RZ 3816 (# Z1202-002) 02/28/2012
Computer Science 29 pages

Research Report

Obstruction-Free Authorization Enforcement: Aligning Security and Business Objectives

D. Basin‡, S.J. Burri‡*, G. Karjoth*

‡Institute of Information Security,
Swiss Federal Institute of Technology (ETH)
8092 Zurich,
Switzerland

*Security Group
IBM Research – Zurich
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies (e.g., payment of royalties). Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.

Research
Almaden • Austin • Brazil • Cambridge • China • Haifa • India • Tokyo • Watson • Zurich

Obstruction-free Authorization Enforcement: Aligning Security and Business Objectives

David Basin [a], Samuel J. Burri [a,b], and Günter Karjoth [b]

[a] Institute of Information Security, ETH Zurich, 8092 Zurich, Switzerland
E-mails: {basin, samuel.burri}@inf.ethz.ch

[b] Security Group, IBM Research – Zurich, 8803 Rüschlikon, Switzerland
E-mails: {sbu, gka}@zurich.ibm.com

Access control is fundamental in protecting information systems but it can also pose 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. We formalize workflows, authorization constraints, and their enforcement using the process algebra CSP and visualize our constraints by extending the workflow modeling language BPMN. 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 complexity bounds for this problem and give an approximation algorithm that performs well when authorizations are evenly distributed among users. We provide tool support for our constraints in an extension of the modeling platform Oryx and report on the performance of our algorithms' implementation.

Keywords: Authorizations, Separation of Duty, Binding of Duty, Workflows, Obstruction, Deadlock, Release

1 Introduction

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 article, 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 the labelled transition system W on the left in 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 right 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.

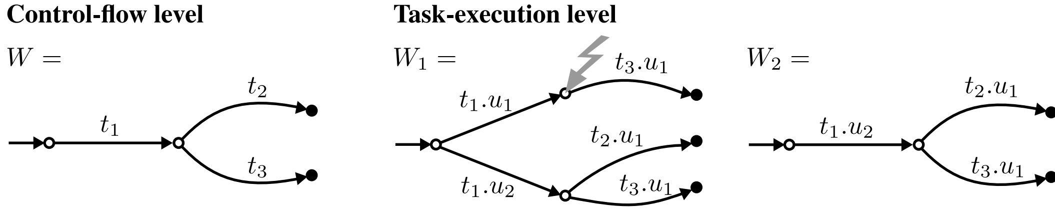


Figure 1: Enforcement with and without obstruction

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.

We distinguish authorization constraints with respect to two criteria: their dependency on previous task executions and their modeling scope. In more detail, we call an authorization constraint whose evaluation depends on who has executed previous tasks *history-dependent*; otherwise we call it *history-independent*. With respect to the modeling scope of authorizations, we distinguish *workflow-specific* and *workflow-independent* constraints. The former are designed in alignment with a given workflow and may be tailored to a workflow's specific properties such as its data-flow and control-flow. In contrast, the latter are specified without knowledge of the workflows on which they will be enforced. As such, workflow-independent authorization constraints are more generic and enforceable across different workflows, but they cannot account for workflow-specific properties.

Concretely, we consider three classes of authorization constraints. What we call *basic access control* subsumes various access control models for specifying workflow- and history-independent authorization constraints. Examples are *Access Control Lists (ACLs)* [28] and *Role-based Access Control (RBAC)* [11]. We augment basic access control with workflow-specific and history-dependent *Separation of Duty (SoD)* and *Binding of Duty (BoD)* constraints. SoD, also known as the four-eyes-principle, aims at reducing fraud and errors by preventing a user from executing tasks that result in a conflict of interest. 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 [15] and required by regulations like SOX [29]. As a result, they 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 [25] and model authorization enforcement as their parallel, synchronized composition. For graphically modeling our constraints, we propose an extension of the *Business Process Modeling Notation (BPMN)* [18], a well-established workflow modeling language. We illustrate our BPMN extension focusing on BPMN's meta-model and report on how we integrated our extension into the modeling platform Oryx [8]. Thereby, we bridge the gap between our process algebraic formalization of authorization constraints and their application 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 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. By providing tool support for our constraints through the extension of Oryx, we enable business experts to extend workflow models

with authorization constraints that are enforceable without introducing obstructions.

Our second 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 include 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.

The remainder of this article is organized as follows. In Section 2, we provide background on both CSP and BPMN. In Section 3, we formalize workflows, introduce our CSP-based approach to formalizing authorization enforcement, and provide a first instance of this approach for basic access control. In Section 4, we introduce SoD and BoD constraints, describe our extension of BPMN to model them graphically, and report on their integration into Oryx. We define obstruction-free authorization enforcement in Section 5. We then introduce **EPE**, analyze its complexity, and present algorithms to solve and approximate it. We review related work in Section 6 and draw conclusions in Section 7. The appendix provides proofs and additional background on CSP and graph coloring, which we use in our reductions. Overall, this article extends our previous paper [5] on this topic.

2 Background

2.1 CSP

We use a subset of Hoare's process algebra CSP [25] 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 $D \subseteq \Sigma$ be a subset of regular events. We write D^τ for $D \cup \{\tau\}$, $D^\checkmark$ for $D \cup \{\checkmark\}$, and $D^{\tau,\checkmark}$ for $D \cup \{\checkmark, \tau\}$. In particular, $\Sigma^{\tau,\checkmark}$ is the set of all events.

A *trace* is a sequence of regular events, possibly ending with $\checkmark$. $\langle\rangle$ 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 $i_1 \hat{} i_2$. D^* is the set of all finite traces over D and its superset $D^{*\checkmark} = D^* \cup \{i \hat{} \langle\checkmark\rangle \mid i \in D^*\}$ includes all traces ending with $\checkmark$. We abuse the set-membership operator $\in$ 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 regular 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. The expression $\sigma : D \to P$ represents a process that engages in a $\sigma \in D$ 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; $\mathsf{T}(STOP) = \{\langle\rangle\}$. 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, $P_1 \,\Box\, P_2$

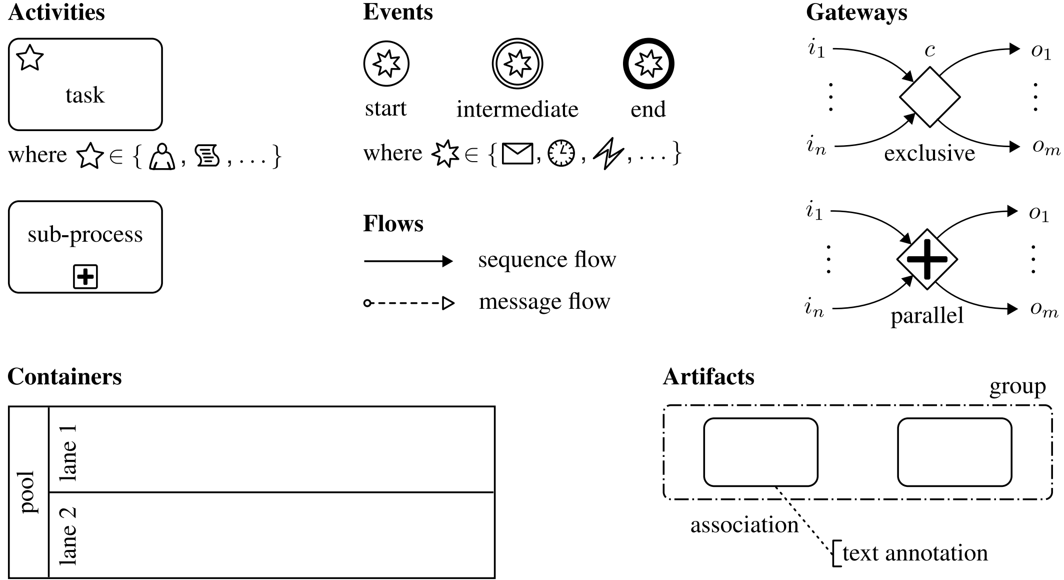


Figure 2: BPMN modeling elements

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, $\mathsf{T}(P_1 \;;\; P_2) = (\mathsf{T}(P_1) \cap \Sigma^*) \cup \{i_1{}^\frown i_2 \mid i_1{}^\frown\langle\checkmark\rangle \in \mathsf{T}(P_1), i_2 \in \mathsf{T}(P_2)\}$. Note that the invisible event τ does not appear in traces, similar to ε-transitions in nondeterministic automata. 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 $\mathsf{F}(P) = \{(i, D) \mid i \in \mathsf{T}(P), D \in rs(P \setminus i)\}$, where $P \setminus i$ represents the process P after engaging in the events in the trace i. The process 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.

