

Tool support for authorization-constrained workflows

Master Thesis

Author(s):

Rüegger, Dominik

Publication date:

2011

Permanent link:

<https://doi.org/10.3929/ethz-a-006660170>

Rights / license:

[In Copyright - Non-Commercial Use Permitted](#)

MASTER'S THESIS

Tool Support for Authorization-Constrained Workflows

Dominik Rügger

July 2011

Supervisors:

Prof. Dr. David BASIN
ETH Zürich

Samuel J. BURRI
IBM Research – Zürich

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair of Information Security
Department of Computer Science
ETH Zürich

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 3 |
| 1.1 | Motivation | 3 |
| 1.2 | Outline | 3 |
| 1.3 | Contributions | 4 |
| 1.4 | Technology Overview | 4 |
| 2 | Background | 6 |
| 2.1 | Notation | 6 |
| 2.2 | BPMN | 6 |
| 2.2.1 | Concepts and Modeling Elements | 7 |
| 2.2.2 | Examples | 8 |
| 2.2.3 | XML Serialization | 9 |
| 2.2.4 | Extension Mechanism | 12 |
| 2.3 | CSP | 12 |
| 2.3.1 | Relations: Definitions and Properties | 12 |
| 2.3.2 | Notation and Definitions | 12 |
| 2.3.3 | Style | 14 |
| 3 | Authorization-Constrained Workflows | 17 |
| 3.1 | Framework | 17 |
| 3.1.1 | Workflow and Execution Model | 17 |
| 3.1.2 | Constraint Model | 18 |
| 3.1.3 | Functions | 18 |
| 3.2 | The [BBK11] Constraint Model | 19 |
| 3.2.1 | Model | 19 |
| 3.2.2 | Properties | 19 |
| 3.2.3 | Authorization Processes | 20 |
| 3.2.4 | Properties of the Authorization Processes | 21 |
| 4 | Extending BPMN to Support SoD/BoD Authorization | 25 |
| 4.1 | Extending the Graphical and Abstract Syntax Models | 25 |
| 4.1.1 | Concepts and Graphical Elements | 25 |
| 4.1.2 | Graphical Representation of Sets of Tasks | 26 |
| 4.1.3 | Mapping BPMN Process Models to CSP Processes | 27 |
| 4.1.4 | Mapping the Formal Model to the Graphical Representation | 27 |
| 4.1.5 | Mapping the Graphical Representation to the Formal Model | 27 |
| 4.1.6 | Examples | 28 |
| 4.1.7 | Abstract Syntax Model of the Extension | 29 |
| 4.2 | Concrete Syntax: Extending the XML Model | 30 |

| | | |
|----------|---|-----------|
| 5 | Constraint Analysis and Enforcement | 34 |
| 5.1 | Definitions | 34 |
| 5.1.1 | Enforcement Process | 34 |
| 5.1.2 | Enforcement Process Existence (EPE) Problem | 34 |
| 5.2 | Approximative Enforcement Processes | 35 |
| 5.2.1 | Satisfying Task-User Assignments | 35 |
| 5.2.2 | Static Approximative Enforcement Process | 36 |
| 5.2.3 | Dynamic Approximative Enforcement Process | 41 |
| 5.3 | Computing a Satisfying Task-User Assignment | 44 |
| 5.4 | Evaluation | 46 |
| 6 | Tool Support | 47 |
| 6.1 | Purpose | 47 |
| 6.2 | Choice of Implementation | 47 |
| 6.3 | Oryx Architecture | 47 |
| 6.4 | Oryx Editor Principles | 48 |
| 6.5 | Oryx Extension Mechanisms | 48 |
| 6.6 | Our Extension | 48 |
| 7 | Related Work | 51 |
| 8 | Conclusions and Future Work | 53 |
| A | Terminology | 56 |

Abstract

Workflow management systems are widely used by organizations to manage and optimize their business processes. Legislation and common business practices mandate authorization constraints, which restrict how people may participate in the execution of workflows. A common and important class of constraints is Separation of Duty, which aims at preventing fraud and errors. Model-based management of workflows is essential for keeping workflow implementations consistent with business requirements and to demonstrate compliance with legislation. Therefore, authorization constraints should be part of workflow models. Recent work introduced the concept of *release* for scoping constraints to prevent them from being overly restrictive. We provide three contributions that build on this approach. First, we extend the industry standard BPMN 2.0, thereby making the release concept available to a large number of workflow management tools. Second, we extend the web-based workflow modeling tool Oryx, proving the applicability of our BPMN extension. Third, for enforcing our constraints we introduce an algorithm which is more general than existing approaches, and prove that it prevents both violations of constraints and obstructions of the workflow.

Acknowledgments

I would like to thank my supervisor Samuel J. Burri for his most helpful guidance and great effort in providing valuable feedback. My thanks also go to the IBM Research Zurich lab for hosting me. I am grateful to my colleagues at the lab, especially Sören Bleikertz, Zoltan Nagy, Animesh Trivedi, Divay Bansal, and Ulrich Dangel for their personal and technical support. I would also like to thank my parents for generously supporting me during my studies.

Dominik Rügger

Zürich, Switzerland

July 2011

Chapter 1

Introduction

1.1 Motivation

According to a survey report by the research firm Gartner [MA11], CIOs of major companies have rated the optimization of processes as a top-five priority during the last four years. For this optimization task, Business Process Management (BPM) systems have been widely deployed. BPM systems are IT systems for representing, managing, and operating an organization’s workflows¹. They are popular in particular to integrate diverse IT systems to a coherent system that is aligned with the organization’s business. Access control for BPM systems is commonly defined in terms of which user can execute which tasks. In particular, organizations would often like to ensure that for a given workflow instance, two tasks must be executed by different users. This is called *Separation of Duties* or the *four-eyes principle* and is an essential method for preventing unintentional errors and fraud. Separation of Duties is central in widely used best practices frameworks such as Control Objectives for Information and Related Technology (COBIT) [Ins05], in particular in sensitive industries such as health and finance. Some companies are required by law to adhere to such practices. For example, publicly traded U.S. companies use COBIT to comply with the Sarbanes-Oxley Act of 2002 [SO002, LP05]. For these organizations, integrating COBIT principles with BPM can facilitate demonstrating their compliance with legislation.

In their recent paper “Obstruction-free Authorization Enforcement: Aligning Security and Business Objectives” [BBK11], David Basin, Samuel Burri, and Günter Karjoth provide a framework for modeling authorization in BPM systems, and for enforcing security constraints in a way that is aligned with the business objectives. They introduce the novel concept of *release* for scoping constraints. Scoping of security constraints can reduce the constraints’ interference with business objectives, while preserving security. However, their method for enforcing constraints does not consider scopes, and is thus more restrictive than necessary. They also describe an informal approach for integrating their framework with the BPMN 2.0 [Obj11] standard for graphical business process notation. The approach is however not tangible enough for practical use. A well designed extension of the BPMN 2.0 standard could make the release concept available to practical applications and thus contribute to BPM systems which minimize the impact of security on business requirements.

1.2 Outline

The basis of this thesis forms the paper “Obstruction-free Authorization Enforcement: Aligning Security and Business Objectives” [BBK11]. We refer to the framework from the paper as the *BBK11 framework*, and to the paper as the *BBK11 paper*.

¹We use *workflow* as a synonym for business process.

This thesis is organized as follows: First, we introduce the standards (in Chapter 2) and models (in Chapter 3) used in the BBK11 framework. In Chapter 4, we introduce our method for modeling security constraints such as Separation of Duties in BPMN 2.0. In Chapter 5, we introduce a new algorithm for enforcing constraints. Finally, in Chapter 6 we describe our extension of a workflow modeling tool.

1.3 Contributions

This thesis makes three contributions:

1. An extension to the industry standard BPMN 2.0 [Obj11] for specifying constraints from the BBK11 framework as part of workflow models. This extension is in accordance with the BPMN 2.0 standard's abstract syntax model and guidelines. Therefore, it makes the concept of release available to the large number of tools which support BPMN 2.0.
2. A software component which extends the modeling tool Oryx [DOW08] to enable modeling of constraints from the BBK11 framework as part of workflow models. This is an implementation and validation of our extension for BPMN 2.0, and also a validation of the BBK11 framework. The component can enable case studies for showing the applicability of the BBK11 framework.
3. A new algorithm for enforcing authorization constraints in a way that is aligned with the business objectives. It takes into account the scopes of constraints, which are defined using the release concept. We show that our algorithm is correct, and that it is more general than the one presented in the BBK11 paper.

1.4 Technology Overview

In this work, we build upon a number of existing standards and software systems. Here, we give a short description of each of them.

- *XML* and *XML Schema*: The Extensible Markup Language (XML) is a widely used markup language for encoding arbitrary semi-structured data. From XML, it is possible to define sub-languages (also known as *XML dialects*) which allow only some data structures. XML Schema is an XML dialect for defining XML dialects. We assume that the reader has basic knowledge of XML and XML Schema. For a detailed treatment of the subject, see [HM04].
- *Business Process Model and Notation (BPMN) 2.0* [Obj11] is an open standard for representing workflows at various levels of detail: from very abstract to detailed enough for automatic execution. It defines a graphical notation as well as a serialization in XML. The syntax of the XML serialization of BPMN 2.0 is specified by an XML Schema. From the BPMN 2.0 standard, we only use *process diagrams*, which describe what we call a *workflow*.
- *CSP*, as described in [Ros05], is a process algebra. It is a notation for describing concurrent processes that communicate via messages. It is useful to formally describe discrete systems for specification and analysis.
- *Oryx* [DOW08] is an open-source web-based modeling tool that supports BPMN 2.0. It has been specifically developed for the BPM research community, by the Hasso-Plattner Institute at the University of Potsdam.

We provide more details about BPMN and CSP in Chapter 2, and about Oryx in Chapter 6. For the definition and analysis of authorization-constrained workflows, we make use of different representations and formalisms. Each of the representations comes with its respective terminology.

As some terms are used by more than one terminology, and with a different meaning, we have to be explicit about which notion we are actually referring to. We try to always indicate the framework from which the terms are taken, when it is not clear from the context. Appendix A contains a list of the terms used within more than one formalism.

Chapter 2

Background

First, we introduce some conventions and notations that are used throughout this document. Then, we present subsets of BPMN 2.0 and CSP. These are formalisms which we use in later chapters.

2.1 Notation

Special fonts indicate special terms and text in this document. We use...

- *italic* font for those terms which have a precisely defined meaning, when we introduce them for the first time, for emphasized words, and for mathematical terms.
- `monospace` font for text that refers to program code, parts of data formats, filenames, and URLs.
- `sans-serif` font for text that refers to elements of models, e.g. classes in abstract syntax models.

In terms of mathematical notation, we use the following special symbols and expressions:

- \emptyset is the empty set.
- \top and \perp are the boolean values true and false.
- 2^S , for a set S , is the power set of S , i.e. the set of all subsets of S .

To refer to a lemma or definition we sometimes use a short form, e.g. *L1* to refer to Lemma 1, and *D1* for Definition 1.

2.2 BPMN

In this work, we extend the widely used Business Process Model and Notation (BPMN) 2.0 standard for graphically specifying authorization constraints in workflows. This means that our extension can easily be integrated with existing tools and methodologies for workflow development and management. We use only a subset of all the elements defined in the BPMN 2.0 standard [Obj11]. In particular, we only consider process diagrams. The following description of BPMN 2.0 only describes the used subset. In this thesis we consider authorization constraints that regulate the execution of tasks and may depend on the workflow's control flow. We do not consider data flows and therefore we ignore the BPMN 2.0 elements dealing with data flows. Furthermore, we consider only the basic elements that regulate control flow, as other elements may alternatively be expressed using the basic elements. A description of the subset, derived from the one given in [BBK11], follows.

2.2.1 Concepts and Modeling Elements

This section provides a short introduction to BPMN, as far as it is relevant for this work. The introduction is based on [BBK11], from where we reuse and extend the graphics and some of the descriptive text.

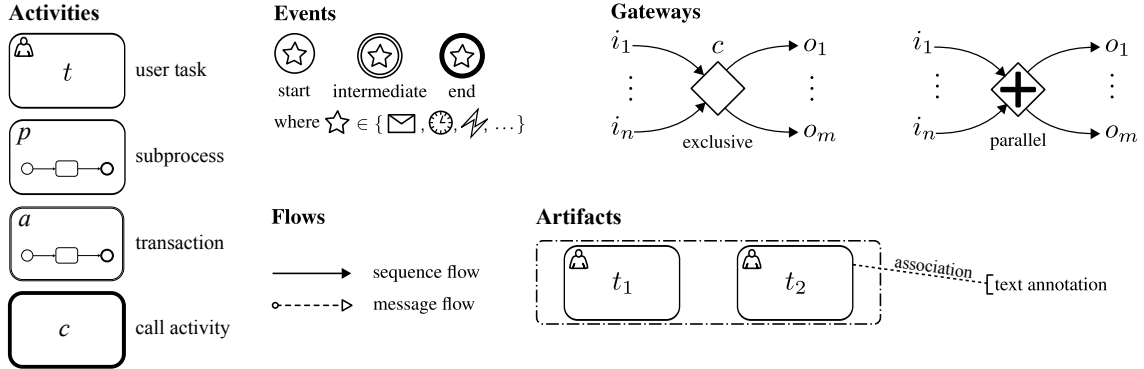


Figure 2.1: BPMN elements (graphics adapted from [BBK11]).

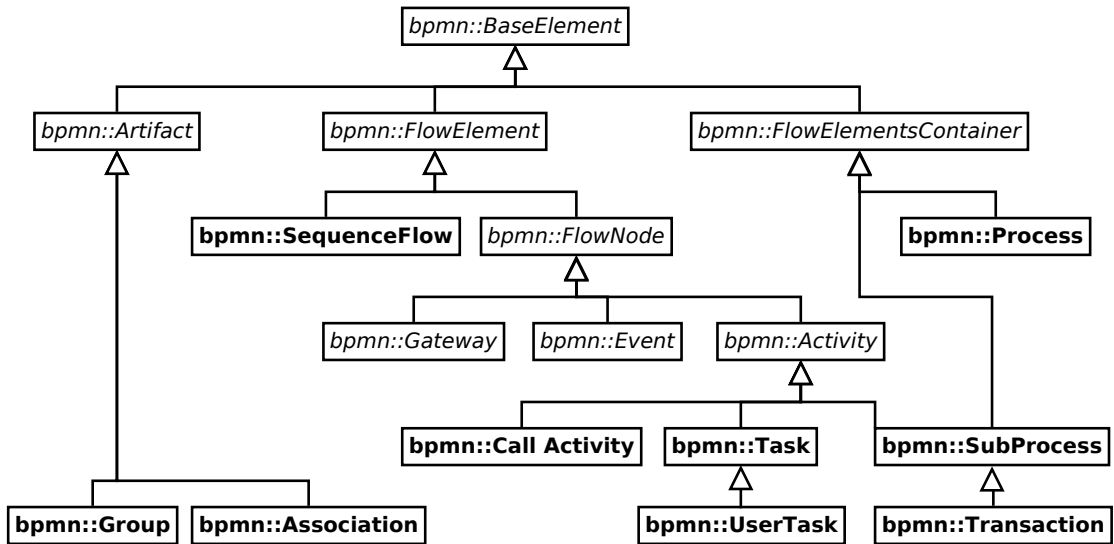


Figure 2.2: Abstract Syntax of BPMN 2.0 (excerpt). Class names in italic font denote abstract classes, class names in bold font denote concrete classes.

The BPMN 2.0 standard describes BPMN using tables of graphical symbols and an abstract syntax model. We use the same approach here. Figure 2.1 shows the graphical symbols, and Figure 2.2 shows an UML¹ class diagram of parts of the abstract syntax classes, with their inheritance relations. In the following, when describing a particular element of BPMN, we mention the name of the corresponding abstract syntax class in sans-serif font (e.g. “Event”).

Figure 2.1 illustrates the five kinds of BPMN elements that we consider. *Activities* (Activity) encompass *tasks* (Task), *subprocesses* (SubProcess), and *call activities* (CallActivity). All activities are represented by rectangles with rounded corners. A task’s type may be indicated by an icon in the upper left corner. A *user task* is denoted by an icon depicting a person. A subprocess

¹We assume that the reader has basic knowledge of the *Unified Modeling Language* (UML). We used [BRJ05] as our reference.

is a process that is embedded in another process, called the *parent process*. A call activity calls another, globally defined activity.

“An *event* (**Event**) models the occurrence of a condition or 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.” [BBK11]

End events that contain a filled circle are *terminate events*. Terminate events immediately trigger the process’ termination, so that any concurrently executed activities and control flow branches are aborted.

“*Flows* describe a workflow’s control-flow. A *sequence flow* (**SequenceFlow**), illustrated by a solid line with an arrow, defines the order in which tasks are executed and events occur. BPMN has other flow elements, such as message flows, but we make only use of sequence flows. Merging and branching of the control-flow is modeled by *gateways* (**Gateway**). A gateway has $n \geq 1$ incoming and $m \geq 1$ outgoing sequence flows. *Exclusive gateways* are depicted by an empty (or with an “x” labeled) diamond. Whenever the control-flow reaches an exclusive gateway on an incoming sequence flow, it passes the control-flow immediately on to exactly one of the m outgoing sequence flows, based on the evaluation of the condition c associated with the gateway. *Parallel gateways* are illustrated by a diamond labeled with the symbol “+”. They synchronize the control flow on the n incoming sequence flows and spawn concurrent execution on the m outgoing sequence flows.” [BBK11]

Artifacts (**Artifact**) are offered by BPMN for annotating models. They cannot participate in sequence or message flows. For example, *text annotations* can be added to any element, as illustrated in Figure 2.1. We use two types of artifacts: The dotted line connecting the text annotation to the task is an *association* (**Association**), and sets of tasks are defined by grouping them with a dot-dashed box (**Group**).

