authorization process A_{UT} is finite by Definition 3, every SoD process A_s, for $s \in S$, is finite by Definition 4, and every BoD process A_b, for $b \in B$, is finite by Definition 5. By Definition 6, A_ϕ is the parallel, synchronized composition of A_{UT} every A_s, for $s \in S$, and every A_b, for $b \in B$. From Definition 12, it follows that A_ϕ is finite too. Let $L_{A_\phi} = (Q_{A_\phi}, \Sigma^{\tau,\checkmark}, \delta_{A_\phi}, q^0_{A_\phi})$.

By Condition 1 of Definition 8, an enforcement process $E_{\phi,W}$ for ϕ on W must trace refine A_ϕ, *i.e.* $A_\phi \sqsubseteq_{\mathsf{T}} E_{\phi,W}$. Therefore, if $E_{\phi,W}$ exists, there exists an LTS $L_{E_{\phi,W}} = (Q_{E_{\phi,W}}, \Sigma^{\tau,\checkmark}, \delta_{E_{\phi,W}}, q^0_{E_{\phi,W}})$ for $Q_{E_{\phi,W}} = Q_{A_\phi}$, $\delta_{E_{\phi,W}} \subseteq \delta_{A_\phi}$, and $q^0_{E_{\phi,W}} = q^0_{A_\phi}$. Because A_ϕ is finite, so is δ_{A_ϕ} and there is a finite number of LTSs that are candidates to be $L_{E_{\phi,W}}$. It is straightforward to construct a process from an LTS. Because there are finitely many LTSs, there is also a finite number of corresponding processes. For each such process P, we can check if $(W[\pi^{-1}] \parallel P)[\pi] =_{\mathsf{F}} W$. Failure equivalence of finite processes is decidable [13], for example using the CSP model-checker FDR [5]. If none of the candidate processes P satisfies the check above, *i.e.* satisfies Condition 2 of Definition 8, there exists no enforcement process for ϕ on W. ■