2.2 BPMN

We introduce a subset of the *Business Process Modeling Notation (BPMN)* [18] that we later extend to model authorization constraints for workflows. BPMN defines graphical elements for visually modeling workflows at a high level of abstraction. BPMN calls a workflow model a *process*. In order to differentiate BPMN processes and CSP processes, we use the term *process* for CSP processes

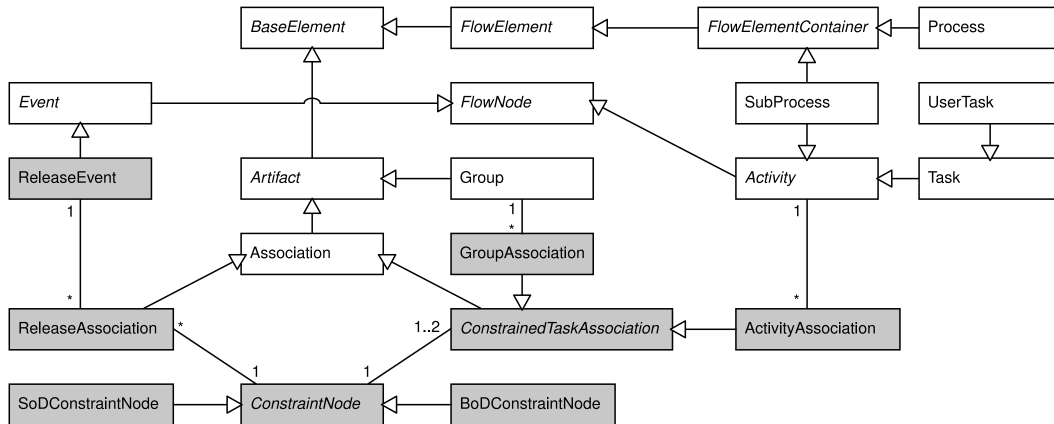


Figure 3: Extract of BPMN meta-model in white and our extensions in gray

and write explicitly *BPMN process* to refer to the concept of a process in BPMN. In this article we consider BPMN processes that are composed of the six kinds of modeling elements shown in Figure 2.

The BPMN standard [18] describes its modeling elements and their relationships in a meta-model using UML class diagrams [19]. Figure 3 shows an extract of this meta-model with the classes relevant to our BPMN extension. Italic class names denote abstract classes. We only explain the classes depicted in white for now, and return to the gray classes in Section 4.2 when we describe our BPMN extension. Based on the meta-model, the BPMN standard also specifies an XML [39] serialization for BPMN processes, which provides software vendors with a tool-independent interchange format for BPMN models. In order not to dive too deeply into XML details, we describe our BPMN extension only in terms of the meta-model. Its mapping to XML Schema [38] is straightforward. In the following, we introduce the modeling elements shown in Figure 2 and reference the corresponding meta-model classes given in Figure 3 in sans-serif font, *e.g.* Event.

BPMN calls a unit of work an *activity* (Activity). We consider two kinds of activities in this article: *tasks* (Task) and *sub-processes* (SubProcess). Tasks are visualized 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. For example, an icon depicting a script visualizes tasks that model the execution of some code. In this article, we consider mostly tasks that are executed by humans, called *user tasks* (UserTask), visualized by an icon depicting a person. Sub-processes are visualized by rectangles with rounded corners, a small boxed "+"-symbol at the bottom, and which are labelled by the name of the sub-process. A sub-process models a BPMN process that constitutes part of the parent BPMN process. The refinement of a BPMN process into sub-processes is a powerful means to model workflows at different levels of abstraction.

An *event* (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 the causal and temporal dependencies between modeling elements. A *sequence flow*, illustrated by a solid line with an arrow, defines the order in which tasks are executed and events occur. Message-based communication is modeled by *message flows*, visualized by a dashed line with a circle at the sender's end and an arrow at the recipient's end. Sequence flows and message flows together determine a workflow's control-flow.

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 on the control-flow immediately 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.

We distinguish two *containers* for partitioning BPMN processes. *Pools* are typically used to model organizational entities participating in a workflow's execution or they are simply used to demarcate a model's main BPMN process. Pools may be further subdivided into *lanes*, called *swim-lanes* in other workflow modeling languages. Control-flow within pools is typically modeled by sequence flows and control-flow across pools may only be defined in terms of message flows. A BPMN process's environment, in particular the source of input messages and the recipient of its return values, is often modeled by an empty pool [31].

Modeling elements for annotations are called *artifacts* (Artifact). Sets of tasks are defined by placing them in a dot-dashed box, called a *group* (Group). Textual annotations as visualized by a dotted line, called an *association* (Association), and a half-open box containing text.

3 Authorization-constrained Workflows

We start with an informal introduction of the life cycle of authorization-constrained workflows. Afterwards we use CSP to formalize the underlying concepts. We shall see that 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 sub-processes, each modeling a separate system aspect.

We call an atomic unit of work a *task*. A *workflow* models the causal and temporal dependencies between a set of tasks, which together implement a business objective. An alternative name for workflow is *business process*. Although we prefer the term *workflow* as it avoids further overloading the term *process*.

We distinguish two phases in a workflow's life cycle. At *design time*, a business expert designs a workflow using a modeling language such as BPMN and afterwards deploys it to a *workflow engine*. At *run time*, the workflow engine executes the workflow. We call a workflow execution a *workflow instance*. A workflow engine may execute multiple instances of the same workflow in parallel. According to the workflow's control-flow and depending on the evaluation of gateway conditions, a workflow engine schedules and instantiates tasks, called *task instances*, during workflow execution. Standard workflow modeling languages, such as BPMN, allow the specification of loops, parallel, and conditional execution. Therefore, there may be zero or more instances of the same task in one workflow instance. Depending on a task's type, its instances are executed by humans, by a software program, through the invocation of a web service, *etc.* In this article, we consider only tasks whose instances are executed by humans, either directly, *e.g.* by completing a form, or indirectly, *e.g.* by executing a program on their behalf. An *authorization (constraint)* specifies whether or not a user is allowed to execute a task instance.

3.1 Workflows

There are numerous translations from BPMN and similar workflow modeling languages to process calculi such as CSP [37] or the π-calculus [22]. 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 article, assume a set of *tasks* $\mathcal{T}$ and a set of *points* $\mathcal{O}$. Points are used to model BPMN events. We formalize workflows at the control-flow level using CSP as follows.