2.2.2 Examples

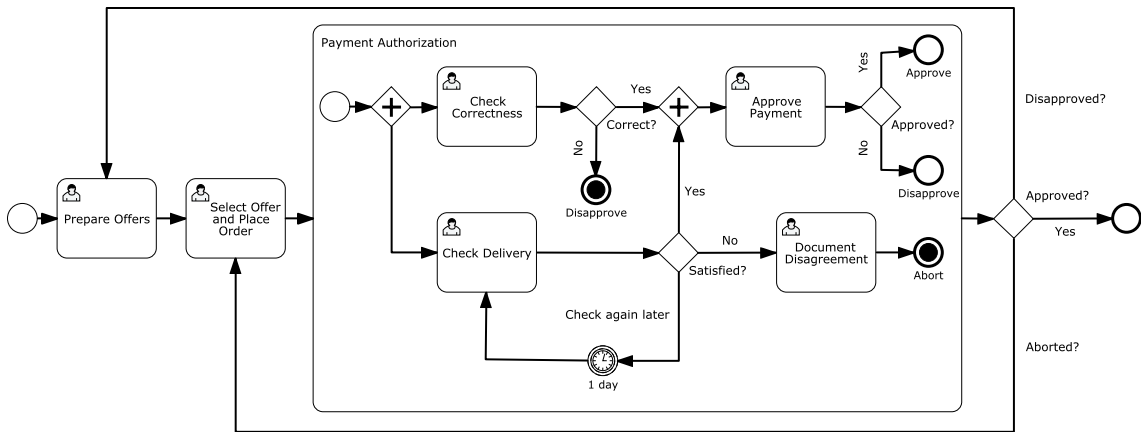


Figure 2.3: Example Workflow 1: Procurement

Example 2.1 (Procurement Workflow). As an example, we describe the process shown in Figure 2.3. It is an extended and simplified version of an actual payment authorization process from industry [Exp09]. To reduce complexity, we omitted some of the tasks. To make the example more interesting, we embedded the payment authorization into the context of procurement. The scenario that it handles is that a company needs to buy an item, let’s say a laptop computer. A

user collects the offers from different suppliers and then a user selects an offer and places an order. After that, the workflow starts a sub-process, where the (financial) correctness of the order, and the appropriate delivery of the goods are checked in parallel. If the result of any of those checks turns out to be negative, the subprocess is terminated immediately. If both checks complete successfully, a user decides on the approval of the payment. Upon termination of the sub-process, if its result is “approve”, the whole workflow terminates, having fulfilled its purpose. If the payment authorization is aborted, then another offer is selected and a new order placed. If the payment is disapproved, then the list of possible offers is revised (possibly extended), the selection task is repeated, and the payment authorization sub-process is executed anew.

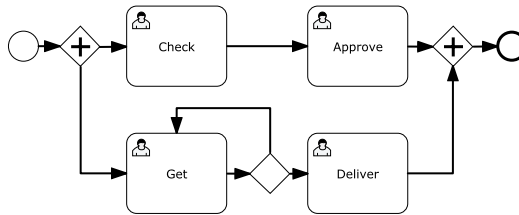


Figure 2.4: Example Workflow 2: A simple process

Example 2.2 (A simple process). To illustrate the semantics of a workflow, we describe sequences of task executions that are possible in the simple workflow given in Figure 2.4, by listing some examples. Possible sequences are the following:

- *Check, Approve, Get, Deliver.*
- *Get, Get, Check, Deliver, Approve.*
- *Get, Check, Get, Approve, Get, Get, Deliver.*

Impossible sequences:

- *Check, Approve, Approve, Get, Deliver.* (Because *Approve* can only be executed once.)
- *Get, Deliver, Get, Check, Approve.* (Because *Get* is not possible after *Deliver*.)

2.2.3 XML Serialization

The BPMN 2.0 standard defines an XML serialization for BPMN models, formally described by an XML Schema that closely follows the abstract syntax model. An XML serialization of a BPMN model consists of a sequence of *definitions*. There are two kinds of definitions, *process definitions* and *diagram definitions*. The process definition describes the elements of the process diagram and their relations. The diagram definition describes how these elements are to be presented in the diagram, for example by giving their exact location as (x, y) coordinates. The diagram definition only complements the process definition for presentation as far as necessary. It does not contain any information which can be derived from the process definition. Therefore, for rendering the diagram, both the process and the diagram definitions are required. For semantic analysis and execution of the process, only the process definition is relevant. In a particular serialization of a BPMN model, diagram definitions are optional or may only describe a partial view of a process definition. In general, it is forbidden for an element of the process definition to be associated with more than one element in the diagram definition, i.e. it can only be rendered once in the diagram.

For the semantic data and the presentational data, the BPMN 2.0 standard each gives an XML Schema file: `Semantic.xsd` and `BPMNDI.xsd`, respectively. The XML Schema file `BPMN20.xsd`

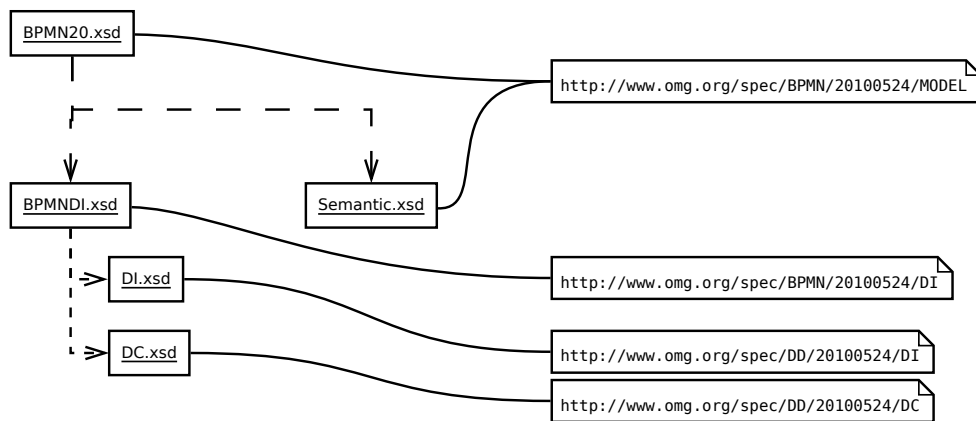


Figure 2.5: Schema Files and Associated Namespaces

combines the semantic and presentational schemata by defining a topmost XML element called **definitions** that contains elements from **Semantic.xsd** and **BPMN20100524/MODEL**.

The letters “DI” in “**BPMN20100524/DI**” refer to *Diagram Interchange (DI)*, which is part of the *Diagram Definition (DD)* standard that is currently developed by the Object Modeling Group (OMG)² to describe various kinds of graphical elements, such as shapes, edges, and labels. So far, it is only used within the BPMN 2.0 standard, but is designed to be very generic, so that it can be used for serializing other modeling languages by OMG. **BPMN20100524/DI** uses type definitions from **DI.xsd** (the XML Schema file for Diagram Interchange) and **DC.xsd** (the XML Schema file for *Diagram Common (DC)*, which is another part of DD that describes its most basic elements, and is used by DI). **DC.xsd** and **DI.xsd** are files taken from a preliminary version of DD and added to the set of the BPMN 2.0 standard’s XML Schema files. **BPMN20100524/DI** extends the data structures defined in DI and DC with additional attributes specific to BPMN.

Every XML Schema file is associated with a *target namespace*. The target namespace is a prefix added to all the names of types and other elements defined in the schema file. Syntactically, it is a URI. Its purpose is to give everything that is defined in the schema a globally unique name. This facilitates the use of tags in XML documents that are defined in different XML Schema files, avoiding name clashes.

The BPMN 2.0 XML serialization uses several namespaces. Elements and types are assigned to a namespace depending on the context that they are used in. What is specific to the BPMN 2.0 context is assigned to the BPMN 2.0 Model namespace:

`http://www.omg.org/spec/BPMN/20100524/MODEL`

This includes everything that is defined in **Semantic.xsd**.

On the other hand, all types and elements defined in **BPMN20100524/DI** are part of the BPMN 2.0 Diagram Interchange namespace:

`http://www.omg.org/spec/BPMN/20100524/DI`

Figure 2.5 shows the XML Schema files of the BPMN 2.0 standard with their dependencies and associated namespaces. A dashed arrow from file A to file B means that A uses objects defined in B. **DI.xsd** and **DC.xsd** also have their own namespaces. They make no reference to BPMN, instead they are part of the “DD” (Diagram Definition) specification.

Listing 2.1 shows an abbreviated example of a BPMN process serialized in XML. The **definitions** tag contains the tags **process** and **BPMNDiagram**, which contain the semantic and presentational data, respectively. Every element in the presentational data references an element from the semantic data using the **bpmnElement** attribute.

²As of June 2011, the Beta 1 version [Obj10] was most recent.

Listing 2.1: Example

```

<definitions
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
  xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
  xmlns:dc="http://www.omg.org/spec/DD/20100524/DC">
  <process processType="None" id="process1">
    <startEvent id="start1"/>
    <task name="Task1" id="task1"/>
    <task name="Task2" id="task2"/>
    <endEvent id="end1"/>
    <sequenceFlow sourceRef="start1"
      targetRef="release1" id="flow1"/>
    [...]
  </process>
  <bpmndi:BPMNDiagram>
    <bpmndi:BPMNPlane id="proce1pl" bpmnElement="process1">
      <bpmndi:BPMNShape id="start1sh" bpmnElement="start1">
        <dc:Bounds height="-1" width="-1" x="80" y="200"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="task1sh" bpmnElement="task1">
        <dc:Bounds height="-1" width="-1" x="300" y="200"/>
        <bpmndi:BPMNLabel id="task1lb"/>
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="flow1ed" bpmnElement="flow1">
        <di:waypoint x="80" y="200"/>
        <di:waypoint x="300" y="200"/>
      </bpmndi:BPMNEdge>
      [...]
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>

```

2.2.4 Extension Mechanism

The XML schema of BPMN 2.0 is, according to the standard [Obj11], designed to be extensible. In the semantic schema, any XML element can be extended by adding a sub-element `extensionElements`, which is allowed to contain arbitrary elements from a different namespace. In the presentational schema, most of the elements (except for the very basic types such as waypoints and shape bounds) can be extended by adding a sub-element `extension`, which is allowed to contain arbitrary XML elements.

Presentational data that is not covered in the standard can therefore be added freely, even without providing an additional schema. To include extensional semantic data in a BPMN XML document, it is enough to reference an additional schema document that defines elements of a namespace different from that of the BPMN elements. It is allowed for the extension schema to build upon elements defined in the BPMN standard's existing XML Schema.

2.3 CSP

For modeling authorization constraints and their enforcement, we use a subset of Hoare's process algebra CSP [Ros05], which we introduce in this section. The chosen subset and explanation are based on what is contained in [BBK11]. Before introducing CSP, however, we provide some relevant definitions and laws about relations, which we use extensively together with CSP.

2.3.1 Relations: Definitions and Properties

Definition 2.1. Let ψ be a relation $\psi \subseteq A \times B$ for the sets A and B . We use the following definitions:

- $a \psi b :\Leftrightarrow (a, b) \in \psi$
- $\psi^{-1} := \{(b, a) \mid (a, b) \in \psi\}$ (inverse relation)
- $\psi(A') := \{b \in B \mid \exists a \in A' : (a, b) \in \psi\}$, for a set $A' \subseteq A$: using ψ as a function $2^A \rightarrow 2^B$.
- $\psi^{-1}(B') := \{a \in A \mid \exists b \in B' : (a, b) \in \psi\}$, for a set $B' \subseteq B$
- $\psi\{a\} := \{b \in B \mid (a, b) \in \psi\}$
- $dom(\psi) := \{a \mid \exists b : (a, b) \in \psi\}$ (domain)
- $ran(\psi) := \{b \mid \exists a : (a, b) \in \psi\}$ (range)

Note that ψ^{-1} is not generally the inverse of ψ when used as functions on sets. However the following properties hold: For a relation $\psi \subseteq A \times B$, with $A', A'' \subseteq A$:

- $\psi(A' \cup A'') = \psi(A') \cup \psi(A'')$.
- $A' \subseteq A'' \Rightarrow \psi(A') \subseteq \psi(A'')$.

2.3.2 Notation and Definitions

Note 2.1 (Source). *In this subsection (2.3.2) we use text from [BBK11]. In an effort to stay compatible with their definitions, we re-use their description of CSP. We reproduce most of their description verbatim, with some changes. We left out parts that are not relevant to this work, and added extra information where needed.*

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

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

For an event $\sigma \in \Sigma$ and a name $n \in \mathcal{N}$, the set of processes \mathcal{P} is inductively defined by the following grammar:

$$\mathcal{P} ::= \sigma \rightarrow \mathcal{P} \mid \text{SKIP} \mid \text{STOP} \mid n \mid \mathcal{P} \square \mathcal{P} \mid \mathcal{P} \sqcap \mathcal{P} \mid \mathcal{P} \parallel \mathcal{P} \mid \mathcal{P} \parallel \parallel \mathcal{P} \mid \mathcal{P}; \mathcal{P} \quad (2.1)$$

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*. Note that because $\mathsf{T}(P)$ is prefix-closed, the empty trace $\langle \rangle$ is an element of the set of traces of any process. The operational semantics describes P as a state-transition system. 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.

Let $P_1, P_2 \in \mathcal{P}$ be two processes. The process $\sigma \rightarrow P_1$ engages in the event σ first and behaves like P_1 afterward. Formally, $\mathsf{T}(\sigma \rightarrow P_1) = \{\langle \sigma \rangle \hat{\ } i \mid i \in \mathsf{T}(P_1)\} \cup \{\langle \rangle\}$. *SKIP* engages in \checkmark and no further event afterward; $\mathsf{T}(\text{SKIP}) = \{\langle \rangle, \langle \checkmark \rangle\}$. *STOP* represents the process that does not engage in any event; $\mathsf{T}(\text{STOP}) = \{\langle \rangle\}$. In other words, *SKIP* represents successful termination and *STOP* a deadlock. We write $n = P_1$ to assign P_1 to the name n . Correspondingly, the process n behaves like P_1 . The process $P_1 \square P_2$ 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, $P_1 \square P_2$ and $P_1 \sqcap P_2$ are indistinguishable, namely the following holds:

$$\mathsf{T}(P_1 \square P_2) = \mathsf{T}(P_1 \sqcap P_2) = \mathsf{T}(P_1) \cup \mathsf{T}(P_2) \quad (2.2)$$

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 :

$$\mathsf{T}(P_1 \parallel P_2) = \mathsf{T}(P_1) \cap \mathsf{T}(P_2) \quad (2.3)$$

Similarly, the process $P_1 \parallel \parallel P_2$ 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 ; $\mathsf{T}(P_1 \parallel \parallel P_2)$ is the set of all interleavings of i_1 and i_2 for $i_1 \in \mathsf{T}(P_1)$ and $i_2 \in \mathsf{T}(P_2)$. For a trace $i \in \mathsf{T}(P_1)$, $P_1 \setminus i$ represents the process P_1 after engaging in all events in i . If $\mathsf{T}(P_1) \subseteq \mathsf{T}(P_2)$, then P_1 is a *trace refinement* of P_2 , denoted $P_2 \sqsubseteq_{\mathsf{T}} P_1$. If $P_2 \sqsubseteq_{\mathsf{T}} P_1$ and $P_1 \sqsubseteq_{\mathsf{T}} P_2$, 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 trace 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 . For each refusal set R of P , all of R 's subsets are also refusal sets of P . The set of *failures* of P is defined as:

$$\mathsf{F}(P) = \{(i, R) \mid i \in \mathsf{T}(P), R \in rs(P \setminus i)\} \quad (2.4)$$

For two processes P_1 and P_2 , P_2 *failure refines* P_1 , written $P_1 \sqsubseteq_{\mathsf{F}} P_2$, if $\mathsf{F}(P_2) \subseteq \mathsf{F}(P_1)$. 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 $P_2 \sqsubseteq_{\mathsf{F}} P_1$.

We define the set of events that a process P offers: $initials : \mathcal{P} \rightarrow 2^{\Sigma^\vee}$.

$$initials(P) = \{e \in \Sigma^\vee \mid \langle e \rangle \in \mathsf{T}(P)\} \quad (2.5)$$

P can immediately engage in any of those events, but it may also refuse some of those events that are in a refusal set of P .

For a relation $\psi \subseteq \Sigma \times \Sigma \cup \{(\checkmark, \checkmark)\}$, with $(\checkmark, \checkmark) \in \psi$, and a process P , $P[\psi]$ denotes P renamed by ψ . For every pair $(\sigma_1, \sigma_2) \in \psi$, $P[\psi]$ engages in σ_2 if P engages in σ_1 . We abuse ψ as a relation between traces, where for traces $a = \langle a_1, \dots, a_n \rangle$ and $b = \langle b_1, \dots, b_n \rangle$ holds $a \psi b$ if and only if $n = m$ and $\forall i \in \{1, \dots, n\} : (a_i, b_i) \in \psi$. Note that $a \psi b$ implies $a \in \text{dom}(\psi)^*$ and $b \in \text{ran}(\psi)^*$ and $b \psi^{-1} a$. Remember that we also use ψ as a function between sets of events: For a set $S \subseteq \Sigma^\vee$ holds $\psi(S) = \{e \mid \exists e' \in S : (e', e) \in \psi\}$, as in Definition 2.1.