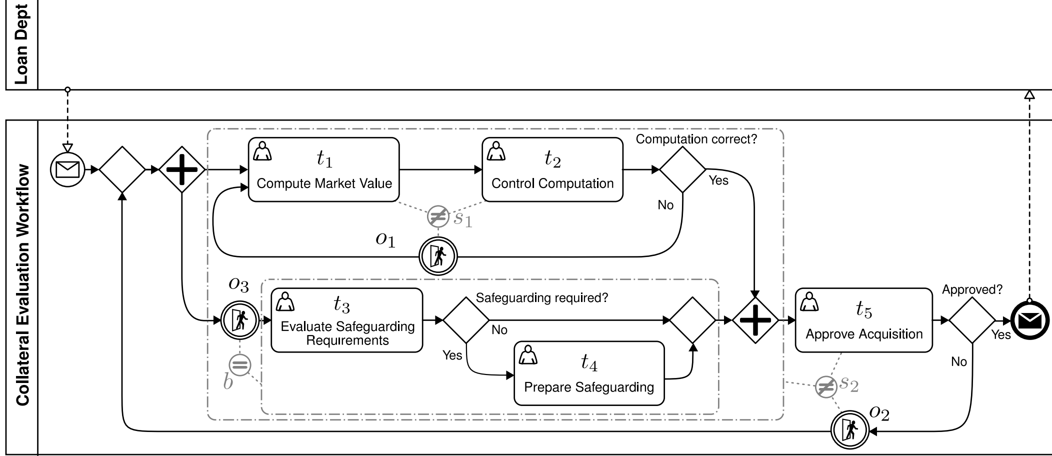


Figure 4: The collateral evaluation workflow modelled in BPMN

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, visualized in BPMN, and a corresponding workflow process. This workflow serves as a running example for the remainder of this article.

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 4 shows a BPMN model of the *collateral evaluation workflow*, which we adopted from IBM's Information FrameWork [14]. Ignore the gray 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 4. The workflow process W models the collateral evaluation workflow in CSP.

$$\begin{aligned} W &= (P_1 \mid\mid\mid P_2) \; ; \; (t_5 \to ((o_2 \to W) \sqcap SKIP)) \\ P_1 &= t_1 \to t_2 \to ((o_1 \to P_1) \sqcap SKIP) \\ P_2 &= o_3 \to t_3 \to ((t_4 \to SKIP) \sqcap SKIP) \end{aligned}$$

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

Next, we model the execution of workflows at the task-execution level. For the reminder of this article, let $\mathcal{U}$ be the set of *users*. For a task t and a user u, the CSP event $t.u$ models the execution of an instance of t by u. We call $t.u$ a *(task) execution event*. Let $\mathcal{X} = \{t.u \mid t \in \mathcal{T}, u \in \mathcal{U}\}$ be the set of all execution events. The auxiliary relation $\pi = \{(t.u, t) \mid t \in \mathcal{T}, u \in \mathcal{U}\}$ 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}, 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 $W[\pi^{-1}]$. We 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 $\mathcal{U} = \{\text{Alice, Bob, Claire, Dave}\}$ for the collateral evaluation workflow. Consider the following workflow traces:

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

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 trace of $W[\pi^{-1}]$ because t_4 can only be executed after t_3 has been executed. ◇

Example 2 supports our previous observation 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.

3.2 Authorization constraint classes and enforcement approach

We model in this article three classes of authorization constraints:

- **Basic access control:** This family encompasses all authorizations that restrict the execution of task instances to users with the necessary qualifications and responsibilities in a history-independent and workflow-independent manner. Examples are access control lists (ACLs) [28] and Role-based Access Control (RBAC) [11] configurations without sessions. What is often called a *permission* in the context of basic access control corresponds to the right to execute a task in this article.
- **Separation of Duties (SoD):** Authorizations to execute task instances are restricted to ensure that instances, whose execution results in a conflict of interest, are executed by different users. For example, consider two tasks t_1 and t_2 and suppose that their execution by the same user results in a conflict of interest. An SoD constraint is then used to prevent such a conflict of interest by not authorizing a user from executing an instance of t_2 after executing an instance of t_1 and *vice versa*.
- **Binding of Duties (BoD):** Authorizations to execute task instances are restricted based on who has executed previous task instances 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 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*.

As defined here, SoD and BoD constraints are history-dependent. We will also model them in a workflow-specific manner. Note that related work on SoD and BoD often uses the term *dynamic* for what we call *history-dependent* and *static* for *history-independent*. We implicitly subsume history-independent SoD and BoD by basic access control. A detailed comparison of our terminology and related work follows in Section 6.

We formalize authorized executions of task instances in terms of processes. More specifically, for each authorization constraint c, we define a process A_c and say that a workflow trace i *satisfies* c if $i \in \mathsf{T}(A_c)$. Given a workflow process W, we then describe the enforcement of c in W by the parallel, fully-synchronized execution of A_c and W, formally $W \parallel A_c$.

3.3 Basic access control

In the interest of ecumenical neutrality and supporting numerous access control languages, we model basic access control abstractly as a relation $UT \subseteq \mathcal{U} \times \mathcal{T}$, called the *user-task assignment*. Given a user-task assignment UT, a user u is authorized to execute instances of a task t if $(u, t) \in UT$.

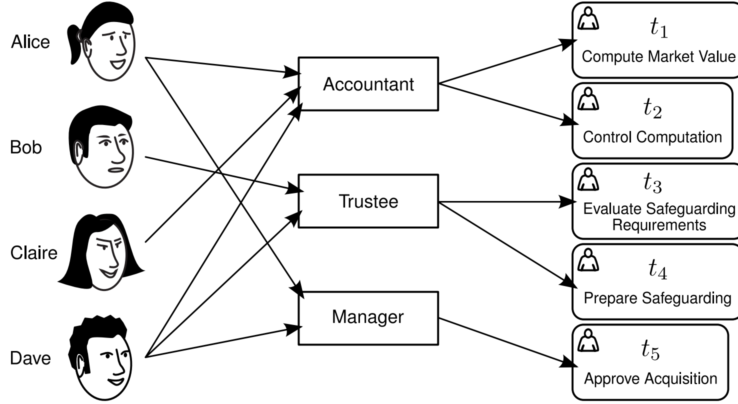


Figure 5: Role-based static authorizations

Definition 3 (Basic Authorization Process) *For a user-task assignment UT, a* basic 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 history-independent 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 mentioned in Section 3.2, we assume that basic access control policies are workflow-independent. A user-task assignment may be specified using a more refined access control model. For example, Figure 5 shows a role-based model [11] that specifies a basic access control policy for Alice, Bob, Claire, and Dave with respect to the tasks of the collateral evaluation workflow.

4 Scoping constraints with release points

We now turn to history-dependent authorizations. To separate or bind duties between tasks, we must keep track of which users executed previous instances of these tasks in order to determine who is authorized to execute future instances. Thus, for history-dependent authorizations we build up associations between task instances and users during workflow execution. Future authorization decisions in turn depend on these associations. In this section we introduce the concept of *releasing*, which removes such associations and thereby scopes history-dependent authorizations to subsets of task instances within workflow instances.

4.1 Formalization

4.1.1 SoD Constraints

Let T_1 and T_2 be two non-empty and disjoint sets of tasks , *i.e.* $|T_1| \geq 1$, $|T_2| \geq 1$, and $T_1 \cap T_2 = \varnothing$, and let O be a set of points. 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*

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

An SoD process $A_s(U_{T_1}, U_{T_2})$ offers the (external) choice between six kinds of events. (1) For a task $t \in T_1$ and user $u \in U_{T_1}$, A_s engages in the execution event $t.u$. Afterward, A_s associates u with T_1 by removing u from U_{T_2} and thereby blocking u from executing future instances of tasks in T_2. (2) Symmetrically, A_s associates a user $u \in U_{T_2}$ with T_2 and blocks u from executing future instances of tasks in T_1 after executing an instance of 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 4 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 4 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 4 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 traces of the form

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

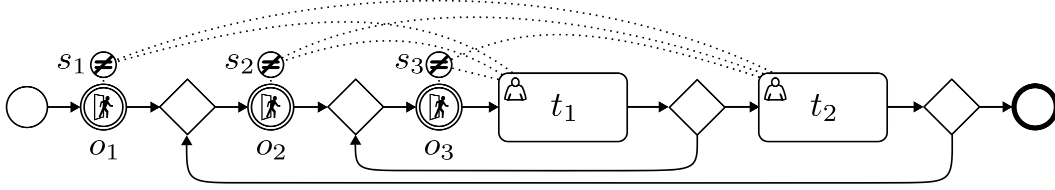


Figure 6: Location matters: The placement of a release point within a workflow effects the semantics of the respective SoD constraint

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. The 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 instances of t_1 executes instances of 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 instances of t_1 and those who execute instances of 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 differ from the user who executes t_2's instance. 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. $\diamond$

Because the semantics of an SoD constraint is only fully defined with respect to a workflow process we classify these constraints as workflow-specific. This classification is further supported by the fact that we visualize them as extensions to existing workflow models.