For the initials of a renamed process holds:

$$initials(P[\psi]) = \psi(initials(P)) \quad (2.6)$$

For the traces of a renamed process holds:

$$\mathsf{T}(P[\psi]) = \{i' \mid \exists i : i \psi i' \wedge i \in \mathsf{T}(P)\} \quad (2.7)$$

Some laws on failures, which hold for any processes P and Q :

$$\mathsf{F}(P \parallel Q) = \{(i, Y \cup Z) \mid (i, Y) \in \mathsf{F}(P) \wedge (i, Z) \in \mathsf{F}(Q)\} \quad (2.8)$$

$$\mathsf{F}(P[\psi]) = \{(i', R') \mid \exists i : i \psi i' \wedge (i, \psi^{-1}(R')) \in \mathsf{F}(P)\} \quad (2.9)$$

$$\forall (i, R) \in \mathsf{F}(P) : \forall R' \subseteq R : (i, R') \in \mathsf{F}(P) \text{ (by the definition of } rs(P)) \quad (2.10)$$

A *deterministic* process D has just one maximal refusal set for a trace i , which is the complement of $initials(D \setminus i)$, i.e. D cannot refuse any of the events that it offers. Therefore:

$$\mathsf{F}(D) = \{(i, R) \mid i \in \mathsf{T}(D), R \subseteq \Sigma^\vee \setminus initials(D \setminus i)\} \quad (2.11)$$

2.3.3 Style

The CSP processes that we use in this document are defined following a common style, which we explain in this section. We also introduce a terminology which we use to reason about some properties of the processes.

Our preferred way to specify processes is by a parameterized recursive formula that makes states and transitions of the process explicit:

Example 2.3. The definition

$$\begin{aligned} P_{2.3}(U) &= v : \{v \in \mathbb{N}\} \rightarrow P_{2.3}(U \cup \{v\}) \\ &\quad \square w : \{w \in \mathbb{Z} \mid -w \in U\} \rightarrow P_{2.3}(U \setminus \{-w\}) \\ &\quad \square SKIP, \end{aligned}$$

$$U \in 2^{\mathbb{N}}$$

specifies an infinite set of processes $\{P_{2.3}(U) \mid U \in 2^{\mathbb{N}}\}$. These processes form a system where generally each process, after engaging in an event behaves like another process from the system. As defined, $P_{2.3}(\emptyset)$ behaves like $P_{2.3}(\{1\})$ after engaging in the event 1, and $P_1(\{1\})$ behaves again like $P_{2.3}(\emptyset)$ after engaging in -1 . The only special case is that the process may behave like *SKIP* at any time, i.e. engaging in \checkmark and then terminating.

Definition 2.2. A process definition P is called a *regular process definition* if it matches the following pattern:

$$\begin{aligned}
P(p) &= e : E_1(p) \rightarrow P(\text{next}_1(p, e)) \\
&\quad \square e : E_2(p) \rightarrow P(\text{next}_2(p, e)) \\
&\quad \dots \\
&\quad \square e : E_n(p) \rightarrow P(\text{next}_n(p, e)) \\
&\quad \square \text{SKIP} , \\
p &\in \varphi .
\end{aligned}$$

The placeholder p stands for a tuple of parameters. φ is the set of all parameters. Each line $e : E_i(p) \rightarrow P(\text{next}_i(p, e))$ is a *rule*, and n is the number of rules. The set $E_i(p)$ is called the rule's *guard*. The guard is a subset of the set of events Σ , and can depend on the parameters. The term after the arrow is called the *subsequent process term*. Each next_i is a function of the parameters p and the event e . The set $G(p) = \{E_1(p), E_2(p), \dots, E_n(p)\}$ is the set of guards of P . The union of all guards, together with \checkmark , is the set of all initials:

$$\forall p \in \varphi : \bigcup_{i=1..n} E_i(p) \cup \{\checkmark\} = \text{initials}(P(p))$$

Definition 2.3. A regular process definition is well-defined if for every rule $e : E_i(p) \rightarrow P(\text{term}_i(p, e))$ holds $\forall e \in E_i(p) : \text{term}_i(p, e) \in \varphi$, and $\text{term}_i(p, e)$ is well-defined for all possible p and e .

Example 2.4. The definition

$$\begin{aligned}
P_{2.4}(n) &= m : \mathbb{N} \rightarrow P_{2.4}(n - m) \\
&\quad \square \text{SKIP} , \\
n &\in \mathbb{N}
\end{aligned}$$

is not well-defined as $P_{2.4}(1)$, after engaging in the event 1, would be $P_{2.4}(0)$, which is not defined, as the parameter n is constrained by $n \in \mathbb{N}$.

Definition 2.4. We say that the process $P(p_1) \setminus i$ for some $i \in \mathbb{T}(P(p_1))$ is in the state p_2 , if $P(p_1) \setminus i = P(p_2)$, i.e. it behaves like $P(p_2)$.

If all guards are pairwise disjoint for all parameters, i.e. $\forall p \in \varphi \forall E_i(p), E_j(p) \in G : i \neq j \Rightarrow E_i(p) \cap E_j(p) = \emptyset$, then the defined process is deterministic. If the process is deterministic and a rule $e : E_i(p) \rightarrow P(\text{term}_i(p, e))$ exists in the definition, then for each event $e \in E_i(p)$ holds $P(p) \setminus \langle e \rangle = P(\text{term}_i(p, e))$.

Note 2.2. CSP according to [Ros05] defines the process $E = P \square \text{SKIP}$, for some process P , as such that it may nondeterministically decide to refuse all events that P offers and only allow \checkmark , i.e. $E = (P \square \text{STOP}) \square \text{SKIP}$. In contrast, we assume that E may not refuse any events that P offers. We strictly leave it to the environment to decide on termination, and therefore E is a deterministic process. In the interest of simplicity we feel that it is appropriate to deviate from [Ros05] in this way. Alternatively, one could introduce an additional event, e.g. $\#$, which marks the termination of a workflow, after which \checkmark will be the only event that can be executed. In this case one would have to replace all instances of SKIP in our definitions with the process $\# \rightarrow \text{SKIP}$. Also one would have to add $\#$ to all sets of events which contain \checkmark .

In general, for the process definitions in this thesis we only state that they are deterministic, without giving a proof, except if it is not straightforward to verify.

Definition 2.5. A rule $e : E_i(p) \rightarrow P(\text{term}_i(p, e))$ is called *repeatable* if $E_i(p) = E_i(\text{term}_i(p, e))$.

If a rule $e : E_i(p) \rightarrow P(\text{term}_i(p, e))$ is repeatable, then after engaging in any event from $E_i(p)$, the guard of the rule is the same as before, and therefore the process can again engage in all the events from $E_i(p)$, i.e.

$$e \in E_i(p) \Rightarrow e \in \text{initials}(P(p) \setminus \langle e \rangle)$$

Chapter 3

Authorization-Constrained Workflows

In this chapter, we first describe the general framework of authorization-constrained workflows used in [BBK11], and then introduce their concrete model.

3.1 Framework

3.1.1 Workflow and Execution Model

There is a global set of all *tasks* \mathcal{T} and of all *points* \mathcal{O} . There is also a global non-empty set of all *users* \mathcal{U} . The set of *task-execution events* is $\mathcal{T} \times \mathcal{U}$. We write $t.u$ for an element $(t, u) \in \mathcal{T} \times \mathcal{U}$. A task-execution event $t.u$ represents the execution of the task t by the user u . *Task events* are elements of \mathcal{T} , and denote the execution of a task independent of the user which executes it. *Point events* are elements from \mathcal{O} , and include all other events that one might want to consider. The sets \mathcal{T} , $\mathcal{T} \times \mathcal{U}$ and \mathcal{O} must be strictly disjoint. The set of all regular events does not include any additional events, so $\Sigma = \mathcal{T} \cup (\mathcal{T} \times \mathcal{U}) \cup \mathcal{O}$. The set of *execution events* $\Sigma_{\mathcal{X}}$ is $(\mathcal{T} \times \mathcal{U}) \cup \mathcal{O}$. The set of all *execution traces* is $\Sigma_{\mathcal{X}}^{\checkmark}$. The set of all *unterminated execution traces* is $\Sigma_{\mathcal{X}}^*$.

Definition 3.1. An *execution process* is a CSP process that engages only in execution events and \checkmark .

We define a *workflow process*, which is the CSP model of a workflow. The set of *workflow events* $\Sigma_{\mathcal{T}}$ is $\mathcal{T} \cup \mathcal{O}$. The set of all *workflow traces* is $\Sigma_{\mathcal{T}}^{\checkmark}$.

Definition 3.2. A *workflow process* is a CSP process that engages only in workflow events and \checkmark . Let \mathcal{W} denote the set of all workflow processes.

The execution process of a workflow process W is the process $W[\pi^{-1}]$, where π is the relation that maps the task-execution event $t.u$ to the task event t , and \checkmark and events in \mathcal{O} to themselves:

$$\begin{aligned} \pi &:= \{(t.u, t) \mid t.u \in (\mathcal{T} \times \mathcal{U})\} \cup \{(o, o) \mid o \in \mathcal{O}\} \cup \{(\checkmark, \checkmark)\} \\ \text{dom}(\pi) &= \Sigma_{\mathcal{X}}^{\checkmark}, \text{ran}(\pi) = \Sigma_{\mathcal{T}}^{\checkmark} \end{aligned}$$

An execution trace of $W[\pi^{-1}]$ models a workflow instance of the workflow modeled by W .

Note that π is also a function (while π^{-1} is not). Because π is a total function on $\Sigma_{\mathcal{X}}^{\checkmark}$, for an execution trace $i \in \Sigma_{\mathcal{X}}^{\checkmark}$ we can define the unique workflow trace $i[\pi] \in \Sigma_{\mathcal{T}}^{\checkmark}$ of the same length, which for $i = \langle e_1, \dots, e_n \rangle$ is defined as $i[\pi] := \langle \pi(e_1), \dots, \pi(e_n) \rangle$. So for any two traces $i, i' \in \Sigma_{\mathcal{X}}^{\checkmark}$ holds:

$$i \pi i' \Leftrightarrow (i \in \Sigma_{\mathcal{X}}^{\checkmark} \wedge i' = i[\pi]) \tag{3.1}$$

Because π is surjective on $\Sigma_{\mathcal{T}}^{\checkmark}$ and $\mathcal{U} \neq \emptyset$, for each workflow trace $i' \in \Sigma_{\mathcal{T}}^{*\checkmark}$ there exists at least one execution trace $i \in \Sigma_{\mathcal{X}}^{*\checkmark}$ such that $i' = i[\pi]$. By the definition of π , the following two properties hold for a set of events $S \subseteq \Sigma^{\checkmark}$:

$$\forall t \in \mathcal{T} : t \in \pi(S) \Rightarrow \exists u \in \mathcal{U} : t.u \in S \quad (3.2)$$

$$\forall e \in \mathcal{O}^{\checkmark} : e \in \pi(S) \Rightarrow e \in S \quad (3.3)$$

The following is a limitation of this model: The execution of tasks is considered *atomic*, i.e. task executions are modeled as events without duration. Though sequences of task executions can be executed in parallel, two task executions cannot be happening at the same time. By the CSP model, one event always happens either before or after other event. This collides with the observation that in practice tasks can be executed in parallel *and* take a certain amount of time to complete. The consequence is that when applying this model for practical systems, an exact point in time has to be defined after which a task is considered executed.

3.1.2 Constraint Model

A workflow is *authorization-constrained*, if there exist constraints that describe which user may execute which tasks in the workflow.

Definition 3.3. An *authorization process* is an execution process which is meant to be used as a reference monitor for preventing the violation of an underlying constraint. The traces of an authorization process are those that *satisfy* the underlying constraint.

We distinguish between *static* and *dynamic* constraints. *Static* constraints can be enforced by an authorization process which offers always the same events, independent of the events that have already been executed. This implies that without executing the workflow, it can be determined whether a certain user executing a certain task would violate the constraint. All other constraints are called *dynamic*. Dynamic constraints depend on events that happen during workflow execution.

Example 3.1. A common model for controlling access to resources is *Role-Based Access Control* (RBAC) [FSG⁺01]: In our context, a simple variant of RBAC could be implemented as follows: Each user gets assigned a set of roles. Each role gets assigned a set of tasks. A user can execute a task if it is assigned to a role that he is assigned to.

In this work, we use a more abstract model, which is simply a relation between users and tasks that specifies which tasks each user can execute. It can be defined by an RBAC system or any other mechanism. In practice, the relation may change over time. However, to simplify our model we assume that it is fixed, and therefore it constitutes a *static* constraint.

Example 3.2. An important kind of dynamic constraint is *Separation of Duties*, which we abbreviate as *SoD*. An SoD constraint for two tasks states that in a workflow instance, they cannot be executed by the same user. It is a common requirement in heavily regulated industries such as finance and healthcare. *Binding of Duties*, or *BoD*, is dual to SoD. A BoD constraint for two tasks states that within a workflow instance, those two tasks *must* be executed by the same user.

SoD and BoD as given by those definitions are dynamic constraints, because after one of two tasks included in an SoD or BoD constraint has been executed, the set of users which are authorized to execute the second task depends on which user has executed the previous task.

3.1.3 Functions

We define some functions that we use on authorization processes:

Definition 3.4. $\chi : \mathcal{P} \rightarrow 2^{\mathcal{T} \times \mathcal{U}}$, $\chi(P) = (\mathcal{T} \times \mathcal{U}) \cap \text{initials}(P)$ is the set of task-execution events that the process P offers.

Definition 3.5. $\omega : \mathcal{P} \rightarrow 2^{\mathcal{O}}$, $\omega(P) = \mathcal{O} \cap \text{initials}(P)$ is the set of point events that the process P offers.

3.2 The [BBK11] Constraint Model

3.2.1 Model

We define what constraints are possible in the model, and, in the next section, what their semantics are. See [BBK11] for a more detailed treatment.

Definition 3.6 (SoD Constraint). An *SoD constraint* s is a triple $s = (T_1, T_2, O)$, where T_1 and T_2 are non-empty sets of tasks with $T_1 \cap T_2 = \emptyset$, and O is a set of points, called the *release points* of s .

Definition 3.7 (BoD Constraint). A *BoD constraint* b is a pair $b = (T, O)$, where T is a non-empty set of tasks, and O is a set of points, called the *release points* of b .

Definition 3.8 (User-Task Assignment). A *user-task assignment* UT , is a relation $UT \subseteq \mathcal{U} \times \mathcal{T}$.

Definition 3.9 (Authorization Policy). An *authorization policy* is a triple (UT, S, B) , where UT is a user-task assignment, S is a set of SoD constraints, and B is a set of BoD constraints.

Note 3.1. *Alternatively, we could define the policy using a relation $\mathcal{T} \times \mathcal{U}$ instead of UT , which would be more convenient for some definitions. However, we chose to stay compatible with the definitions in [BBK11].*

3.2.2 Properties

We define some properties which are not part of the approach described in [BBK11]. We use them for defining our new algorithm.

For a given authorization policy $\phi = (UT, S, B)$, we define the following relations between tasks:

Definition 3.10. Two tasks t_1, t_2 are called ...

- *s-separated*, denoted $t_1 \neq_s t_2$, if for an SoD constraint $s = (T_1, T_2, O)$ holds that $t_1 \in T_1 \wedge t_2 \in T_2$ or $t_1 \in T_2 \wedge t_2 \in T_1$.
- *ϕ -separated*, denoted $t_1 \neq_\phi t_2$, if there exists an SoD constraint $s \in S$, $s = (T_1, T_2, O)$ such that $t_1 \neq_s t_2$.
- *b-bound*, denoted $t_1 =_b t_2$, if for the BoD constraint $b = (T, O)$ holds that $t_1 \in T$ and $t_2 \in T$.
- *ϕ -bound*, denoted $t_1 =_\phi t_2$, if there exists a sequence of tasks t'_1, \dots, t'_n and a sequence of BoD constraints b_1, \dots, b_{n+1} from B , such that $t_1 =_{b_1} t'_1$, $t'_k =_{b_{k+1}} t'_{k+1}$ for $1 \leq k \leq n$, and $t'_n =_{b_{n+1}} t_2$.

As suggested by the used notation symbols, these relations have certain properties:

Lemma 3.1. *For any authorization policy ϕ , the relation \neq_ϕ on \mathcal{T} is symmetric, and the relation $=_\phi$ on \mathcal{T} symmetric and transitive.*

Proof. Let $\phi = (UT, S, B)$ be an authorization policy.

- Let $t_1, t_2 \in \mathcal{T}$, and $t_1 \neq_\phi t_2$. Therefore, an SoD constraint $s = (T_1, T_2, O)$ exists such that $t_1 \in T_1 \wedge t_2 \in T_2 \vee t_1 \in T_2 \wedge t_2 \in T_1$. This implies $t_2 \in T_1 \wedge t_1 \in T_2 \vee t_2 \in T_2 \wedge t_1 \in T_1$. Therefore also $t_2 \neq_\phi t_1$ holds, which proves that \neq_ϕ is symmetric.
- Let $t_1, t_2 \in \mathcal{T}$, and $t_1 =_\phi t_2$. Therefore there exists a sequence of tasks t'_1, \dots, t'_n and a sequence of BoD constraints b_1, \dots, b_{n+1} from B , such that $t_1 =_{b_1} t'_1$, $t'_k =_{b_{k+1}} t'_{k+1}$ for $1 \leq k \leq n$, and $t'_n =_{b_{n+1}} t_2$. Since for any $b \in B$, $t_1, t_2 \in \mathcal{T}$, holds $t_1 =_b t_2 \Rightarrow t_2 =_b t_1$, it follows with the reverse sequences of b_{n+1}, \dots, b_1 and t'_n, \dots, t'_1 that $t_2 =_\phi t_1$ holds. Therefore $=_\phi$ is symmetric.