4.1.2 BoD Constraints

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

Let T be a non-empty set of tasks, *i.e.* $|T| \geq 1$, and let O be a set of points. 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*

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

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 an

instance of 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 instances of 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 events 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 4 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. For the same reasons as with SoD constraints, we therefore classify BoD constraints as workflow-specific.

4.1.3 Composition

Let UT be a user-task assignment, S be a set of SoD constraints, and B be a set of BoD constraints. The triple (UT, S, B), called an *authorization policy*, combines workflow-independent and history-independent authorizations in the form of UT and workflow-specific, history-dependent authorizations in the form of S and B. We define the semantics of authorization policies by composing the respective processes.

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

$$A_\phi = A_{UT} \parallel (\mathop{\parallel}_{s \in S} A_s) \parallel (\mathop{\parallel}_{b \in B} A_b) .$$

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. Given a workflow process W, we say ϕ is an *authorization policy for* W if all tasks and points in ϕ appear in W. 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 4. 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 instances of 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 an instance of t_2 and an instance of t_5, b is violated because the instances of t_3 and t_4 are not executed by the same user, and Claire is not authorized to execute instances of t_5 with respect to UT. However, i_4 satisfies ϕ. ◇

4.2 BPMN extension and serialization

We return to BPMN's meta-model introduced in Section 2.2. Figure 3 shows BPMN's meta-model classes in white and the new classes that we defined for our extension in gray. Furthermore, Figure 7 shows the concrete notation used to visualize the respective new modeling elements. New modeling elements are shown in black. How they are connected to existing or other new elements is illustrated in gray.

Let $s = (T_1, T_2, O)$ be an SoD constraint. Using our BPMN extension, s is modeled by a combination of new and existing modeling elements. The class SoDConstraintNode connects all relevant elements. Each set of tasks is either modeled by an instance of Activity or they are identified by an instance of Group. The respective classes are connected to the SoDConstraintNode by

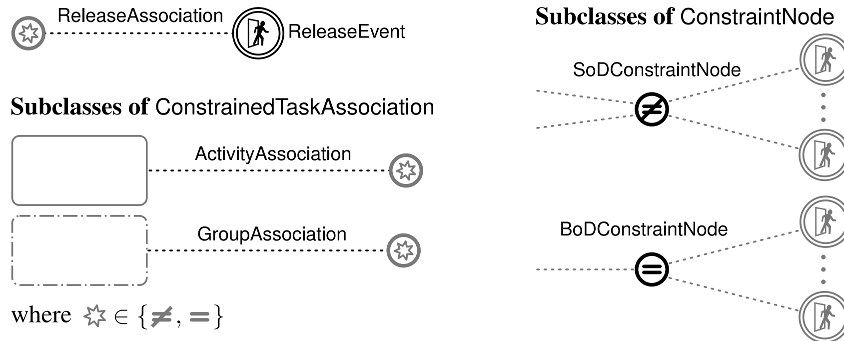


Figure 7: BPMN extension for modeling authorization constraints

instances of ConstrainedTaskAssociation. Each release point in O is modeled by an instance of class ReleaseEvent and all of them are connected to the SoDConstraintNode by instances of ReleaseAssociation. A BoD constraint $b = (T, O)$ is modeled analogously. Instead of SoDConstraintNode, the central class is BoDConstraintNode and instead of two sets of tasks only one set of tasks is connected to it.

The BPMN standard provides an extension mechanism [18]. Using the extended meta-model as a blueprint, we specified an XML schema for our BPMN extension. As Figure 3 shows, new modeling elements can be easily defined by connecting to, and inheriting from, existing elements. Correspondingly, our new schema file is only a few dozen lines long. We modeled the collateral evaluation workflow in BPMN including our extensions and serialized the model in XML. Afterward, we successfully validated the XML file against the official BPMN XML schema and the XML schema specifying our extension.

4.3 Tool support

We implemented tool support for our BPMN extension by extending the modeling platform Oryx [8]. Our objective was to gain modeling experience with our BPMN extension, demonstrate its expressivity in a hands-on fashion, and validate its ease of use.

Oryx is a good choice for our purpose. First, Oryx is designed to be extensible. As a result, our implementation required little programming effort. Second, Oryx's architecture and code is well-documented and mature. In particular, it is the basis for commercial tools such as Signavio's Process Editor [2] and the Activiti BPM Platform [1]. Third, Oryx's source code is freely available under the MIT license [20], which gives us full access to all implementation details and does not impede a potential commercial exploitation. Finally, Oryx's web-based architecture is ideal for demonstration purposes because BPMN processes are modeled directly in a web browser and no extra software need be installed.

Oryx adopts a standard three-tier architecture, with a web browser acting as the presentation tier, a J2EE server as the application tier, and a database as the data tier. The implementation provides extension mechanisms in the presentation and application tier. We report on the performance of an extension we made to the application layer in Section 5.4.3. In the following we describe our extension of the presentation layer to support our BPMN extension.

Oryx groups modeling elements and defines their visualization in so-called *stencil sets* [21]. A stencil set may extend existing stencil sets, thereby extending an existing modeling language. We defined a stencil set that specifies the modeling elements of our BPMN extension, as introduced in Section 4.2, extending Oryx's existing BPMN stencil sets. Figure 8 shows the user interface of Oryx's BPMN editor. Each palette on the left corresponds to a stencil set; BPMN's standard stencil sets are located on the top and our additional stencil set is at the bottom. A BPMN model of the

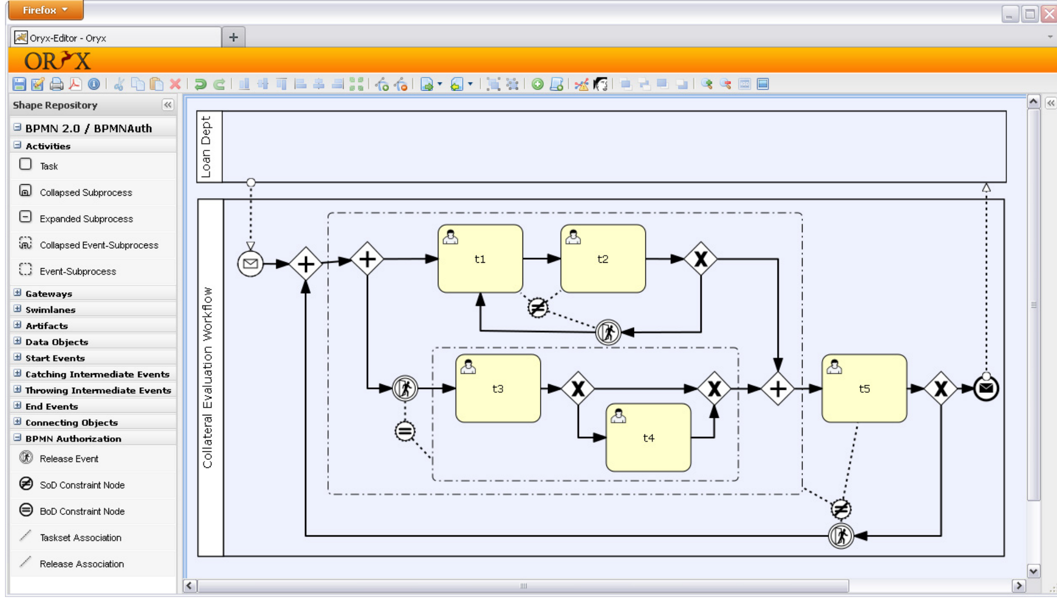


Figure 8: Screenshot of Oryx BPMN editor including authorization extension

collateral evaluation workflow is shown on the right, combining modeling elements from various stencil sets including our new one.

5 Aligning security and business objectives

In this section, we investigate the question of how to enforce an authorization policy on a workflow without obstructing the workflow's underlying business objective. To this end, we introduce the notion of an *obstruction*, formalizing the misalignment of security and business objectives. We proceed by formulating the existence of an obstruction-free authorization enforcement as a decision problem and analyzing its complexity.

5.1 Obstructions

We link the control-flow and task-execution level by the notion of an 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 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. $\diamond$

5.2 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 with $W[\pi^{-1}]$, formally $W[\pi^{-1}] \parallel E$.

Definition 8 (Enforcement Process) *Let a workflow process W and an authorization policy ϕ for W be given. 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 is indistinguishable from the process W constrained by $E_{\phi,W}$.

Suppose $E_{\phi,W}$ is an enforcement process for ϕ on W. By CSP's traces model, if $i \in \mathsf{T}(W[\pi^{-1}] \parallel E_{\phi,W})$ then $i \in \mathsf{T}(W[\pi^{-1}])$ and $i \in \mathsf{T}(E_{\phi,W})$. For a task t, it follows by the failure equivalence of $(W[\pi^{-1}] \parallel E_{\phi,W})[\pi]$ and W, *i.e.* Condition (2), that if $i[\pi]\hat{}t \in \mathsf{T}(W)$ then there exists a user u such that $i\hat{}\langle t.u\rangle \in \mathsf{T}(E_{\phi,W})$. Therefore, $i\hat{}\langle t.u\rangle$ satisfies ϕ because of Condition (1). Hence, i is not obstructed and $E_{\phi,W}$ 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.

$$\begin{aligned} E &= (E_1 \mid\mid\mid E_2)\ ;\ (t_5.\mathsf{Dave} \rightarrow ((o_2 \rightarrow E) \sqcap \mathit{SKIP})) \\ E_1 &= t_1.\mathsf{Alice} \rightarrow t_2.\mathsf{Claire} \rightarrow ((o_1 \rightarrow E_1) \sqcap \mathit{SKIP}) \\ E_2 &= o_3 \rightarrow t_3.\mathsf{Bob} \rightarrow ((t_4.\mathsf{Bob} \rightarrow \mathit{SKIP}) \sqcap \mathit{SKIP}) \end{aligned}$$

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. Thus, E is an enforcement process for ϕ on W. $\diamond$

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.

5.3 The enforcement process existence problem

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

Definition 9 (The Enforcement Process Existence Problem **EPE**)

Input: *A workflow process* W *and an authorization policy* ϕ.

Output: YES *if there exists an enforcement process for* ϕ *on* W *or* 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**.

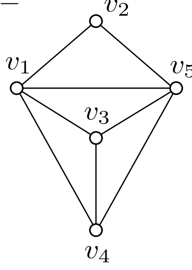
Lemma 1 **EPE** *is* **NP**-*hard.*

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 and only 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 9 illustrates this construction for a graph with $n = 5$ and $k = 4$.

Input

$k = 4$

$G =$



Construction

$\mathcal{T} = \{v_1, ..., v_5\}$ $\quad\mathcal{U} = \{1, 2, 3, 4\}$

$UT = \mathcal{U} \times \mathcal{T}$ $\quad S = \{(v_1, v_2, \varnothing), (v_1, v_3, \varnothing), ...\}$ $\quad B = \varnothing$

$W = v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to SKIP$

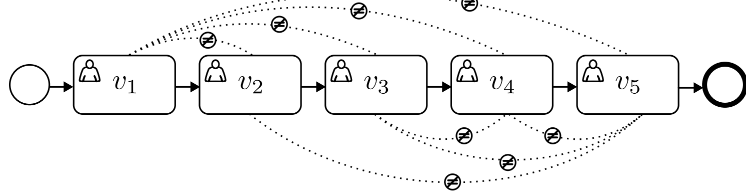


Figure 9: Illustration of polynomial reduction from k**-COLORING** to **EPE**

Let $h = \langle v_1, v_2, \ldots, v_n, \checkmark\rangle$. Note that by the construction of W, $h \in \mathsf{T}(W)$. 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.

Conversely, let $col : V \to \{1, \ldots, k\}$ be a k-coloring for G and consider the process $P = v_1.col(v_1) \to v_2.col(v_2) \to \ldots \to v_n.col(v_n) \to SKIP$. Let $i = \langle v_1.col(v_1), v_2.col(v_2), \ldots, v_n.col(v_n), \checkmark\rangle$. By our construction and because col is a k coloring of G, $i \in \mathsf{T}(A_s)$ for every $s \in S$. Furthermore, by our definition of UT, $i \in \mathsf{T}(A_{UT})$. It follows by Definition 6 and $B = \varnothing$ that $i \in \mathsf{T}(A_\phi)$. Because every trace in $\mathsf{T}(P)$ is a prefix of i, $A_\phi \sqsubseteq_{\mathsf{T}} P$. Furthermore, $P[\pi] =_{\mathsf{F}} W$ and therefore $(W[\pi^{-1}] \parallel P)[\pi] =_{\mathsf{F}} W$. Hence, P is an enforcement process for ϕ on W by Definition 8.

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.

Proof Sketch. If $\mathcal{U}$ and W are finite, it follows by Definitions 3–5 and the operational semantics of CSP that A_ϕ is finite too. 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 [24], 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. $\square$

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 CSP's 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 [24]. 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.

5.4 Approximations

We now present an approximation algorithm EPEA for **EPE**. EPEA is an approximation in that it may return NO even when an enforcement process for the given input exists. However, EPEA makes no approximation error in the opposite case: if there does not exist an enforcement process for ϕ on W, then EPEA's output is always NO. EPEA has an exponential runtime complexity. We show in a second step how to change EPEA to approximate **EPE** in polynomial time using bounds from graph-coloring.

5.4.1 Exponential approximation

EPEA, defined in Algorithm 1, takes an instance of **EPE** as input and returns NO or a relation that can be transformed to an enforcement process for the given **EPE** instance. In detail, EPEA is defined as the composition of CGRAPH and LCOL. CGRAPH, defined 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. These problems and graph-coloring terminology are reviewed 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.

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 1: EPEA(T, ϕ)

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

```
1 if T = ∅ then
2 |  return ∅
3 else
4 |  col_L ← LCOL(CGRAPH(T, φ))
5 |  if CGRAPH or LCOL return NO then
6 |  |  return NO
7 |  else
8 |  |  return {(t, t.u) | (T ↦ u) ∈ col_L, t ∈ T}
```

Example 7 (Graph Returned by CGRAPH) Figure 10 depicts 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. ◇

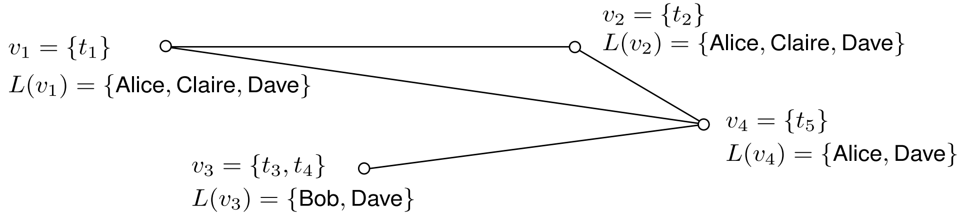


Figure 10: Constraint graph of the collateral evaluation workflow

LCOL is a standard algorithm for solving **LISTCOLORING**. In Appendix B, we described LCOL 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 fails to build 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* EPEA(T, ϕ) *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 EPEA(T, ϕ) returns a relation R. We refer to a line i of CGRAPH as CGi and to line j of EPEA as EAj.

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 ✓, 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 $T \neq \varnothing$, CGRAPH(T, ϕ) returns a graph (V, E) and a function L by EA1, EA4, and EA5. Furthermore, 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.