- Let $t_1, t_2, t_3 \in \mathcal{T}$, and $t_1 =_\phi t_2$ and $t_2 =_\phi t_3$. Therefore there exists a sequence of tasks t'_1, \dots, t'_n and a sequence of BoD constraints b_1, \dots, b_{n+1} from B , such that $t_1 =_{b_1} t'_1$, $t'_k =_{b_{k+1}} t'_{k+1}$ for $1 \leq k \leq n$, and $t'_n =_{b_{n+1}} t_2$. Also there exists a sequence of tasks $t''_1, \dots, t''_{n'}$ and a sequence of BoD constraints $b'_1, \dots, b'_{n'+1}$ from B , such that $t_2 =_{b'_1} t''_1$, $t''_k =_{b'_{k+1}} t''_{k+1}$ for $1 \leq k \leq n'$, and $t''_{n'} =_{b'_{n'+1}} t_3$. With the sequence of tasks $t'_1, \dots, t'_n, t_2, t'_1, \dots, t'_n, t''_1, \dots, t''_{n'}$ and the sequence of BoD constraints $b_1, \dots, b_{n+1}, b'_1, \dots, b'_{n'+1}$ follows that $t_1 =_\phi t_3$, which proves that $=_\phi$ is transitive. □

Note 3.2. *Contrary to what the notation might suggest, $t_1 \neq_\phi t_2$ is not equivalent to $\neg(t_1 =_\phi t_2)$. For example, two tasks may be neither ϕ -bound nor ϕ -separated.*

3.2.3 Authorization Processes

In this section, we define the semantics of the authorization constraints, using authorization processes. The authorization process, for a given authorization constraint, may engage in any point event and any task execution event not prohibited by the authorization constraint. The definitions are from [BBK11], with only slight changes.

Definition 3.11 (BoD Authorization Process). For a BoD constraint $b = (T, O)$, $A_b(\mathcal{U})$ is the *authorization process for b* , with

$$\begin{aligned}
A_b(U) &= (t.u) : T \times U \rightarrow A_b(\{u\}) \\
&\quad \square o : O \rightarrow A_b(\mathcal{U}) \\
&\quad \square (t.u) : (T \setminus T) \times \mathcal{U} \rightarrow A_b(U) \\
&\quad \square o' : \mathcal{O} \setminus O \rightarrow A_b(U) \\
&\quad \square \text{SKIP} , \\
U &\in 2^{\mathcal{U}} .
\end{aligned}$$

$A_b(\mathcal{U})$ is well-defined. It is also deterministic, as the sets $T \times U$, O , $(T \setminus T) \times \mathcal{U}$, $\mathcal{O} \setminus O$ and $\{\checkmark\}$ are pairwise disjoint. The process $A_b(\mathcal{U})$ for some BoD constraint $b = (T, O)$ can be informally described as follows: $A_b(\mathcal{U})$ restricts which users can execute tasks from the set T . After some trace $i \in \Sigma^*$, it behaves like $A_b(U)$ for some $U \subseteq \mathcal{U}$. Then, U is the set of users that can still execute any task in T .

Definition 3.12 (SoD Authorization Process). For an SoD constraint $s = (T_1, T_2, O)$, $A_s(\mathcal{U}, \mathcal{U})$ is the *authorization process for s* , with

$$\begin{aligned}
A_s(U_1, U_2) &= (t.u) : T_1 \times U_1 \rightarrow A_s(U_1, U_2 \setminus \{u\}) \\
&\quad \square (t.u) : T_2 \times U_2 \rightarrow A_s(U_1 \setminus \{u\}, U_2) \\
&\quad \square o : O \rightarrow A_s(\mathcal{U}, \mathcal{U}) \\
&\quad \square (t.u) : (T \setminus \{T_1 \cup T_2\}) \times \mathcal{U} \rightarrow A_s(U_1, U_2) \\
&\quad \square o' : \mathcal{O} \setminus O \rightarrow A_s(U_1, U_2) \\
&\quad \square \text{SKIP} , \\
U_1, U_2 &\in 2^{\mathcal{U}} .
\end{aligned}$$

$A_s(\mathcal{U}, \mathcal{U})$ is well-defined. It is also deterministic, as the sets $T_1 \times U_1$, $T_2 \times U_2$, O , $(T \setminus \{T_1 \cup T_2\}) \times \mathcal{U}$, $\mathcal{O} \setminus O$, and $\{\checkmark\}$ are pairwise disjoint (T_1 and T_2 are disjoint by definition).

Definition 3.13 (Static Authorization Process). For a user-task assignment UT , the *static authorization process* for UT is the process

$$\begin{aligned} A_{UT} &= (t.u) : UT^{-1} \rightarrow A_{UT} \\ &\square o : \mathcal{O} \rightarrow A_{UT} \\ &\square SKIP . \end{aligned}$$

It is easily recognizable that A_{UT} is well-defined and deterministic.

Definition 3.14 (Authorization Process). For an authorization policy $\phi = (UT, S, B)$, the *authorization process* for ϕ is the process

$$A_\phi = A_{UT} \parallel \left(\parallel_{s \in S} A_s \right) \parallel \left(\parallel_{b \in B} A_b \right).$$

The authorization process for the policy as a whole is the parallel composition of the policy's static authorization process and all of the SoD and BoD authorization processes of the SoD and BoD constraints contained in the policy. By the trace semantics of CSP the set of traces of a parallel composition of processes is the intersection of the sets of traces of the composed processes.

$$\mathsf{T}(A_\phi) = \mathsf{T}(A_{UT}) \cap \bigcap_{s \in S} \mathsf{T}(A_s) \cap \bigcap_{b \in B} \mathsf{T}(A_b).$$

3.2.4 Properties of the Authorization Processes

The algorithm for enforcing constraints, which is presented in [BBK11], does not depend on the dynamic semantics of the enforcement constraints. However, our improved algorithm, which we introduce in Chapter 5, does. It depends on which traces exactly are permitted by the authorization processes. We describe the permissiveness of the authorization processes in terms of the possible continuations for a given trace. We show in the following lemmas that our authorization processes, after engaging in any trace of events, always offer all events in \mathcal{O} , and never offer less task-execution events after engaging in an event in \mathcal{O} . Also, that for two task-execution events, after engaging in one the other is still offered unless it is prohibited by the underlying constraint. Furthermore, we show which task-execution events the authorization processes offer initially.

Lemma 3.2 (Properties of the SoD Authorization Process). *Let s be an SoD Constraint $s = (T_1, T_2, O)$. Let $i \in \mathsf{T}(A_s(\mathcal{U}, \mathcal{U})) \cap \Sigma^*$. Let $A_s^i := A_s(\mathcal{U}, \mathcal{U}) \setminus i$. Then the following statements hold:*

1. $\mathcal{O} \subseteq \text{initials}(A_s^i)$
2. $\forall o \in \omega(A_s^i) : \chi(A_s^i) \subseteq \chi(A_s^i \setminus \langle o \rangle)$
3. $\forall t_1.u_1, t_2.u_2 \in \chi(A_s^i) : (t_1 \neq_s t_2 \Rightarrow u_1 \neq u_2) \Rightarrow t_2.u_2 \in \chi(A_s^i \setminus \langle t_1.u_1 \rangle)$
4. $\mathcal{T} \times \mathcal{U} \subseteq \text{initials}(A_s(\mathcal{U}, \mathcal{U}))$

Proof. Let s be an SoD Constraint $s = (T_1, T_2, O)$. Let $i \in \mathsf{T}(A_s) \cap \Sigma^*$. Let $A_s^i = A_s(\mathcal{U}, \mathcal{U}) \setminus i$.

Statement 1: This holds by rules 3 and 5 of the definition of A_s .

Statement 2: Consider rules 3 and 5 of the definition of A_s . Let $o \in \omega(A_s^i)$.

Case $o \in \mathcal{O}$: Then $A_s^i \setminus \langle o \rangle = A_s(\mathcal{U}, \mathcal{U})$. Therefore, by rules 1, 2, and 4:

$$\chi(A_s^i \setminus \langle o \rangle) = (T_1 \times \mathcal{U}) \cup (T_2 \times \mathcal{U}) \cup ((\mathcal{T} \setminus (T_1 \cup T_2)) \times \mathcal{U}) = \mathcal{T} \times \mathcal{U}$$

So $\chi(A_s^i \setminus \langle o \rangle)$ is maximal and therefore $\chi(A_s^i) \subseteq \chi(A_s^i \setminus \langle o \rangle)$ holds.

Case $o \in \mathcal{O} \setminus \mathcal{O}$: Then $A_s^i \setminus \langle o \rangle = A_s^i$ so by equality $\chi(A_s^i) \subseteq \chi(A_s^i \setminus \langle o \rangle)$ holds.

Statement 3: Assume $s = (T_1, T_2, O)$ and $i \in \mathsf{T}(A_s) \cap \Sigma^*$ and $t_1.u_1, t_2.u_2 \in \chi(A_s \setminus i)$. Assume

$t_1 \neq_s t_2 \Rightarrow u_1 \neq u_2$. From the definition of SoD constraints, $T_1 \cap T_2 = \emptyset$. Let U_1, U_2 be so that $A_s(U_1, U_2) =_{\top} A_s(\mathcal{U}, \mathcal{U}) \setminus i$.

1. Case $t_1 \in T_1 \cup T_2 \wedge t_2 \in T_1 \cup T_2$:
 - (a) Case $t_1 \in T_1$:
 - i. Case $t_2 \in T_2$: In this case $t_1 \neq_s t_2$ holds, and by assumption $u_1 \neq u_2$. Rule 1 applies for $t_1.u_1$. In $A_s^i \setminus \langle t_1.u_1 \rangle =_{\top} A_s(U_1, U_2 \setminus \{u_1\})$, Rule 2 applies for $t_2.u_2$.
 - ii. Case $t_2 \notin T_2$: Here $t_2 \in T_1$. The first rule of A_s applies. This rule is repeatable. Therefore, $t_2.u_2 \in \chi(A_s^i \setminus \langle t_1.u_1 \rangle)$.
 - (b) Case $t_1 \notin T_1$: Here $t_1 \in T_2$.
 - i. Case $t_2 \in T_2$: The second rule of A_s applies. This rule is repeatable. Therefore, $t_2.u_2 \in \chi(A_s^i \setminus \langle t_1.u_1 \rangle)$.
 - ii. Case $t_2 \notin T_2$: Here $t_2 \in T_1$, so $t_1 \neq_s t_2$ holds, and by assumption $u_1 \neq u_2$. Rule 2 applies for $t_1.u_1$. Rule 1 applies for $t_2.u_2$ in $A_s(U_1 \setminus \{u_1\}, U_2) =_{\top} A_s^i \setminus \langle t_1.u_1 \rangle$.
2. Case $\neg(t_1 \in T_1 \cup T_2 \wedge t_2 \in T_1 \cup T_2)$:
 - (a) Case $t_2 \notin T_1 \cup T_2$: Here $t_2.u_2$ is a member of the guard of rule 4, independently of the state. Therefore, $t_2.u_2 \in \chi(A_s^i \setminus \langle t_1.u_1 \rangle)$.
 - (b) Case $t_2 \in T_1 \cup T_2$:
 - i. Case $t_1 \in T_1 \cup T_2$: This contradicts the assumption $\neg(t_1 \in T_1 \cup T_2 \wedge t_2 \in T_1 \cup T_2)$.
 - ii. Case $t_1 \notin T_1 \cup T_2$: Rule 4 applies. This rule is repeatable. Therefore, $t_2.u_2 \in \chi(A_s^i \setminus \langle t_1.u_1 \rangle)$.

Statement 4: This follows from the guards of the rules 1, 2, and 4 of the definition of $A_s(\mathcal{U}, \mathcal{U})$ with $(T_1 \times \mathcal{U}) \cup (T_2 \times \mathcal{U}) \cup ((\mathcal{T} \setminus (T_1 \cup T_2)) \times \mathcal{T}) = \mathcal{T} \times \mathcal{U}$. \square

Lemma 3.3 (Properties of the BoD Authorization Process). *Let b be an BoD Constraint $b = (T, O)$. Let $i \in \top(A_b) \cap \Sigma^*$. Let $A_b^i := A_b(\mathcal{U}) \setminus i$. Then the following statements hold:*

1. $\emptyset \subseteq \text{initials}(A_b^i)$
2. $\forall o \in \omega(A_b^i) : \chi(A_b^i) \subseteq \chi(A_b^i \setminus \langle o \rangle)$
3. $\forall t_1.u_1, t_2.u_2 \in \chi(A_b^i) : (t_1 =_b t_2 \Rightarrow u_1 = u_2) \Rightarrow t_2.u_2 \in \chi(A_b^i \setminus \langle t_1.u_1 \rangle)$.
4. $\mathcal{T} \times \mathcal{U} \subseteq \text{initials}(A_b(\mathcal{U}))$

Proof. Let b be an BoD Constraint $b = (T, O)$. Let $i \in \top(A_b) \cap \Sigma^*$. Let $A_b^i = A_b(\mathcal{U}) \setminus i$.

Statement 1: This holds by rules 2 and 4 of the definition of A_b .

Statement 2: Consider rules 2 and 4 of the definition of A_b . Let $o \in \omega(A_b^i)$.

Case $o \in O$: Then $A_b^i \setminus \langle o \rangle = A_b(\mathcal{U})$. Therefore, by rules 1 and 3:

$$\sigma(A_b^i \setminus \langle o \rangle) = (T \times \mathcal{U}) \cup ((\mathcal{T} \setminus T) \times \mathcal{U}) = \mathcal{T} \times \mathcal{U}$$

So $\chi(A_b^i \setminus \langle o \rangle)$ is maximal and therefore $\chi(A_b^i) \subseteq \chi(A_b^i \setminus \langle o \rangle)$ holds.

Case $o \in \mathcal{O} \setminus O$: Then $A_b^i \setminus \langle o \rangle = A_b^i$ so by equality $\chi(A_b^i) \subseteq \chi(A_b^i \setminus \langle o \rangle)$ holds.

Statement 3: Assume $b = (T, O)$, $i \in \top(A_b)$, $t_1.u_1, t_2.u_2 \in \chi(A_b^i)$. Let U be so that $A_b(U) =_{\top} A_b^i$.

1. Assume $u_1 = u_2$:
 - (a) Case $t_1 \in T$: Rule 1 applies. $A_b(U) \setminus t_1.u_1 =_{\top} A_b(\{u_1\})$.
 - i. Case $t_2 \in T$: Now $t_2.u_2 \in T \times \{u_1\}$, and therefore $t_2.u_2 \in \chi(A_b^i \setminus \langle t_1.u_1 \rangle)$.
 - ii. Case $t_2 \notin T$: Here $t_2.u_2 \in \chi(A_b^i \setminus \langle t_1.u_1 \rangle)$, because of rule 3.
 - (b) Case $t_1 \notin T$: Rule 3 applies. $A_b(U) \setminus \langle t_1.u_1 \rangle =_{\top} A_b(U)$. Therefore $t_2.u_2 \in \chi(A_b^i \setminus \langle t_1.u_1 \rangle)$.

2. Assume $\neg(t_1 =_b t_2)$: Then by definition of $=_b$ holds $\neg(t_1 \in T \wedge t_2 \in T)$.
 - (a) Case $t_1 \in T$: So $t_2 \notin T$. Then $t_2.u_2 \in \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$, because of rule 3.
 - (b) Case $t_1 \notin T$: Then rule 3 applies and $A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle =_{\top} A_b(\mathcal{U}) \setminus i$.

Statement 4: This follows from the guards of the rules 1 and 3 of the definition of $A_b(\mathcal{U})$ with $(T \times \mathcal{U}) \cup ((\mathcal{T} \setminus T) \times \mathcal{T}) = \mathcal{T} \times \mathcal{U}$. \square

Lemma 3.4 (Properties of the Static Authorization Process). *Let UT be an user-task assignment. Let i be an unterminated trace of A_{UT} , i.e. $i \in \mathbb{T}(A_{UT}) \cap \Sigma^*$.*

1. $\mathcal{O} \subseteq \text{initials}(A_{UT} \setminus i)$
2. $\chi(A_{UT} \setminus i) = \chi(A_{UT})$
3. $\chi(A_{UT}) = UT^{-1}$

Proof. Let UT be an user-task assignment. Let $i \in \mathbb{T}(A_{UT}) \cap \Sigma^*$. Statement 1: This holds by rule 2 of the definition of A_{UT} .

Statement 2: This holds because all guards are constant.

Statement 3: This holds by rule 1, and because there are no other rules including task-execution events in its guard. \square

The authorization process for a policy $\phi = (UT, S, B)$ permits any behavior that *all* of the authorization processes A_{UT} , $A_s(\mathcal{U}, \mathcal{U})$ for each $s \in S$, and $A_b(\mathcal{U})$ for each $b \in B$, permit.

Lemma 3.5 (Properties of the Authorization Process for a Policy). *Let $\phi = (UT, S, B)$ be an authorization policy. Let i be an unterminated trace of A_ϕ , i.e. $i \in \mathbb{T}(A_\phi) \cap \Sigma^*$. Then the following statements hold:*

1. $\mathcal{O} \subseteq \text{initials}(A_\phi \setminus i)$
2. $\forall o \in \omega(A_\phi \setminus i) : \chi(A_\phi \setminus i) \subseteq \chi(A_\phi \setminus i^\wedge \langle o \rangle)$
3. $\forall t_1.u_1, t_2.u_2 \in \chi(A_\phi \setminus i) :$
 $(t_1 =_\phi t_2 \Rightarrow u_1 = u_2) \wedge (t_1 \neq_\phi t_2 \Rightarrow u_1 \neq u_2) \Rightarrow t_2.u_2 \in \chi(A_\phi \setminus i^\wedge \langle t_1.u_1 \rangle)$
4. $UT^{-1} \subseteq \text{initials}(A_\phi)$

Proof. Let $\phi = (UT, S, B)$. Let $i \in \mathbb{T}(A_\phi) \cap \Sigma^*$.

Statement 1: This follows from the definition of A_ϕ and Lemmas 3.3, 3.2 and 3.4.

Statement 2: From Lemma 3.4, statement 2, follows $\forall o \in \omega(A_{UT} \setminus i) : \chi(A_{UT} \setminus i) \subseteq \chi(A_{UT} \setminus i^\wedge \langle o \rangle)$ since $i^\wedge \langle o \rangle \in \mathbb{T}(A_{UT}) \cap \Sigma^*$ and $\forall i \in \mathbb{T}(A_{UT}) \cap \Sigma^* : \chi(A_{UT} \setminus i) = \chi(A_{UT})$. Then the statement follows from the definition of A_ϕ and Lemmas 3.3, 3.2.