Algorithm 2: CGRAPH(T, ϕ)

```
   Input: T and φ = (UT, S, B)
   Output: returns a graph (V, E) and a list coloring function L : V → 2^U or NO
 1 V, E, L ← ∅
 2 foreach t ∈ T do
 3 |  V ← V ∪ {{t}}
 4 |_ L ← L ∪ {({t} ↦ {u | (u, t) ∈ UT})}
 5 foreach (T1, O) ∈ B do
 6 |  pick a t1 ∈ T1
 7 |  let v1 ∈ V s.t. t1 ∈ v1
 8 |  foreach t2 ∈ T1 \ {t} do
 9 |  |  let v2 ∈ V s.t. t2 ∈ v2
10 |  |  V ← (V \ {v1, v2}) ∪ {v1 ∪ v2}
11 |_ |_ L ← (L \ {(v1 ↦ L(v1)), (v2 ↦ L(v2))}) ∪ {(v1 ∪ v2 ↦ L(v1) ∩ L(v2))}
12 foreach (T1, T2, O) ∈ S do
13 |  foreach t1 ∈ T1 do
14 |  |  let v1 ∈ V s.t. t1 ∈ v1
15 |  |  foreach t2 ∈ T2 do
16 |  |  |  let v2 ∈ V s.t. t2 ∈ v2
17 |  |  |  if v1 ≠ v2 then
18 |  |  |  |_ E ← E ∪ {(v1, v2)}
19 |  |  |  else
20 |_ |_ |_ |_ return NO
21 return V, E, L
```

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 traces model and because $R \subseteq \pi^{-1}$, $(W[\pi^{-1}] \parallel W[R])[\pi] =_{\mathsf{F}} W[R][\pi] =_{\mathsf{F}} W$. ■

5.4.2 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 to execute a task with respect to the user-task assignment and the BoD constraints, then there exists an enforcement process. Simplified, there exists an enforcement process if the set of users is large and their static authorizations are evenly 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.

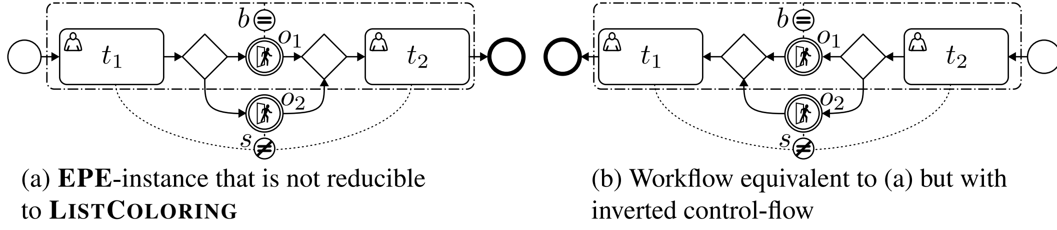
5.4.3 Experimental results and evaluation

As presented in Section 4.3, how our extension of Oryx's presentation tier enables us to graphically model workflows including SoD and BoD constraints. Additionally, we extended Oryx by a window for specifying user-task assignments. We now describe how we analyze **EPE**-instances, which are specified using these extensions in Oryx's presentation tier, with our EPEA implementation in Oryx's application tier.

For performance reasons, we do not use LCOL to solve **LISTCOLORING**-instances returned by CGRAPH. Instead, we transform them into Boolean formulae with a variable for each vertex-color combination and clauses encoding the coloring constraints imposed by edges and the requirement that a color must be chosen for every vertex. Such a reduction is standard and we therefore omit a correctness proof. We then use the SAT-solver *sat4j* [6] to compute satisfying assignments for these formulae. Transforming assignments back into colorings for the initial **LISTCOLORING**-instances is straightforward and if a formula is unsatisfiable then no coloring exists.

We decompose the total running time required for solving an **EPE**-instance into three parts. The *communication time* is the time required to send the **EPE**-instance from Oryx's presentation tier to its application tier and finally to return the result back to the presentation tier. The *transformation time* is the time it takes to transform the **EPE**-instance to a Boolean formula and the *solving time* is the time it takes to compute a satisfying assignment for the formula with sat4j.

The communication time depends on various factors such as the network throughput, latency, and the payload size. We run the presentation tier and the application tier on two different computers, which are connected by a standard enterprise network with an average latency of 1 millisecond. Computing an enforcement process for our running example has on the average a communication time of 100 milliseconds, a transformation time of 80 milliseconds, and a solving time of 15 milliseconds, summing up to a total running time of 200 milliseconds. We also tested our implementation with random user-task assignments with up to 50 users and could observe a minor increase in the



(a) **EPE**-instance that is not reducible to **LISTCOLORING**

(b) Workflow equivalent to (a) but with inverted control-flow

Figure 11: Limitations of abstractions made by EPEA

solving time while the communication and transformation time remained relatively stable. However, with these numbers of users, the communication time still overshadows the solving time.

By means of examples, we now illustrate the limitations of our graph-based **EPE**-approximation. These examples also illustrate the expressivity of enforcement processes and properties of the **EPE** problem in general. Example 8 shows a case where EPEA returns NO even though there exists an enforcement process for the given **EPE**-instance.

Example 8 (Release points matter) Consider the workflow shown in Figure 11 (a), which we formalize by the workflow process $W_a = t_1 \rightarrow ((o_1 \rightarrow t_2 \rightarrow SKIP) \sqcap (o_2 \rightarrow t_2 \rightarrow SKIP))$. Furthermore, let $b = (\{t_1, t_2\}, o1)$ and $s = (t_1, t_2, o_2)$ be the BoD and SoD constraints shown in Figure 11 (a) and assume the user-task assignment $UT = \{(\mathsf{Alice}, t_1), (\mathsf{Alice}, t_2), (\mathsf{Bob}, t_2)\}$. Given $T = (t_1, t_2)$ and $\phi = (UT, \{s\}, \{b\})$, CGRAPH fails to produce an instance of **LISTCOLORING** because t_1 and t_2 are both bound and separated by b and s, respectively. However, no matter how the workflow is executed, when t_2 is instantiated either b's or s's history has been released and the respective constraint is irrelevant at this point. Correspondingly, $E = t_1.\mathsf{Alice} \rightarrow ((o_1 \rightarrow t_2.\mathsf{Bob} \rightarrow SKIP) \sqcap (o_2 \rightarrow t_2.\mathsf{Alice} \rightarrow SKIP))$ is an enforcement process for ϕ on W_a. ◇

This example shows that by abstracting away from release points, our approximation misses information needed to determine whether an enforcement process exists. Moreover, our approximation not only abstracts away release points but also control-flow. Example 9 illustrates how the existence of an enforcement process may only depend on control-flow.

Example 9 (Control-flow matters) Consider the **EPE**-instance shown in Figure 11 (b), which is equivalent to the **EPE**-instance examined in Example 8 except that the workflow's control-flow is inverted. Let $W_b = t_2 \rightarrow ((o_1 \rightarrow t_1 \rightarrow SKIP) \sqcap (o_2 \rightarrow t_1 \rightarrow SKIP))$ be the corresponding workflow process. There exists no enforcement process for ϕ on W_b because when t_2 is instantiated one cannot know whether the execution will later pass through o_1 or o_2 and whether Alice or Bob should therefore execute t_2's instance. ◇

6 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 [30]. 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* [17], have been proposed. We follow another direction by incorporating knowledge about the system's control-flow, modeled 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.

modeling scope \ history dependence	history-independent	history-dependent
workflow-independent	basic access control	SoDA
workflow-specific	lanes	SoD and BoD constraints

administrable
non-administrable

Figure 12: Classification

Expressivity of enforcement processes. An authorization policy is sometimes called *satisfiable* with respect to a workflow if there exists an assignment of users to tasks that does not violate the policy, see *e.g.* [7,33]. 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 [32].

The question of what constitutes a well-formed workflow model has been extensively investigated in the business process community. For example, van der Aalst calls a workflow *sound* if it has no dead transitions and it does not deadlock before completing its final task [34]. Obstruction-freedom is a complementary property; combined with soundness it guarantees that the workflow will always successfully terminate and thus will achieve its business objectives.

Workflow abstraction and constraint models. Early work on authorization constraints, such as the *Transaction Control Expressions* proposed by Sandhu [27], 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 [7]. Later, Tan, Crampton, and Gunter refine a workflow to be a partially ordered set of tasks and thereby explicitly model workflows and task instances [33]. 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 [32], who models a workflow as a directed graph. However, constraints in the presence of loops are restricted such that the first task must always be executed by the same person. Given a sufficient number of users per task, this restriction ensures that a workflow instance can always successfully terminate if there are no conflicts between SoD and BoD constraints. The graph transformation used in CGRAPH is inspired by Solworth's conflict graph [32].

We impose restrictions neither on workflows nor 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 a fine-grained control of constraints in the presence of loops, scoping authorization constraints to subsets of task instances.

Classification of authorization constraints for workflows. Figure 12 shows the classification of our authorization constraints with respect to the criteria introduced in Section 1. We added in gray a third criteria distinguishing between constraints that may be administrated at run time, *i.e.* changed during workflow execution, and constraints that are not administrable at run time. In this article we implicitly assumed that constraints are non-administrable and classified basic access control constraints as both workflow-independent and history-independent, and our SoD and BoD constraints as workflow-specific and history-dependent.

Most of the authorization models mentioned above, *e.g.* [7,27,32,33], are also workflow-specific and history-dependent. However, there are exceptions, such as Li and Wang's *Separation of Duty Algebra (SoDA)* [16], that are history-dependent but workflow-independent. Li and Wang reason that SoDA is well suited to formalize abstract SoD requirements and thereby bridges the gap between high-level, *i.e.* workflow-independent, requirements and their implementation in a workflow environment.

Lanes are often used to group tasks with respect to the organizational unit that is responsible for their execution [31]; sometimes lanes are directly interpreted as workflow-specific roles. As such, they represent workflow-specific, history-independent authorization constraints. However, if tasks can be executed by multiple roles, a workflow may have exponentially many lanes in the number of roles, which complicates its graphical representation.

We are only aware of a few publications that analyze the administration of authorization constraints during workflow execution. Crampton and Khambhammettu [10] investigate how to delegate rights to execute tasks at run time without obstructing the underlying workflow and delegation can be interpreted as temporary administrative changes. In [4] we propose an enforcement mechanism for SoDA constraints, which also accounts for changing user-task assignments, reflecting administrative activities. Related work, *e.g.* [13], uses often the term *dynamic* for what we call *history-dependent* and *static* for *history-independent*. However, if you distinguish authorization constraints both with respect to their dependency on a workflows execution history and with respect to whether they are administrable at run time, the term *dynamic* is not sufficiently refined. Hence, we avoid the term *dynamic* in this article.

BPMN extension and tool support. Different authors use BPMN's extension mechanism to augment workflow models with security requirements [23,35]. We see this approach as an instance of *Model Driven Security (MDS)* [3], which aims at synthesizing a system's implementation from the composition of models specifying its business and security requirements. Our BPMN and Oryx extension builds on Rüegger's master thesis [26]. Wolter, Miseldine, and Meinel also extend Oryx to provide tool support for their workflow authorization language [36]. In particular, they invoked the model checker SPIN in Oryx's application tier to reason about their constraint models.

7 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 a system's business objectives, and the task-execution level, modeling who executes which tasks. 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 extended the well-established workflow modeling language BPMN to visualize our constraints. Furthermore, we extended the modeling platform Oryx to provide tool support for them. We thus maintain the intuition and visual appeal of graphical modeling languages, making it easier for workflow designers and security administrators to cooperate in specifying and aligning security and business objectives.

Our work gives rise to many interesting follow-up 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 possibility 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.

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] Activiti BPM Platform, `www.activiti.org`, 2012.

[2] Signavio, `www.signavio.com`, 2012.

[3] D. Basin, J. Doser, and T. Lodderstedt, Model driven security: From uml models to access control infrastructures, *ACM Transactions on Software Engineering Methodologies (TOSEM)*, **15**(1) (2006), 39–91.

[4] D. Basin, S. J. Burri, and G. Karjoth, Dynamic enforcement of abstract separation of duty constraints, in: *ESORICS'09: Proc. of the 14th European Symposium on Research in Computer Security*, Springer-Verlag, Berlin, Germany, 2009, pp. 250–267.

[5] D. Basin, S. J. Burri, and G. Karjoth, Obstruction-free authorization enforcement: Aligning security with business objectives, in: *CSF'11: Proc. of the 24th IEEE Computer Security Foundations Symposium*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2011, pp. 99–113.

[6] D. Le Berre and A. Parrain, The sat4j library, release 2.2, *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, **7**(2-3) (2010), 59–6.

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

[8] Business process technology group at Hasso-Plattner-Institute, Oryx. `http://bpt.hpi.uni-potsdam.de/Oryx`, 2012.

[9] G. Chartrand and P. Zhang, *Chromatic Graph Theory*, Chapman & Hall, 2008.

[10] J. Crampton and H. Khambhammettu, Delegation and satisfiability in workflow systems, in: *SACMAT'08: Proc. of the 13th ACM Symposium on Access Control Models and Technologies*, ACM, New York, NY, USA, 2008, pp. 31–40.

[11] 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)*, **4**(3) (2001), 224–274.

[12] Formal Systems (Europe) Ltd, *Failures-divergence refinement, FDR2 user manual*, 2005.

[13] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo, On the formal definition of separation-of-duty policies and their composition, in: *S&P'98: Proc. of the 19th IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 172–183.

[14] IBM information framework (IFW), `www.ibm.com/software/industry/banking`, 2012.

[15] IT Governance Institute, *Control objectives for information and related technology (COBIT) 4.1*, 2005.

[16] N. Li and Q. Wang, Beyond separation of duty: An algebra for specifying high-level security policies, *Journal of the ACM (JACM)*, **55**(3) (2008), 12:1–12:46.

[17] J. Ligatti, L. Bauer, and D. Walker, Edit automata: Enforcement mechanisms for run-time security policies, *International Journal of Information Security*, **4**(1-2) (2005), 2–16.

[18] Object Management Group (OMG), *Business Process Model and Notation (BPMN), v2.0*, OMG Standard, 2011.

[19] Object Management Group (OMG), *Unified Modeling Language (UML)*, OMG Standard, 2011.

[20] Open Source Initiative (OSI), *The MIT License*, `www.opensource.org/licenses/MIT`, 2012.

[21] D. Polak, *Oryx – BPMN stencil set implementation*, Bachelor Thesis, Hasso-Plattner-Institute, 2007.

[22] F. Puhlmann and M. Weske, Using the π-calculus for formalizing workflow patterns, in: *BPM '05: Proc. of the 3rd International Conference on Business Process Management*, Springer-Verlag, Berlin, Germany, 2005, pp. 153–168.

[23] A. Rodríguez, E. Fernández-Medina, and M. Piattini, A BPMN extension for the modeling of security requirements in business processes, *IEICE Transactions on Information and Systems*, **E90-D**(4) (2007), 745–752.

[24] A. W. Roscoe, Model-checking CSP, *A classical mind: Essays in honour of C. A. R. Hoare*, Prentice Hall, Hertfordshire, UK, 1994, pp. 353–378.

[25] A. W. Roscoe, *The theory and practice of concurrency*, Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[26] D. Rüegger, *Tool support for authorization-constrained workflows*, Master's thesis, ETH Zurich, 2011.

[27] R. S. Sandhu, Transaction control expressions for separation of duties, in: *Proc. of the 4th IEEE Aerospace Computer Security Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1988, pp. 282–286.