Statement 3: Let $t_1.u_1, t_2.u_2 \in \chi(A_\phi \setminus i)$. Therefore $\forall b \in B : t_1.u_1, t_2.u_2 \in \chi(A_b(\mathcal{U}))$ and $\forall s \in S : t_1.u_1, t_2.u_2 \in \chi(A_s(\mathcal{U}, \mathcal{U}))$ and $t_1.u_1, t_2.u_2 \in \chi(A_{UT})$. Let $t_1 =_\phi t_2 \Rightarrow u_1 = u_2$ and $t_1 \neq_\phi t_2 \Rightarrow u_1 \neq u_2$. This implies that $\forall b \in B : t_1 =_b t_2 \Rightarrow u_1 = u_2$, and $\forall s \in S : t_1 \neq_s t_2 \Rightarrow u_1 \neq u_2$. From the definition of A_ϕ we know that if

1. $t_2.u_2 \in \chi(A_{UT} \setminus i^\wedge \langle t_1.u_1 \rangle)$
2. $\forall b \in B : t_2.u_2 \in \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$
3. $\forall s \in S : t_2.u_2 \in \chi(A_s(\mathcal{U}, \mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$

then $t_2.u_2 \in \chi(A_\phi \setminus i^\wedge \langle t_1.u_1 \rangle)$. Therefore we prove the above three conditions.

1. This follows from Lemma 3.4.
2. Let $b \in B$. $t_1 =_b t_2 \Rightarrow u_1 = u_2$ holds. Therefore by Lemma 3.3, $t_2.u_2 \in \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$.

3. Let $s \in S$. $t_1 \neq_s t_2 \Rightarrow u_1 \neq u_2$ holds. Therefore by Lemma 3.2, $t_2.u_2 \in \chi(A_s(\mathcal{U}, \mathcal{U}) \setminus i^{\sim}(t_1.u_1))$.

Statement 4: This holds because of the definition of A_ϕ and because according to Lemma 3.4, $UT^{-1} \subseteq \text{initials}(A_{UT})$, according to Lemma 3.2, $\mathcal{T} \times \mathcal{U} \subseteq \text{initials}(A_s(\mathcal{U}, \mathcal{U}))$, according to Lemma 3.3, $\mathcal{T} \times \mathcal{U} \subseteq \text{initials}(A_b(\mathcal{U}))$, and $UT^{-1} \subseteq \mathcal{T} \times \mathcal{U}$. \square

Chapter 4

Extending BPMN to Support SoD/BoD Authorization

In this chapter we describe how we extend the abstract and concrete syntax of BPMN 2.0 to allow specification of dynamic SoD and BoD constraints. We show first how we extend the graphical notation and the abstract syntax model, and then the extension of the XML serialization. For improved readability, we use a sans-serif font to refer to classes in the Abstract Syntax model, so that they are easily recognizable (e.g., “ReleaseEvent”).

4.1 Extending the Graphical and Abstract Syntax Models

4.1.1 Concepts and Graphical Elements





| Graphic | Name | Abstract Syntax Class |
|---|---------------------|-----------------------|
|  | release event | ReleaseEvent |
|  | SoD constraint node | SoDConstraintNode |
|  | BoD constraint node | BoDConstraintNode |
|  | group node | GroupNode |

Table 4.1: Graphical Elements of the Extension and Corresponding Abstract Syntax Classes

We add four new graphical elements, which are listed in Table 4.1. We describe the use of these elements formally using an extension to the abstract syntax model of BPMN 2.0. This extension is given as a UML diagram in Figure 4.3. The abstract syntax classes corresponding to the graphical elements are also indicated in Table 4.1.

To allow expression of SoD and BoD constraints in BPMN, we use the *association* element already defined in BPMN 2.0. An association connects two diagram elements and is represented in BPMN diagrams as a dotted line.

Our extension adds the concept of an *association node* (`AssociationNode`). It allows to form a “higher-level association”, which involves arbitrarily many diagram elements. This is done by connecting all elements directly to one association node using associations (dotted lines). We add two kinds of association nodes: *constraint nodes* (`ConstraintNode`) and *group nodes* (`GroupNode`). Constraint nodes serve to express constraints, and group nodes add an additional method to group a set of tasks. A constraint node can be either a *BoD constraint node* or *SoD constraint node*.

The graphical representation for an association node is a small circle (smaller than an event). A constraint node has a thick black border and contains a symbol. The SoD constraint node contains the symbol “≠”, and the BoD constraint node contains the symbol “=”. A group node is represented by a filled circle.

We also add a new kind of event, the ReleaseEvent. The symbol used for the release event is a person walking through a door case.

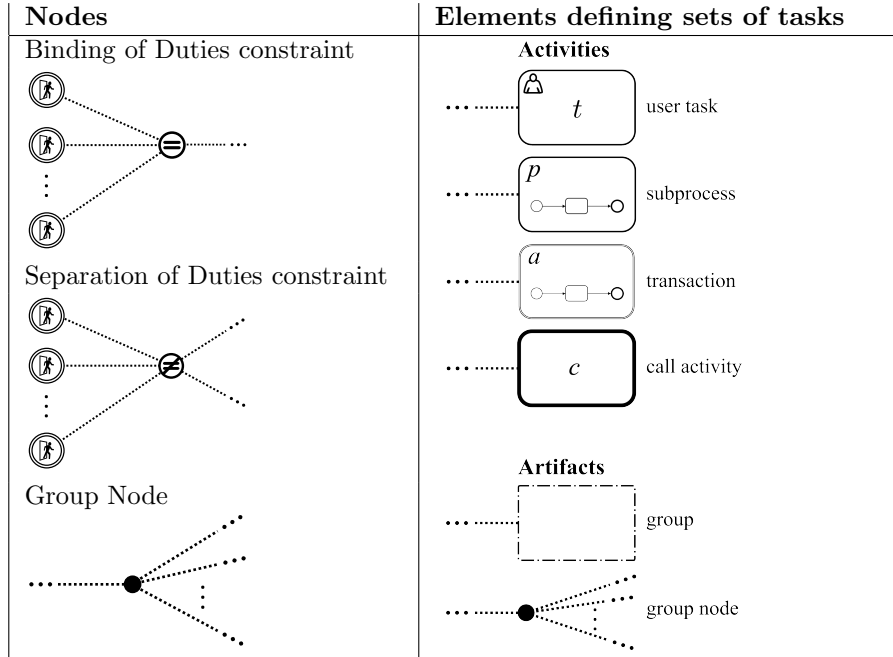


Table 4.2: Concrete Syntax: BoD constraint, SoD constraint and group node can be connected to one (BoD), two (SoD), or arbitrarily many (group node) activities (i.e. tasks, sub-processes, transactions, or call activities), groups, or group nodes.

4.1.2 Graphical Representation of Sets of Tasks

BPMN 2.0 uses the term *group* to denote a set of diagram elements. For specifying the sets of tasks that are involved in a constraint, we connect a graphical representation of the group to the constraint node using an association. We allow all the graphical elements that the BPMN 2.0 standard defines for expressing groups of diagram elements (these are also called *containers*):

- A dot-dashed rectangle with rounded corners (a Group)
- Composite activities, such as Transaction, Sub-Process, and Call Activity

Non-containers can also be used: A single task connected to a constraint node or a group node, is understood as the set containing only that task.

Additionally, for visually expressing sets of tasks, which are positioned in such a way that they cannot be conveniently grouped using a rectangle, we introduce the group node. The group node can be used to denote a set of tasks by individually connecting the tasks to the node by associations. It represents the union of all sets of tasks which are defined by elements associated with it. Several group nodes can be connected to each other, forming a tree of group nodes (no loops are allowed). Every group node in the tree can be used to represent the whole group.

4.1.3 Mapping BPMN Process Models to CSP Processes

For describing the relations between our BPMN annotations and our formal constraint model, we first show an example of how a BPMN process model can be formalized as a workflow process in CSP. The details of this formalization are not important for this work. For a general method for formalizing workflows in CSP, see Wong and Gibbons' paper [WG07].

We use our simple workflow example from Example 2.2.

Example 4.1. We can model the BPMN process given in Figure 2.4 as a CSP process P in the following way. We have $\{Check, Approve, Get, Deliver\} \subseteq \mathcal{T}$ and $\{start, end\} \subseteq \mathcal{O}$, where $start$ and end represent the start and end event.

$$\begin{aligned}
 P &= start \rightarrow (P_{upperbranch} \parallel P_{lowerbranch}) ; end \rightarrow SKIP \\
 &\text{with} \\
 P_{upperbranch} &= Check \rightarrow Approve \rightarrow SKIP \\
 P_{lowerbranch} &= Get \rightarrow (P_{lowerbranch} \sqcap (Deliver \rightarrow SKIP))
 \end{aligned}$$

For example, the following traces are in $\mathbb{T}(P)$, by the trace semantics of CSP:

$$\begin{aligned}
 &\langle start, Check, Approve, Get, Deliver, end, \checkmark \rangle \\
 &\langle start, Check, Get, Get, Deliver, Approve, end, \checkmark \rangle \\
 &\langle start, Get, Check, Get, Approve, Get, Get, Deliver, end, \checkmark \rangle
 \end{aligned}$$

Note how these traces correspond to the sequences mentioned in Example 2.2. The following traces are not in $\mathbb{T}(P)$:

$$\begin{aligned}
 &\langle start, Check, Approve, Approve, Get, Deliver, end, \checkmark \rangle \\
 &\langle start, Get, Deliver, Get, Check, Approve, end, \checkmark \rangle
 \end{aligned}$$

Note that P is not deterministic because the \sqcap operator (internal choice) is used. For example, the following are failures of P , even though Get and $Deliver$ are in $initials(P \setminus \langle start, Get \rangle)$:

$$(\langle start, Get \rangle, \{Get\}), (\langle start, Get \rangle, \{Deliver\})$$

P can decide non-deterministically whether it will accept Get or $Deliver$ after the trace $\langle start, Get \rangle$, which corresponds to the workflow execution engine deciding on which path to take at the exclusive gateway.

4.1.4 Mapping the Formal Model to the Graphical Representation

Adding an SoD constraint (T_1, T_2, O) to a BPMN model involves adding an SoD constraint node and associating it with the groups of tasks T_1 and T_2 , as well each of the release points in O . A release point is expressed in our BPMN extension as a *release event*. Adding a BoD constraint (T, O) works analogously, in this case only one group of tasks is associated with the constraint node.

4.1.5 Mapping the Graphical Representation to the Formal Model

As there is a very clear mapping between the graphical representation of the constraints and our constraint model, it is also clear how to translate any graphically specified constraint into a formally defined constraint:

- A BoD constraint node is associated with a set of release events (each connected individually), and one set of tasks (T) connected through a single association. The set of release events maps to a set of release points O . This specifies the BoD constraint (T, O) .

- An SoD constraint node is associated with a set of release events (each connected individually), and two sets of tasks (T_1, T_2), connected through a single association each. The set of release events maps to a set of release points O . This specifies the SoD constraint (T_1, T_2, O) .

For constraint nodes which are an element of a sub-process, we add an additional release point, which the process engages in when the sub-process is terminated. This has the advantage that the effect of the constraint on the sub-process is independent of the context in which the sub-process is executed.

4.1.6 Examples

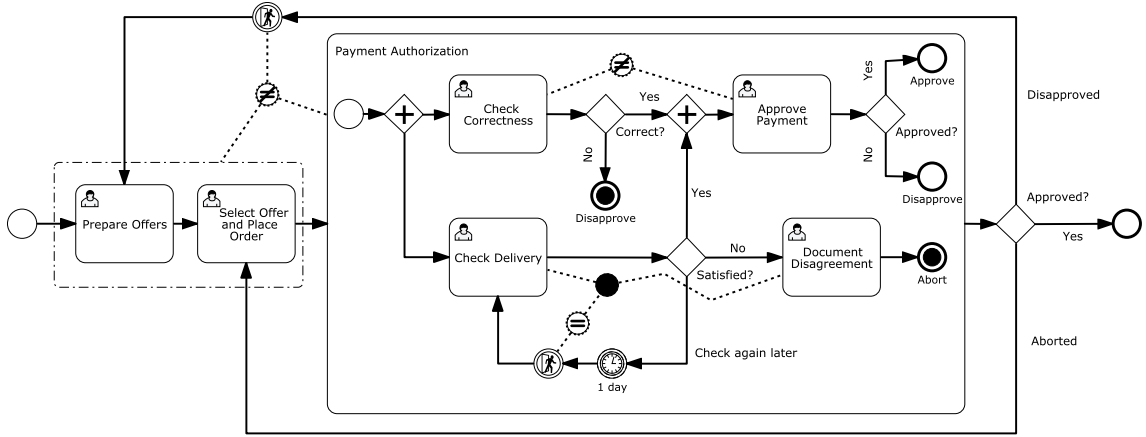


Figure 4.1: Example Workflow 1: Procurement with Constraints

Example 4.2 (Procurement Workflow). Figure 4.1 shows the procurement example workflow, annotated with constraints. There are two SoD constraints, and one BoD constraint. The task *Procurement* is separated from all tasks of the *Payment Authorization* sub-process. The task *Approve Payment* is separated from *Check Correctness*. *Document Disagreement* is bound to *Check Delivery*.

Example 4.3 (An interesting workflow). We use a simpler but no less interesting workflow to demonstrate the relation with our constraint model in detail. It is interesting because it allows us to point out how constraints can conflict with business objectives.

Figure 4.2 shows the workflow of Figure 2.4, annotated with constraints. It also depicts the static authorizations for the set of users $\mathcal{U} = \{Alice, Bob, Claire, Dave\}$, by showing the users as their names and icons next to the tasks that they are authorized to execute:

$$UT = \{(Alice, Check), (Alice, Approve), (Bob, Check), (Bob, Get), (Claire, Get), (Claire, Approve), (Claire, Deliver), (Dave, Get), (Dave, Deliver)\}$$

The set of tasks in this example is: $\mathcal{T} = \{Check, Approve, Get, Deliver\}$. The set S of SoD constraints and the set B of BoD constraints are:

$$S = \{sod_1, sod_2\}, \quad sod_1 = (\{Check\}, \{Approve\}, \emptyset), \quad sod_2 = (\{Approve\}, \{Deliver\}, \emptyset)$$

$$B = \{bod\}, \quad bod = (\{Get, Deliver\}, \{release\})$$

A possible execution trace of this workflow, which also satisfies the constraints is:

$$\langle start, Check.Alice, Get.Bob, Approve.Claire, release, Get.Dave, Deliver.Dave, end, \checkmark \rangle$$

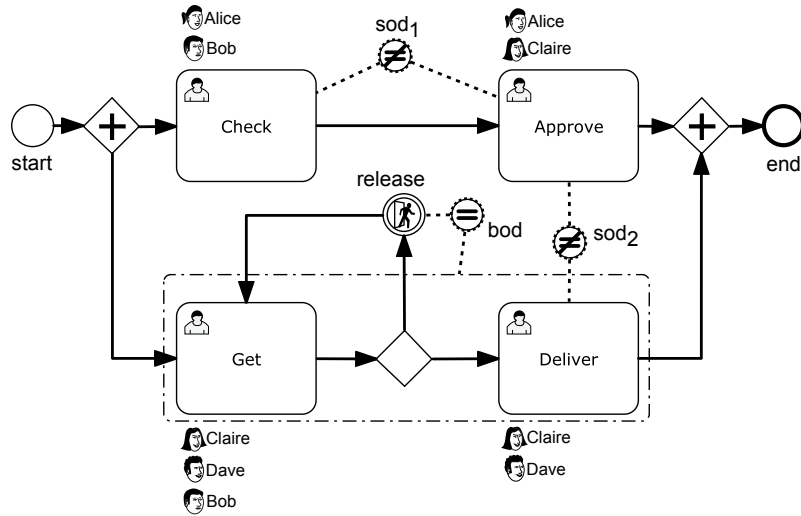


Figure 4.2: Example Workflow 2: An interesting workflow

After the following trace however, the workflow cannot continue without violating a constraint:

$$\langle \text{start}, \text{Check.Alice}, \text{Get.Claire}, \text{Deliver.Claire} \rangle$$

By UT and sod_1 , no user but *Claire* is authorized to execute *Approve*. However, she is not authorized to do so by sod_2 . Therefore there is no user which is authorized to execute the task *Approve*. We call such a situation an *obstruction*.

In any case where *Bob* executes *Get*, then unless the workflow's control flow continues through the release event *release*, the task *Deliver* can never be executed: By the BoD constraint bod no user other than *Bob* can execute it, but *Bob* is not authorized to do so by UT . Therefore, *Bob* should never execute *Get*, in order not to constrain the workflow's control flow.

4.1.7 Abstract Syntax Model of the Extension

The abstract syntax model presented in Figure 4.3 shows how the elements we introduce relate to existing BPMN 2.0 elements: They are subclasses of *Event*, *Artifact* or *Association* (which itself is a subclass of *Artifact*). Of our extension classes, all except *ReleaseEvent* are therefore subclasses of *Artifact*. As such, they inherit the special properties of artifacts. For example, as explicitly required by the BPMN 2.0 specification [Obj11], they cannot participate in message or sequence flows.

The subclasses of *Association* exist in the abstract syntax model only for specifying what kind of associations are allowed as part of a constraint specification. *targetRef* and *sourceRef* are attributes of *Association* and indicate the two elements that the association connects. We redefine these attributes in subclasses of *Association*, restricting the set of objects that they may refer to. We do this using directed UML associations. For example, a *release association* is an association that connects release events to constraint nodes. Consequently, in the abstract syntax only a *ReleaseEvent* can be the *sourceRef* for *ReleaseAssociation*, and only a *ConstraintNode* can be the *targetRef*.

TaskSetAssociation is an abstract class that generalizes associations which connect association nodes to sets of tasks. *TaskSetAssociation* plays the role *incomingTaskSet* for *AssociationNode*. In subclasses of *AssociationNode* corresponding to constraint nodes, the cardinality of *incomingTaskSet* is restricted, as described in the diagram by OCL [BRJ05] constraints. They specify that instances of *BoDConstraintNode* must be associated with exactly one set of task. For instances of *SoDConstraintNode*, the number of associated task sets must be exactly two. The subclasses

Listing 4.1: Full Listing of BPMNAuthorization.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://zurich.ibm.com/dru/bpmnauth"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:auth="http://zurich.ibm.com/dru/bpmnauth"
  xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  elementFormDefault="qualified">

  <import namespace="http://www.omg.org/spec/BPMN/20100524/MODEL"
    schemaLocation="../../spec/BPMN/20100501/BPMN20.xsd"/>

  <element name="sodConstraintNode" type="auth:tAssociationNode"/>
  <element name="bodConstraintNode" type="auth:tAssociationNode"/>
  <element name="groupNode" type="auth:tAssociationNode"/>
  <element name="tasksetAssociation" type="bpmn:tAssociation"/>
  <element name="releaseAssociation" type="bpmn:tAssociation"/>
  <element name="releaseEventDefinition"
    type="auth:tReleaseEventDefinition"/>

  <complexType name="tAssociationNode">
    <complexContent>
      <extension base="bpmn:tArtifact"/>
    </complexContent>
  </complexType>

  <complexType name="tReleaseEventDefinition">
    <complexContent>
      <extension base="bpmn:tEventDefinition"/>
    </complexContent>
  </complexType>
</schema>
```

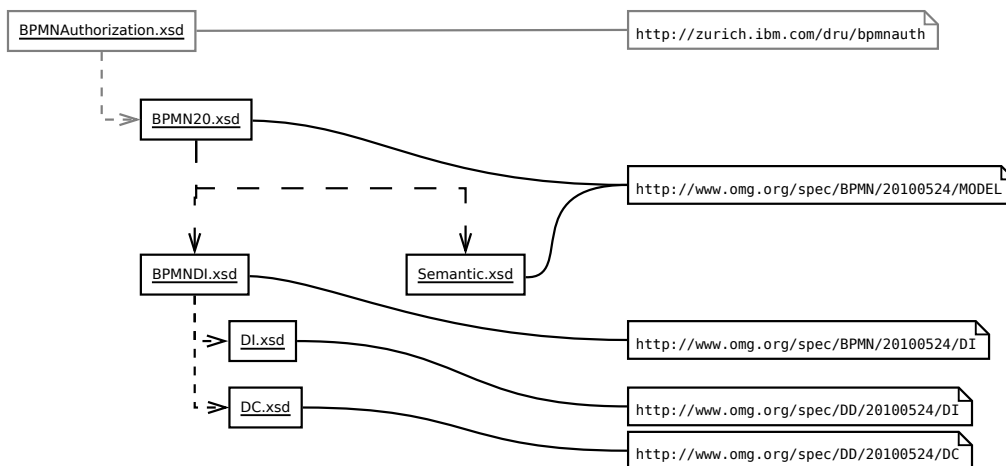


Figure 4.4: Schema Files with Extension

The new elements can in principle be used within any element of the existing BPMN schema, wrapped in an `<extensionElements>` element. They should however be used within the context that they depend on. For example, artifacts describing constraints should be used within a process definition.

An example XML representation of a process that includes elements of our extension is given in Listing 4.2.

Listing 4.2: Example Workflow Serialization

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions id="example_definitions-01"
typeLanguage="http://www.w3.org/2001/XMLSchema"
expressionLanguage="http://www.w3.org/1999/XPath"
targetNamespace="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:auth="http://zurich.ibm.com/dru/bpmnauth">
<process isClosed="false" processType="None" id="process-01">
  <extensionElements>
    <auth:sodConstraintNode id="sod-01"/>
    <auth:tasksetAssociation id="tsa-01"
      targetRef="task-01" sourceRef="sod-01"/>
    <auth:tasksetAssociation id="tsa-02"
      targetRef="task-02" sourceRef="sod-01"/>
    <auth:releaseAssociation id="rla-01"
      targetRef="release-01" sourceRef="sod-01"/>
  </extensionElements>
  <laneSet id="laneset-01">
    <lane id="lane-01">
      <flowNodeRef>event-01</flowNodeRef><flowNodeRef>event-02</flowNodeRef>
      <flowNodeRef>task-01</flowNodeRef><flowNodeRef>task-02</flowNodeRef>
      <flowNodeRef>release-01</flowNodeRef>
    </lane>
  </laneSet>
  <intermediateThrowEvent id="release-01">
    <extensionElements>
      <auth:releaseEventDefinition id="release_def-01"/>
    </extensionElements>
  </intermediateThrowEvent>
  <startEvent name="" id="event-01"/> <endEvent name="" id="event-02"/>
  <task name="Task_A" id="task-01"/> <task name="Task_B" id="task-02"/>
  <sequenceFlow sourceRef="event-01" targetRef="task-01" id="seqflow-01"/>
  <sequenceFlow sourceRef="task-01" targetRef="task-02" id="seqflow-02"/>
  <sequenceFlow sourceRef="task-02" targetRef="release-01" id="seqflow-03"/>
  <sequenceFlow sourceRef="release-01" targetRef="event-02" id="seqflow-04"/>
</process>
<bpmndi:BPMNDiagram>
  <bpmndi:BPMNPlane>
    <bpmndi:BPMNShape bpmnElement="event-01">
      <dc:Bounds x="0" y="1" width="1" height="1"/>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNEdge bpmnElement="seqflow-01">
      <di:waypoint x="0" y="1"/><di:waypoint x="1" y="1"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNShape bpmnElement="task-01">
      <dc:Bounds x="2" y="1" width="1" height="1"/>
    </bpmndi:BPMNShape>
    <!-- [...] -->
    <bpmndi:BPMNShape bpmnElement="release-01">
      <dc:Bounds x="4" y="1" width="1" height="1"/>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNShape bpmnElement="sod-01">
      <dc:Bounds x="2.5" y="2" width="1" height="1"/>
    </bpmndi:BPMNShape>
    <bpmndi:BPMNEdge bpmnElement="tsa-01">
      <di:waypoint x="2" y="1"/><di:waypoint x="2.5" y="2"/>
    </bpmndi:BPMNEdge>
    <bpmndi:BPMNEdge bpmnElement="tsa-02">
      <di:waypoint x="3" y="1"/><di:waypoint x="2.5" y="2"/>
    </bpmndi:BPMNEdge>
  </bpmndi:BPMNPlane>
</bpmndi:BPMNDiagram>
</definitions>

```

Chapter 5

Constraint Analysis and Enforcement

In this chapter, we introduce two methods for enforcing constraints while preventing obstructions. The first method is based on the approach from [BBK11]. The second is our improved approach.

5.1 Definitions

Authorization policies may contain contradictions. For example, a policy can specify that tasks t_1 and t_2 have to be executed by *different* users, while at the same time requiring that they be executed by the *same* user. In our model, this means that after executing one of t_1 and t_2 , no user would be authorized to execute the other task. If the workflow however allows executing both t_1 and t_2 in a given workflow instance, the authorization policy may produce a situation we call *obstruction*, i.e. the workflow permits a certain task to be executed, but the policy does not authorize any user to do so.

Definition 5.1 (Obstruction, from [BBK11]). Let W be a workflow process, ϕ an authorization policy, and $i \in \mathbb{T}(W [\pi^{-1}])$ a workflow trace of W . We say that i is *obstructed* if there exists a task t such that $i[\pi] \hat{\ } t \in \mathbb{T}(W)$ but there does not exist a user u such that $i \hat{\ } \langle t.u \rangle \in \mathbb{T}(A_\phi)$.

5.1.1 Enforcement Process

Definition 5.2 (Enforcement Process, from [BBK11]). For a workflow W and an authorization policy ϕ for W , an enforcement process for ϕ on W is a process E that satisfies the conditions:

1. $A_\phi \sqsubseteq_{\mathbb{T}} E$
2. $(W [\pi^{-1}] \parallel E) [\pi] =_{\mathbb{F}} W$

Condition 1 states that the traces of E must be a subset of the traces that satisfy the policy ϕ . Condition 2 states that the constrained workflow $W [\pi^{-1}] \parallel E$ is equivalent to the original workflow at the workflow level, which implies that $W [\pi^{-1}] \parallel E$ is never obstructed.

5.1.2 Enforcement Process Existence (EPE) Problem

The problem of deciding whether an enforcement process exists for some policy ϕ on some workflow W , has been formalized in [BBK11] as the *Enforcement Process Existence (EPE) Problem*. The authors showed it to be at least NP-hard, but decidable if the set of users \mathcal{U} and the set of states of W are finite. They also provide approximative algorithms for solving EPE. The algorithms are approximative in the sense that there may be false negatives, i.e. cases where algorithm cannot confirm the existence of an enforcement process, even though such a process exists.

5.2 Approximative Enforcement Processes

Note 5.1. We assume in this chapter that a workflow process W and an authorization policy $\phi = (UT, S, B)$ are given. We discuss authorization processes for ϕ on W . We define $T = \{t \in \mathcal{T} \mid \exists i \in \mathsf{T}(W), t \in i\}$, i.e. the set of tasks that the workflow W engages in.

In this thesis, we do not attempt to provide a better solution to EPE than the one given in [BBK11]. Rather, we improve the enforcement process that they provide, without relaxing the condition for its existence, but so that it allows more traces *if* it exists: First we introduce the *static* enforcement process $E_{\phi, T, V}^{static}$, which corresponds to their enforcement process. We then present our *dynamic* enforcement process $D_{\phi, T}$, for which the following holds:

$$\forall V : \mathsf{T}(E_{\phi, T, V}^{static}) \subseteq \mathsf{T}(D_{\phi, T})$$

In addition, we give a trace from $\mathsf{T}(D_{\phi, T})$ which is not an element of $E_{\phi, T, V}^{static}$ for any V (see Example 5.2). Therefore, our dynamic enforcement process is more general in the sense that it allows traces which no static enforcement process allows. Formally, the set of traces of $D_{\phi, T}$ is thus a proper superset:

$$\bigcup_V \mathsf{T}(E_{\phi, T, V}^{static}) \subset \mathsf{T}(D_{\phi, T})$$

5.2.1 Satisfying Task-User Assignments

We introduce the term of a “satisfying” task-user assignment, which we use in the next sections for presenting enforcement processes:

One approach to satisfy constraints imposed by a policy ϕ on a workflow W , is to try and find a task-user assignment $V \subseteq \mathcal{T} \times \mathcal{U}$ which relates each task in T to one or more users, with the property that letting tasks be executed only by the assigned users prevents any violation of the policy ϕ . We call a task-user assignment V with this property an assignment that *satisfies* the policy ϕ , the set of tasks T , and the assignments \bar{V} and \hat{V} , if $\bar{V} \subseteq V \subseteq \hat{V}$ holds. We call \bar{V} a *minimal assignment*, and \hat{V} a *maximal assignment*. We now define this formally:

Definition 5.3. A relation $V \subseteq \mathcal{T} \times \mathcal{U}$ *satisfies* an authorization policy $\phi = (UT, S, B)$, a set of tasks T , and two task-user assignments \bar{V} and \hat{V} , written $V \mid (\phi, T, \bar{V}, \hat{V})$, if the following conditions hold:

1. $\bar{V} \subseteq V \subseteq \hat{V} \cap UT^{-1}$
2. $\forall i \in \mathsf{T}(A_\phi) : V \subseteq \chi(A_\phi \setminus i) \Rightarrow (\forall e \in (V \cup \mathcal{O}) : V \subseteq \chi(A_\phi \setminus i \cdot \langle e \rangle))$
3. $\forall t \in T \exists u : t.u \in V$

We state condition 2 of the definition in words, in an attempt to provide a more intuitive description: *For any trace i which satisfies ϕ , if $A_\phi \setminus i$ offers all events in V , then after engaging in any event from $V \cup \mathcal{O}$, it still offers all events in V .*

For convenience, we define a variant where \bar{V} and \hat{V} have a default value:

Definition 5.4. A relation $V \subseteq \mathcal{T} \times \mathcal{U}$ *satisfies* an authorization policy $\phi = (UT, S, B)$, a set of tasks T , written $V \mid (\phi, T)$, if and only if $V \mid (\phi, T, \emptyset, T \times \mathcal{U})$.

Note that $V \mid (\phi, T, \bar{V}, \hat{V})$ implies $V \mid (\phi, T)$ for any \bar{V} and \hat{V} . Informally we call a task-user assignment V such that $V \mid (\phi, T)$ simply a *satisfying* assignment, if ϕ and T are clear from the context, such as given in a diagram. We describe in Section 5.3 how to compute a satisfying assignment for a given policy.

Example 5.1. By revisiting Example 4.3, we illustrate how satisfying assignments can be used for enforcing constraints and prevent obstructions. Recall the values of ϕ and T from Example 4.3.

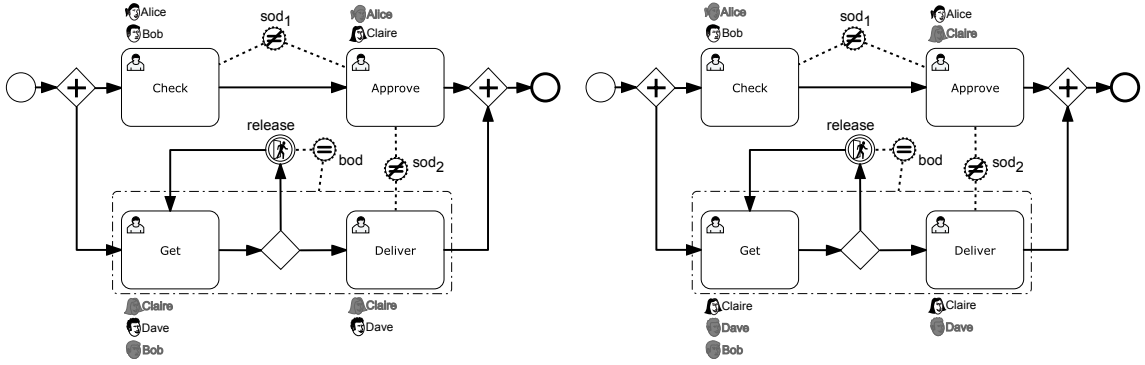


Figure 5.1: Example BPMN-Process 2: Two satisfying assignments

In Figure 5.1, we show two task-user assignments V_1 and V_2 (indicated by icons and names of the users which are not faded to gray):

$$V_1 = \{(Check, Alice), (Check, Bob), (Approve, Claire), (Get, Dave), (Deliver, Dave)\}$$

$$V_2 = \{(Check, Bob), (Approve, Alice), (Get, Claire), (Deliver, Claire)\}$$

$V_1|(\phi, T)$ and $V_2|(\phi, T)$ hold (we show in Section 5.3 how to verify this). If only events from either $V_1 \cup \mathcal{O}^\vee$ or $V_2 \cup \mathcal{O}^\vee$ are executed, then no obstruction can occur, by condition 2 of Definition 5.3. Condition 3 of Definition 5.3 guarantees that for each task in T , there is at least one user who is authorized to execute it. Therefore we can either use a process that only engages in events from $V_1 \cup \mathcal{O}^\vee$, or only in events from $V_2 \cup \mathcal{O}^\vee$, as an enforcement process for any workflow W for which T are the only tasks it engages in. We call this kind of enforcement process *static*, as it does not change its state and always offers the same set of events. We define such a process formally in the following section.

5.2.2 Static Approximative Enforcement Process

Given a set of tasks T , a policy ϕ , and a relation $V \subseteq T \times \mathcal{U}$ such that $V|(\phi, T)$, we can construct an approximative enforcement process as follows:

$$E_{\phi, T, V}^{static} = (t.u) : V \rightarrow E_{\phi, T, V}^{static}$$

$$\square o : \mathcal{O} \rightarrow E_{\phi, T, V}^{static}$$

$$\square SKIP .$$

Note 5.2. As explained in Note 2.2, we interpret the semantics of the process *SKIP* slightly differently than as defined in CSP [Ros05]: We treat $E_{\phi, T, V}^{static}$ as a deterministic process, which will only terminate when the environment allows it.

Theorem 1. Let W be a workflow process. Let T be the set of tasks that W engages in. Let ϕ be an authorization policy. Let $V \subseteq T \times \mathcal{U}$ be such that $V|(\phi, T)$. Then $E_{\phi, T, V}^{static}$ is an enforcement process for ϕ on W .

Theorem 1 has essentially already been proved in [BBK11], but by constructing the enforcement process differently: Their approach corresponds to renaming the workflow process with a relation that is an extension of the task-user assignment. We could use the same approach and show that with $R := V \cup (\mathcal{O} \times \mathcal{O}) \cup \{\checkmark, \checkmark\}$, then $W[R]$ is an enforcement process for ϕ on W . However, since this form of construction is not possible for the dynamic approximative enforcement process that we present in the next section, we use our more general form of construction and prove that it is correct. This enables us to introduce principles and proof techniques in this section that we can

re-use in the next section. We start with establishing some facts about the process $E_{\phi, T, V}^{static}$, and then proceed to introducing the proof principles and lemmas necessary for proving Theorem 1.

Lemma 5.1. *Let T be a set of tasks. Let ϕ be an authorization policy. Let $V \subseteq T \times \mathcal{U}$ be such that $V | (\phi, T)$. Then the following statements hold:*

1. $E_{\phi, T, V}^{static}$ is deterministic.
2. $\forall i \in \mathsf{T}(E_{\phi, T, V}^{static}) \cap \Sigma^* : (T \cup \mathcal{O})^\checkmark \subseteq \pi(\mathit{initials}(E_{\phi, T, V}^{static} \setminus i))$

Proof. Let T be a set of tasks. Let ϕ be an authorization policy. Let $V \subseteq T \times \mathcal{U}$ be such that $V | (\phi, T)$.