[28] R. S. Sandhu and P. Samarati, Access control: Principle and practice, *IEEE Communications Magazine*, **32**(9) (1994), 40–48.

[29] *Sarbanes-Oxley Act of 2002*, Public Law 107-204 (116 Statute 745), United States Senate and House of Representatives in Congress, 2002.

[30] F. B. Schneider, Enforceable security policies, *ACM Transactions on Information and System Security (TISSEC)*, **3**(1) (2000), 30–50.

[31] B. Silver, *BPMN Method & Style*, Cody-Cassidy Press, 2009.

[32] J. A. Solworth, Approvability, in: *ASIACCS'06: Proc. of the 1st ACM Symposium on Information, Computer and Communications Security*, ACM, New York, NY, USA, 2006, pp. 231–242.

[33] K. Tan, J. Crampton, and C. A. Gunter, The consistency of task-based authorization constraints in workflow systems, in: *CSFW'04: Proc. of the 17th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2004, pp. 155–169.

[34] W. M. P. van der Aalst, The application of petri nets to workflow management, *Journal of Circuits, Systems, and Computers (JCSC)*, **8**(1) (1998), 21–66.

[35] C. Wolter and A. Schaad, Modeling of task-based authorization constraints in BPMN, in: *BPM'07: Proc. of the 5th International Conference on Business Process Management*, Springer-Verlag, Berlin, Germany, 2007, pp. 64–79.

[36] C. Wolter, P. Miseldine, and C. Meinel, Verification of business process entailment constraints using SPIN, in: *ESSoS'09: Proc. of the 1st International Symposium on Engineering Secure Software and Systems*, Springer-Verlag, Berlin, Germany, 2009, pp. 1–15.

[37] P. Y. H. Wong and J. Gibbons, A process-algebraic approach to workflow specification and refinement, in: *SC'07: Proc. of the 6th International Symposium on Software Composition*, Springer-Verlag, Berlin, Germany, 2007, pp. 51–65.

[38] World Wide Web Consortium (W3C), *XML Schema (XSD)*, `www.w3.org/XML/Schema`, 2010.

[39] World Wide Web Consortium (W3C), *Extensible Markup Language (XML)*, `www.w3.org/XML`, 2011.

A CSP

A labelled transition system (LTS) is a quadruple (Q, D, δ, q_0), where Q is a set of states, D is a set of input symbols, $\delta \subseteq Q \times D \times Q$ is a 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 $\langle \sigma_1, \ldots, \sigma_n \rangle \in D^*$, we write $q_0 \xrightarrow{\langle \sigma_1, \ldots, \sigma_n \rangle} q_n$ if there exists a sequence of states $\langle q_1, \ldots, q_{n-1} \rangle$ 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.* $D \subseteq \Sigma^{\tau,\checkmark}$. Let L be an LTS $(Q, \Sigma^{\tau,\checkmark}, \delta, q_0)$. For $q_1, q_2 \in Q$ and a trace $i \in \Sigma^{*\checkmark}$, we write $q_1 \overset{i}{\Longrightarrow} q_2$ 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 $q \in Q$ be a state and $D \subseteq \Sigma^\checkmark$ be a set of events. The set D is a *refusal set* of q, written $q \mathit{ref} D$, if $D \subseteq \{\sigma \in \Sigma^\checkmark \mid \neg\exists q' \in Q, (q, \sigma, q') \in \delta\}$. We say an LTS $L = (Q, \Sigma^{\tau,\checkmark}, \delta, q_0)$ *corresponds*[1] to a process P, denoted L_P, if

$$\mathsf{F}(P) = \{(i, D) \mid \exists q \in Q, q_0 \overset{i}{\Longrightarrow} q, q \mathit{ref} D\} \cup \{(i, D) \mid \exists q \in Q, q_0 \overset{i\,\hat{}\,\langle\checkmark\rangle}{\Longrightarrow} q, D \subseteq \Sigma^\checkmark\}\ .$$

Note that there may be multiple LTSs that correspond to the same process.

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)

Input: *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) or* 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 [9]. The following problem generalizes k-**COLORING**.

Definition 11 (The **LISTCOLORING** problem)

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

Output: YES *if there exists a function $col_L : V \to Z$ such that for every vertex $v \in V$, $col_L(v) \in L(v)$ and for every edge $(v_1, v_2) \in E$, $col_L(v_1) \neq col_L(v_2)$ or* 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 $L(v) \subseteq Z$. 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.

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

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 [9]. See also [9] for a proof of Lemma 3.

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

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

```
   Input: |V| ≥ 1, E ⊆ V × V, and L : V → 2^Z, for a set Z
   Output: returns an L-coloring for (V, E) if it exists and NO otherwise
 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 |   |   r ← LCOL(V', E', L')
21 |   |   if r ≠ NO then
22 |   |   |   return r ∪ {(v ↦ c)}
23 |   return NO
```

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

Correctness: Base case: Assume $V = \{v\}$ and let $col_L = \{v \mapsto z\} = \text{LCol}(\{v\}, E, L)$, for $z \in Z$. 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 $V' = V \setminus \{v\}$, $E' \subseteq V' \times V'$, and $L' : V' \to 2^Z$. 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 $col_L = r \cup \{(v \mapsto z)\}$. 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, $col_L(v_2) \neq z$. 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 and 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 $V' = V \setminus \{v\}$, $E' \subseteq V' \times V'$, and $L' : V' \to 2^Z$. 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 $z = col_L(v)$. 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 z is chosen in LC11. Let V', E', and L' as defined in LC12–LC19. The coloring $col_{L'} = col_L \setminus \{(v \mapsto z)\}$ 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 and the completeness property of Lemma 4 follows. ■

C Proof of 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 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) *Let P_1 and P_2 be two processes and 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 LTS $L_{P_1 \parallel P_2} = (Q^{P_{12}}, \Sigma^{\tau,\checkmark}, \delta^{P_{12}}, q_0^{P_{12}})$ corresponding to the process $P_1 \parallel P_2$ can be constructed as follows:*

- $Q^{P_{12}} = Q^{P_1} \times Q^{P_2}$
- $\delta^{P_{12}} = \{((q_1^{P_1}, q_1^{P_2}), \sigma, (q_2^{P_1}, q_2^{P_2})) \mid (q_1^{P_1}, \sigma, q_2^{P_1}) \in \delta^{P_1},\ (q_1^{P_2}, \sigma, q_2^{P_2}) \in \delta^{P_2},\ \sigma \in \Sigma^{\checkmark}\}$
  $\cup \{((q_1^{P_1}, q^{P_2}), \tau, (q_2^{P_1}, q^{P_2})) \mid (q_1^{P_1}, \tau, q_2^{P_1}) \in \delta^{P_1},\ q^{P_2} \in Q^{P_2}\}$
  $\cup \{((q^{P_1}, q_1^{P_2}), \tau, (q^{P_1}, q_2^{P_2})) \mid (q_1^{P_2}, \tau, q_2^{P_2}) \in \delta^{P_2}, q^{P_1} \in Q^{P_1}\}$
- $q_0^{P_{12}} = (q_0^{P_1}, q_0^{P_2})$

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: $Q^{W[\pi^{-1}]} = Q^W$, $\delta^{W[\pi^{-1}]} =$

$\{(q_1, t.u, q_2) \mid (q_1, t, q_2) \in \delta^W, t \in \mathcal{T}, u \in \mathcal{U}\} \cup \{(q_1, \sigma, q_2) \mid (q_1, \sigma, q_2) \in \delta^W, \sigma \in (\Sigma^{\tau,\checkmark} \setminus \mathcal{T})\}$, 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 $L_{W[\pi^{-1}]}$, for every user $u \in \mathcal{U}$.

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 are finitely many 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 are also finitely many 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 [24], for example using the CSP model-checker FDR [12]. If none of the candidate processes P satisfies the above check, *i.e.* satisfies Condition (2) of Definition 8, there exists no enforcement process for ϕ on W. ■