1. This follows from the fact that V , \mathcal{O} and $\{\checkmark\}$ are pairwise disjoint.
2. Let $i \in \mathsf{T}(E_{\phi, T, V}^{static})$. Let $I := \mathit{initials}(E_{\phi, T, V}^{static} \setminus i) = V \cup \mathcal{O} \cup \{\checkmark\}$. Let $e' \in (T \cup \mathcal{O})^\checkmark$. We need to show that there exists an $e \in I$ such that $e' = \pi(e)$.
 Case $e' \in T$: Because of $V | (\phi, T)$ there exists an $u \in \mathcal{U}$ such that $e'.u \in I$ and, by the definition of π , $(e'.u, e') \in \pi$.
 Case $e' \in \mathcal{O}$: $e' \in I$ and $(e', e') \in \pi$.
 Case $e' = \checkmark$: $e' \in I$ and $(e', e') \in \pi$.

□

We state two forms of the principle of structural induction, which we use in our proofs. The first one is convenient for proving statements about the traces of a process. The second one is more universal and can be used to prove statements about *all* traces in general.

Induction Principle 1. Given a CSP process P and a predicate on traces $S : \Sigma^{*\checkmark} \rightarrow \{\top, \perp\}$. For convenience we write $\mathsf{T}_j(P)$ for the traces of P with length j . Note that $\mathsf{T}_0(P) = \{\langle \rangle\}$ and $\mathsf{T}_{j+1}(P) = \{i \hat{\ } \langle e \rangle \mid i \in \mathsf{T}_j(P), e \in \mathit{initials}(P \setminus i)\}$. If

1. $S(\langle \rangle)$
2. $\forall i \in \mathsf{T}_j(P) : S(i) \Rightarrow \forall e \in \mathit{initials}(P \setminus i) : S(i \hat{\ } \langle e \rangle)$

then $\forall i \in \mathsf{T}(P) : S(i)$.

In words: If the predicate S holds for all traces of length 0 (there is only one, $\langle \rangle$), and if it is the case that S holds for traces of length j implies that S also holds for traces of length $j + 1$, then S holds for all traces; of arbitrary finite length.

Induction Principle 2. Given a predicate on traces $S : \Sigma^{*\checkmark} \rightarrow \{\top, \perp\}$. For convenience, we write $\Sigma_j^{*\checkmark}$ for the traces with length j . Note that $\Sigma_0^{*\checkmark} = \{\langle \rangle\}$ and $\Sigma_{j+1}^{*\checkmark} = \{i \hat{\ } \langle e \rangle \mid i \in \Sigma_j^*, e \in \Sigma^{\checkmark}\}$. If

1. $S(\langle \rangle)$
2. $\forall i \in \Sigma_j^* : S(i) \Rightarrow \forall e \in \Sigma^{\checkmark} : S(i \hat{\ } \langle e \rangle)$

then $\forall i \in \Sigma^{*\checkmark} : S(i)$.

In words: If the predicate S holds for all traces of length 0 (there is only one, $\langle \rangle$), and if it is the case that S holds for traces of length j implies that S also holds for traces of length $j + 1$, then S holds for all traces; of arbitrary finite length.

The following lemma proves an instance of the rather intuitive idea that if after any of its traces, a process offers a particular set of events, then all traces that contain only elements of this set of events are traces of the process.

Lemma 5.2. *Let E be an execution process. Let $T \subseteq \mathcal{T}$. Then:*

$$\forall i \in \mathsf{T}(E) \cap \Sigma^* : (T \cup \mathcal{O})^\checkmark \subseteq \pi(\mathit{initials}(E \setminus i)) \quad \Rightarrow \quad (T \cup \mathcal{O})^{*\checkmark} \subseteq \mathsf{T}(E[\pi])$$

Proof. Assume:

$$\forall i \in \mathsf{T}(E) \cap \Sigma^* : (T \cup \mathcal{O})^\vee \subseteq \pi(\mathit{initials}(E \setminus i)) \quad (5.1)$$

Proof by structural induction (Induction Principle 2). To be shown:

$$\forall i \in \Sigma^{*\vee} : i \in (T \cup \mathcal{O})^{*\vee} \Rightarrow i \in \mathsf{T}(E[\pi])$$

By equation 2.7, $\mathsf{T}(E[\pi]) = \{i' \mid \exists i : i \pi i' \wedge i \in \mathsf{T}(E)\}$.

Induction base, $i = \langle \rangle$: By the traces model $i \in \mathsf{T}(E[\pi])$.

Induction step, $i \neq \langle \rangle$: Let $i \in \Sigma^*$. *Induction hypothesis:* Assume $i \in (T \cup \mathcal{O})^{*\vee} \Rightarrow i \in \mathsf{T}(E[\pi])$. Let $e \in \Sigma^\vee$. We show that the hypothesis holds for $i \hat{\ } \langle e \rangle$: Assume $i \hat{\ } \langle e \rangle \in (T \cup \mathcal{O})^{*\vee}$. Therefore $i \in (T \cup \mathcal{O})^*$ and by the induction hypothesis $i \in \mathsf{T}(E[\pi])$, and $e \in (T \cup \mathcal{O})^\vee$. Therefore $\exists j : j \pi i \wedge j \in \mathsf{T}(E) \cap \Sigma^*$. By (5.1), $e \in \pi(\mathit{initials}(E \setminus j))$, so let e' be an event that π maps to e . So we have $j \hat{\ } \langle e' \rangle \in \mathsf{T}(E)$ and $(j \hat{\ } \langle e' \rangle) \pi (i \hat{\ } \langle e \rangle)$. Therefore $i \hat{\ } \langle e \rangle \in \mathsf{T}(E[\pi])$. \square

Lemma 5.3. *Given a workflow process W , an authorization policy ϕ for W . Let T be the tasks that W engages in. A authorization process E is an enforcement process for ϕ on W , if the following conditions hold:*

1. $A_\phi \sqsubseteq_{\mathsf{T}} E$
2. $\forall i \in \mathsf{T}(E) \cap \Sigma^* : (T \cup \mathcal{O})^\vee \subseteq \pi(\mathit{initials}(E \setminus i))$
3. E is deterministic

Proof. Let W be a workflow process and ϕ an authorization policy for W . Let T be the tasks that W engages in. Let E be a authorization process. Assume

1. $A_\phi \sqsubseteq_{\mathsf{T}} E$
2. $\forall i \in \mathsf{T}(E) \cap \Sigma^* : (T \cup \mathcal{O})^\vee \subseteq \pi(\mathit{initials}(E \setminus i))$
3. E is deterministic

From assumption 2 follows by Lemma 5.2, and because W is a workflow process that does not engage in tasks other than those in T :

$$\mathsf{T}(W) \subseteq (T \cup \mathcal{O})^{*\vee} \subseteq \mathsf{T}(E[\pi]) \quad (5.2)$$

To be shown is, by Definition 5.2:

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

1. This follows from assumption 1.

2. For improved readability we use the following convention for denoting traces in $\Sigma_{\mathcal{X}}^{*\vee}$ and $\Sigma_{\mathcal{T}}^{*\vee}$: We use plain names (such as “ i ”) to refer to traces in $\Sigma_{\mathcal{X}}^{*\vee}$, and primed names (such as “ i' ”) to refer to traces in $\Sigma_{\mathcal{T}}^{*\vee}$. Similarly, we use plain names for refusal sets of execution processes, and primed names for refusal sets of workflow processes.

First, we show some preliminary results: Since E is deterministic:

$$\mathsf{F}(E) = \{(i, S) \mid i \in \mathsf{T}(E), S \subseteq \Sigma^\vee \setminus \mathit{initials}(E \setminus i)\} \quad (5.3)$$

Let $(i, S) \in \mathsf{F}(E)$ with $i \in \Sigma^*$. Then $S \subseteq \Sigma^\vee \setminus \mathit{initials}(E \setminus i)$. Assumption 2 implies that $\forall t \in T : \exists u \in \mathcal{U} : t.u \in \mathit{initials}(E \setminus i)$. Let $t \in T$, so $\exists u \in \mathcal{U} : t.u \in \mathit{initials}(E \setminus i)$. Since for any event $e \in \mathit{initials}(E \setminus i)$ follows that $e \notin S$, we conclude that

$$\forall (i, S) \in \mathsf{F}(E) : i \in \Sigma^* \Rightarrow \forall t \in T : \exists u \in \mathcal{U} : t.u \notin S \quad (5.4)$$

Let $R, R' \subseteq \Sigma^\vee$ be such that $\pi^{-1}(R') = R$. Let $t.u \in \mathcal{T} \times \mathcal{U} \cap R$. Then by the definition of π , $t \in R'$, so $\forall u \in \mathcal{U} : t.u \in R$.

By the definition of failures of renamed processes (equation (2.9)):

$$F(W [\pi^{-1}]) = \{(i, R) \mid \exists i' : i' \pi^{-1} i \wedge (i', \pi(R)) \in F(W)\} \quad (5.5)$$

$$= \{(i, R) \mid i \in \Sigma_{\mathcal{X}}^{*\vee} \wedge (i [\pi], \pi(R)) \in F(W)\} \text{ (from (3.1))} \quad (5.6)$$

From the definition of failures of synchronized parallel processes:

$$F(W [\pi^{-1}] \parallel E) = \{(i, R \cup S) \mid (i, R) \in F(W [\pi^{-1}]) \wedge (i, S) \in F(E)\} \quad (5.7)$$

Because the processes W and $(W [\pi^{-1}] \parallel E) [\pi]$ do not engage in task-execution events ($\mathcal{T} \times \mathcal{U}$), for any set $X \subseteq \mathcal{T} \times \mathcal{U}$ of task-execution events follows:

$$\forall (i', R') \in F(W) : i' \in \Sigma_{\mathcal{T}}^{*\vee} \wedge \forall X \subseteq \mathcal{T} \times \mathcal{U} : (i', R' \cup X) \in F(W) \quad (5.8)$$

$$\forall (i', R') \in F((W [\pi^{-1}] \parallel E) [\pi]) : i' \in \Sigma_{\mathcal{T}}^{*\vee} \wedge \forall X \subseteq \mathcal{T} \times \mathcal{U} : (i', R' \cup X) \in F((W [\pi^{-1}] \parallel E) [\pi]) \quad (5.9)$$

Note that π^{-1} is total on $\Sigma_{\mathcal{T}}^{*\vee}$.

By the definition of failures of renamed processes, equation (2.9):

$$\begin{aligned} & F((W [\pi^{-1}] \parallel E) [\pi]) \\ &= \{(i [\pi], R') \mid \pi^{-1}(R') = R \cup S \wedge (i, R) \in F(W [\pi^{-1}]) \wedge (i, S) \in F(E)\} \\ &= \{(i [\pi], R') \mid \pi^{-1}(R') = R \cup S \wedge i \in \Sigma_{\mathcal{X}}^{*\vee} \wedge (i [\pi], \pi(R)) \in F(W) \wedge (i, S) \in F(E)\} \text{ (from (5.6))} \end{aligned}$$

We show separately the following propositions:

1. $F((W [\pi^{-1}] \parallel E) [\pi]) \cap \Sigma_{\mathcal{T}}^{*\vee} \times 2^{\Sigma_{\mathcal{T}}^\vee} \subseteq F(W)$
2. $F(W) \cap \Sigma_{\mathcal{T}}^{*\vee} \times 2^{\Sigma_{\mathcal{T}}^\vee} \subseteq F((W [\pi^{-1}] \parallel E) [\pi])$

We then show that this implies $(W [\pi^{-1}] \parallel E) [\pi] \sqsubseteq_F W$ and $W \sqsubseteq_F (W [\pi^{-1}] \parallel E) [\pi]$, which concludes the proof.

1. Let $(i [\pi], R') \in F((W [\pi^{-1}] \parallel E) [\pi]) \cap \Sigma_{\mathcal{T}}^{*\vee} \times 2^{\Sigma_{\mathcal{T}}^\vee}$. Therefore let $i \in \Sigma_{\mathcal{X}}^{*\vee}$, R, S be such that $\pi^{-1}(R') = R \cup S \wedge (i [\pi], \pi(R)) \in F(W) \wedge (i, S) \in F(E)$. From $\pi^{-1}(R') = R \cup S$ follows that $R, S \subseteq \Sigma_{\mathcal{X}}^\vee$.

- Case $i [\pi] \in \Sigma_{\mathcal{T}}^*$: We show that $R' = \pi(R)$:
 - \subseteq : Let $r' \in R'$. Since $R' \subseteq \Sigma_{\mathcal{T}}^\vee$ holds $r' \in \mathcal{T} \vee r' \in \mathcal{O}^\vee$.
 - Case $r' \in \mathcal{T}$: Since $\pi^{-1}(R') = R \cup S$, $\{r'\} \times \mathcal{U} \subseteq R \cup S$. Therefore, with (5.4) we know $\exists u \in \mathcal{U} : r'.u \in R$, and so $r' \in \pi(R)$.
 - Case $r' \in \mathcal{O}^\vee$: In this case $r' \in R \cup S$, but $r' \notin S$, since $\mathcal{O}^\vee \subseteq \text{initials}(E \setminus i)$. Therefore $r' \in R$ and by the definition of π follows $r' \in \pi(R)$.
 - \supseteq : Let $r' \in \pi(R)$. Then $\exists r \in R : (r, r') \in \pi$. Therefore $\exists r \in R \cup S : (r, r') \in \pi$, so $r' \in R'$.
- Case $i [\pi] \notin \Sigma_{\mathcal{T}}^*$ (i ends with \checkmark): In this case $(i [\pi], \Sigma^\vee) \in F(W)$, so by (2.10) also $(i [\pi], \pi(R)) \in F(W)$.

2. Let $(i', R') \in F(W) \cap \Sigma_{\mathcal{T}}^{*\vee} \times 2^{\Sigma_{\mathcal{T}}^\vee}$. Therefore i' is a workflow trace. Let $i \in \Sigma_{\mathcal{X}}^{*\vee}$ be such that $i [\pi] = i'$. By (5.2), $i \in \mathsf{T}(E)$. Since $R' \subseteq \Sigma_{\mathcal{T}}^\vee$, $\exists R \subseteq \Sigma_{\mathcal{X}}^\vee : \pi(R) = R' \wedge \pi^{-1}(R') = R$. With $S := \emptyset$ follows $(i, S) \in F(E)$ from $i \in \mathsf{T}(E)$ and (2.4) and (2.10). Therefore $(i', R') \in F((W [\pi^{-1}] \parallel E) [\pi])$.

We show first $(W [\pi^{-1}] \parallel E) [\pi] \sqsubseteq_F W$ and then $W \sqsubseteq_F (W [\pi^{-1}] \parallel E) [\pi]$.

- Let $(i', R') \in F(W)$. Therefore $(i', R' \cap \Sigma_{\mathcal{T}}^{\checkmark}) \in F(W)$, and by proposition 2 and (5.8) follows with $X = R' \setminus (R' \cap \Sigma_{\mathcal{T}}^{\checkmark}) \subseteq \mathcal{T} \times \mathcal{U}$, that $(i', R') \in F((W [\pi^{-1}] \parallel E) [\pi])$.
- Let $(i', R') \in F((W [\pi^{-1}] \parallel E) [\pi])$. Therefore $(i', R' \cap \Sigma_{\mathcal{T}}^{\checkmark}) \in F((W [\pi^{-1}] \parallel E) [\pi])$, and by proposition 2 and (5.9) follows with $X = R' \setminus (R' \cap \Sigma_{\mathcal{T}}^{\checkmark}) \subseteq \mathcal{T} \times \mathcal{U}$, that $(i', R') \in F(W)$.

We conclude that $(W [\pi^{-1}] \parallel E) [\pi] =_F W$ holds. \square

Finally, we prove Theorem 1 using the lemmas that we proved before:

Proof of Theorem 1. Let W be a workflow process. Let T be the set of tasks that W engages in. Let $\phi = (UT, S, B)$ be an authorization policy. Let $V \subseteq T \times \mathcal{U}$ be such that $V \mid (\phi, T)$. We show that $E_{\phi, T, V}^{static}$ is an enforcement process for ϕ on W . By Lemma 5.1:

1. $E_{\phi, T, V}^{static}$ is deterministic.
2. $\forall i \in \mathsf{T}(E_{\phi, T, V}^{static}) \cap \Sigma^* : (T \cup \mathcal{O})^{\checkmark} \subseteq \pi(\mathit{initials}(E_{\phi, T, V}^{static} \setminus i))$

We show that $A_{\phi} \sqsubseteq_{\mathsf{T}} E_{\phi, T, V}^{static}$ and then apply Lemma 5.3 to conclude the proof.

Let $i \in \mathsf{T}(E_{\phi, T, V}^{static})$. Proof by structural induction (Induction Principle 2). We need to show $\forall i \in \Sigma^{*\checkmark} : i \in \mathsf{T}(E_{\phi, T, V}^{static}) \Rightarrow i \in \mathsf{T}(A_{\phi})$, which implies $A_{\phi} \sqsubseteq_{\mathsf{T}} E_{\phi, T, V}^{static}$. However, for convenience we show that $\forall i \in \Sigma^{*\checkmark} : i \in \mathsf{T}(E_{\phi, T, V}^{static}) \Rightarrow i \in \mathsf{T}(A_{\phi}) \wedge (i \in \Sigma^* \Rightarrow V \subseteq \chi(A_{\phi} \setminus i))$.

Induction base, $i = \langle \rangle$: By the traces model, $i \in \mathsf{T}(A_{\phi})$. Also $V \subseteq \chi(A_{\phi})$, since $V \mid (\phi, T)$ implies $V \subseteq UT^{-1}$ and $UT^{-1} \subseteq \chi(A_{\phi})$ by Lemma 3.5.

Induction step, $i \neq \langle \rangle$: Let $i \in \Sigma^*$. *Induction hypothesis:* Assume $i \in \mathsf{T}(E_{\phi, T, V}^{static}) \Rightarrow i \in \mathsf{T}(A_{\phi}) \wedge (i \in \Sigma^* \Rightarrow V \subseteq \chi(A_{\phi} \setminus i))$. Let $e \in \Sigma^{\checkmark}$. We show that the hypothesis holds for $i \hat{\ } \langle e \rangle$: Assume $i \hat{\ } \langle e \rangle \in \mathsf{T}(E_{\phi, T, V}^{static})$, therefore $i \in \mathsf{T}(E_{\phi, T, V}^{static})$, and by the induction hypothesis $i \in \mathsf{T}(A_{\phi})$ and $V \subseteq \chi(A_{\phi} \setminus i)$. Furthermore $e \in \mathit{initials}(E_{\phi, T, V}^{static} \setminus i) = \mathit{initials}(E_{\phi, T, V}^{static})$. From the definition of $E_{\phi, T, V}^{static}$ follows that $e \in V \vee e \in \mathcal{O} \vee e = \checkmark$. From $V \mid (\phi, T)$ follows $\forall e \in (V \cup \mathcal{O}) : V \subseteq \chi(A_{\phi} \setminus i \hat{\ } \langle e \rangle)$.
Case $e \in V$: $e \in (V \cup \mathcal{O})$, so $V \subseteq \chi(A_{\phi} \setminus i \hat{\ } \langle e \rangle)$. From $V \subseteq \chi(A_{\phi} \setminus i)$ follows $i \hat{\ } \langle e \rangle \in \mathsf{T}(A_{\phi})$.
Case $e \in \mathcal{O}$: $e \in (V \cup \mathcal{O})$, so $V \subseteq \chi(A_{\phi} \setminus i \hat{\ } \langle e \rangle)$. From Lemma 3.5 follows $i \hat{\ } \langle e \rangle \in \mathsf{T}(A_{\phi})$.
Case $e = \checkmark$: By the definition of A_{ϕ} follows $i \hat{\ } \langle e \rangle \in \mathsf{T}(A_{\phi})$. Furthermore $(i \hat{\ } \langle e \rangle \in \Sigma^* \Rightarrow V \subseteq \chi(A_{\phi} \setminus i \hat{\ } \langle e \rangle))$ holds since $i \hat{\ } \langle e \rangle \notin \Sigma^*$.

Therefore we have shown that $A_{\phi} \sqsubseteq_{\mathsf{T}} E_{\phi, T, V}^{static}$, and by Lemma 5.3 follows that $E_{\phi, T, V}^{static}$ is an enforcement process for ϕ on W . \square

5.2.3 Dynamic Approximative Enforcement Process

We show how to construct an enforcement process that is more general than the static approximative enforcement process. It is still approximative in the sense that its construction may not be possible even though *some* enforcement process exists, and if it is possible, the enforcement process may not permit all possible traces allowed by the workflow and the constraints. The construction is possible exactly if a static approximative enforcement process exists. Therefore, it does not allow any solutions for cases where no static approximative enforcement process exists, but solutions which allow more traces.

Definition 5.5. A relation $V \subseteq \mathcal{T} \times \mathcal{U}$ satisfies an authorization policy $\phi = (UT, S, B)$, a task-user assignment \bar{V} , and a set of tasks T after a trace $i \in \Sigma^*$, written $V \models (\phi, T, \bar{V}, i)$, if the following conditions hold:

1. $i \in \mathcal{T}(A_\phi) \wedge \bar{V} \subseteq V \subseteq \chi(A_\phi \setminus i)$
2. $\forall i' \in \mathcal{T}(A_\phi) : V \subseteq \chi(A_\phi \setminus i') \Rightarrow \forall e \in (V \cup \mathcal{O}) : V \subseteq \chi(A_\phi \setminus i' \hat{\ } \langle e \rangle)$
3. $\forall t \in T \exists u : t.u \in V$

Note that for a trace $i \in \mathcal{T}(A_\phi)$ holds $V \models (\phi, T, \bar{V}, i)$ if and only if $i \in \mathcal{T}(A_\phi)$ and $V \models (\phi, T, \bar{V}, \chi(A_\phi \setminus i))$.

We show now by an example how a *dynamic* enforcement process can be more general than a static enforcement process, by taking into account that release events relax constraints.

Example 5.2. We revisit Example 5.1. Recall the values of ϕ and T from Example 4.3. Recall the two task-user assignments V_1 and V_2 from Example 5.1, where $V_1 \models (\phi, T)$ and $V_2 \models (\phi, T)$ hold. Consider the trace $i = \langle \text{start}, \text{Get.Claire}, \text{Check.Bob} \rangle$. Figure 5.2 illustrates the state of the

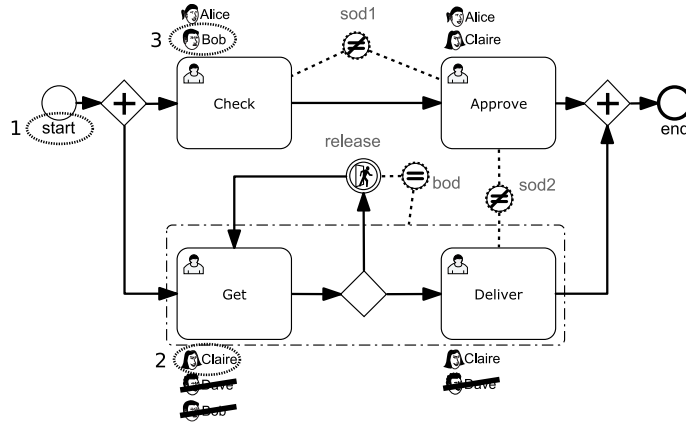


Figure 5.2: Example BPMN-Process 2: State after trace $\langle \text{start}, \text{Get.Claire}, \text{Check.Bob} \rangle$.

workflow after i , with the events in i marked as dotted ellipses with a number that indicates the order of their occurrence. The task-user pairs $(\text{Get}, \text{Dave})$, (Get, Bob) , and $(\text{Deliver}, \text{Dave})$ are crossed out since they are not authorized by the constraint bod . They are not in $\chi(A_\phi \setminus i)$. In this state, who can be authorized to execute *Approve*, without inducing an obstruction? We can authorize any user u for which the following holds:

$$\exists V : V \models (\phi, T, \{\text{Approve}.u\}, \langle \text{start}, \text{Get.Claire}, \text{Check.Bob} \rangle)$$

The reason for this is that by the conditions of Definition 5.5, if such a V exists then the workflow is able to complete by only engaging in events from $V \cup \mathcal{O}$. For example the following holds:

$$V_2 \models (\phi, T, \{\text{Approve}.Alice\}, \langle \text{start}, \text{Get.Claire}, \text{Check.Bob} \rangle)$$

Therefore, the workflow instance can continue with *Approve.Alice* and complete as

$$\langle start, Get.Claire, Check.Bob, Approve.Alice, Deliver.Claire, end, \surd \rangle.$$

This trace is a trace of the static enforcement process $E_{\phi, T, V_2}^{static}$. Consider however

$$i_r := i \hat{\langle release \rangle} = \langle start, Get.Claire, Check.Bob, release \rangle.$$

Then $V_1 \models (\phi, T, \{Approve.Claire\}, i_r)$ holds and therefore the workflow instance can complete as

$$i_t = \langle start, Get.Claire, Check.Bob, release, Approve.Claire, Get.Dave, Deliver.Dave, end, \surd \rangle.$$

This trace satisfies the policy ϕ , but is not a trace of any static enforcement process: A static enforcement process would have to *always* permit any event from the trace i_t , independent of past events. For example, it would permit the trace $\langle start, Get.Claire, Deliver.Dave \rangle$. However, this trace does not satisfy ϕ , because of the BoD constraint *bod*.

Therefore, we have shown by example how a dynamic enforcement process can allow more traces than any static enforcement process. After any unterminated trace i , it can authorize the following task-execution events: $\{t.u \mid \exists V \subseteq \mathcal{T} \times \mathcal{U} : V \models (\phi, T, \{t.u\}, i)\}$. We now formally construct an approximative dynamic enforcement process using this method, and prove its correctness:

Definition 5.6. $D_{\phi, T} = E_{\phi, T}(\langle \rangle)$, where

$$\begin{aligned} E_{\phi, T}(i) &= (t.u) : \{t.u \mid \exists V \subseteq \mathcal{T} \times \mathcal{U} : V \models (\phi, T, \{t.u\}, i)\} \rightarrow E_{\phi, T}(i \hat{\langle t.u \rangle}) \\ &\quad \square o : \mathcal{O} \rightarrow E_{\phi, T}(i \hat{\langle o \rangle}) \\ &\quad \square SKIP, \\ &\quad i \in \Sigma^*. \end{aligned}$$

Theorem 2. *Let W be a workflow process. Let T be the set of tasks that W engages in. Let ϕ be an authorization policy. If $\exists V : V \models (\phi, T, \emptyset, \langle \rangle)$, then $D_{\phi, W}$ is an enforcement process for ϕ on W .*

For proving Theorem 2 we first prove some lemmas. We prove statements about all traces of a process P using structural induction (Induction Principle 1).

Lemma 5.4 (Internal state). $\forall i \in \mathsf{T}(E_{\phi, T}(\langle \rangle)) \cap i \in \Sigma^* : E_{\phi, T}(\langle \rangle) \setminus i = E_{\phi, T}(i)$

Proof. We show that $\forall i \in \mathsf{T}(E_{\phi, T}(\langle \rangle)) : i \in \Sigma^* \Rightarrow E_{\phi, T}(\langle \rangle) \setminus i = E_{\phi, T}(i)$. Proof by structural induction (Induction Principle 1). Let $i \in \mathsf{T}(E_{\phi, T}(\langle \rangle))$.

Induction base, $i = \langle \rangle$: Trivially $E_{\phi, T}(\langle \rangle) \setminus \langle \rangle = E_{\phi, T}(\langle \rangle)$.

Induction step, $i \neq \langle \rangle$: *Induction hypothesis:* Assume $i \in \Sigma^* \Rightarrow E_{\phi, T}(\langle \rangle) \setminus i = E_{\phi, T}(i)$. Let $i \hat{\langle e \rangle} \in \mathsf{T}(E_{\phi, T}(\langle \rangle))$. Therefore $e \in \mathit{initials}(E_{\phi, T}(\langle \rangle) \setminus i)$. Assume $i \hat{\langle e \rangle} \in \Sigma^*$. Thus $i \in \Sigma^*$ and $e \in \Sigma$. Now with the induction hypothesis follows from $e \in \mathit{initials}(E_{\phi, T}(\langle \rangle) \setminus i)$ that $e \in \mathit{initials}(E_{\phi, T}(i))$. Therefore $e \in \mathcal{T} \times \mathcal{U} \vee e \in \mathcal{O}$.

Case $e \in \mathcal{T} \times \mathcal{U}$: Then $E_{\phi, T}(\langle \rangle) \setminus i \hat{\langle e \rangle} \stackrel{\text{induction hyp.}}{=} E_{\phi, T}(i) \setminus \langle e \rangle \stackrel{\text{rule 1}}{=} E_{\phi, T}(i \hat{\langle e \rangle})$.

Case $e \in \mathcal{O}$: Then $E_{\phi, T}(\langle \rangle) \setminus i \hat{\langle e \rangle} \stackrel{\text{induction hyp.}}{=} E_{\phi, T}(i) \setminus \langle e \rangle \stackrel{\text{rule 2}}{=} E_{\phi, T}(i \hat{\langle e \rangle})$. □

Lemma 5.5 (Trace refinement). *Given an authorization policy ϕ and a set of tasks T . Then $A_\phi \sqsubseteq_{\mathsf{T}} E_{\phi, T}(\langle \rangle)$.*

Proof. Let ϕ be an authorization policy and T a set of tasks. Proof by structural induction (Induction Principle 2). To be shown: $\forall i \in \Sigma^{*\surd} : i \in \mathsf{T}(E_{\phi, T}(\langle \rangle)) \Rightarrow i \in \mathsf{T}(A_\phi)$.

Induction base, $i = \langle \rangle$: By the traces model, $i \in \mathsf{T}(A_\phi)$.

Induction step, $i \neq \langle \rangle$: Let $i \in \Sigma^*$. *Induction hypothesis:* Assume $i \in \mathbb{T}(E_{\phi,T}(\langle \rangle)) \Rightarrow i \in \mathbb{T}(A_\phi)$. Let $e \in \Sigma^\vee$. We show that the hypothesis holds for $i^\wedge \langle e \rangle$: Assume $i^\wedge \langle e \rangle \in \mathbb{T}(E_{\phi,T}(\langle \rangle))$, so $i \in \mathbb{T}(E_{\phi,T}(\langle \rangle))$ and, by the induction hypothesis, $i \in \mathbb{T}(A_\phi)$. Also, $e \in \text{initials}(E_{\phi,T}(\langle \rangle) \setminus i)$. By Lemma 5.4, $e \in \text{initials}(E_{\phi,T}(i))$. Due to the definition of $E_{\phi,T}$ holds $e \in \mathcal{T} \times \mathcal{U} \vee e \in \mathcal{O} \vee e = \checkmark$. Case $e \in \mathcal{T} \times \mathcal{U}$: By rule 1 of the definition of $E_{\phi,T}$: $\exists V \subseteq \mathcal{T} \times \mathcal{U} : V \models (\phi, T, \{e\}, i)$. Let V_1 be such that $V_1 \subseteq \mathcal{T} \times \mathcal{U} \wedge V_1 \models (\phi, T, \{e\}, i)$. Therefore $e \in V_1$ and $V_1 \subseteq \chi(A_\phi \setminus i)$, so $e \in \text{initials}(A_\phi \setminus i)$. Case $e \in \mathcal{O}$: By Lemma 3.5, $\mathcal{O} \subseteq \text{initials}(A_\phi \setminus i)$. Case $e = \checkmark$: By the definition of A_ϕ holds $\checkmark \in \text{initials}(A_\phi)$. From this follows $e \in \text{initials}(A_\phi \setminus i)$ and therefore $i^\wedge \langle e \rangle \in \mathbb{T}(A_\phi)$. \square

We prove an essential invariant on $E_{\phi,T}(\langle \rangle)$, which says that in each state of $E_{\phi,T}(\langle \rangle)$, there exists a user-task assignment that satisfies (ϕ, T) , and which is a subset of the task execution events that $A_\phi \setminus i$ offers, *if* such an assignment exists in the initial state.

Lemma 5.6. *Given an authorization policy ϕ and a set of tasks T . Then the following holds:*

$$\exists V : V \models (\phi, T, \emptyset, \langle \rangle) \Rightarrow (\forall i \in \mathbb{T}(E_{\phi,T}(\langle \rangle)) \cap \Sigma^* : \exists V : V \models (\phi, T, \emptyset, i))$$

Proof. Let V be such that $V \models (\phi, T, \emptyset, \langle \rangle)$. Let $i \in \mathbb{T}(E_{\phi,T}(\langle \rangle)) \cap \Sigma^*$. By Lemma 5.5 we find that $i \in \mathbb{T}(A_\phi)$. Proof by structural induction (Induction Principle 1).

Induction base, $i = \langle \rangle$: By assumption, $V \models (\phi, T, \emptyset, \langle \rangle)$ holds.

Induction step, $i \neq \langle \rangle$: *Induction hypothesis:* Assume $\exists V : V \models (\phi, T, \emptyset, i)$. Let V_i be such that $V_i \models (\phi, T, \emptyset, i)$ holds. Let e be an event such that $i^\wedge \langle e \rangle \in \mathbb{T}(E_{\phi,T}(\langle \rangle)) \cap \Sigma^*$. Therefore $e \in \text{initials}(E_{\phi,T}(\langle \rangle) \setminus i)$. Now $\text{initials}(E_{\phi,T}(\langle \rangle) \setminus i) = \text{initials}(E_{\phi,T}(i))$, by Lemma 5.4. Since $E_{\phi,T}(i)$ does not engage in events from \mathcal{T} , holds $e \in \mathcal{T} \times \mathcal{U}$ or $e \in \mathcal{O}$.

Case $e \in \mathcal{T} \times \mathcal{U}$: By rule 1 of the definition of $E_{\phi,T}$: $\exists V \subseteq \mathcal{T} \times \mathcal{U} : V \models (\phi, T, \{e\}, i)$. Let V_e be such that $V_e \models (\phi, T, \{e\}, i)$. This implies that $V_e \models (\phi, T, \{e\}, \chi(A_\phi \setminus i))$ and therefore $e \in V_e$ and $V_e \subseteq \chi(A_\phi \setminus i)$. Now $V_e \models (\phi, T, \{e\}, \chi(A_\phi \setminus i))$ also implies that $\forall e \in (V_e \cup \mathcal{O}) : V_e \subseteq \chi(A_\phi \setminus i^\wedge \langle e \rangle)$. Therefore $V_e \models (\phi, T, \emptyset, i^\wedge \langle e \rangle)$.

Case $e \in \mathcal{O}$: $V \models (\phi, T, \emptyset, i)$ implies that $V \models (\phi, T, \emptyset, \chi(A_\phi \setminus i))$, from which $\forall e \in (V \cup \mathcal{O}) : V \subseteq \chi(A_\phi \setminus i^\wedge \langle e \rangle)$ follows. Therefore $V \models (\phi, T, \emptyset, i^\wedge \langle e \rangle)$. \square

Now we finally prove that our construction of a dynamic enforcement process is correct:

Proof of Theorem 2. Let W be a workflow process. Let T be the set of tasks that W engages in. Let ϕ be an authorization policy. Assume $\exists V \subseteq \chi(A_\phi) : A \models (\phi, T)$. To be shown, by Definition 5.2:

1. $A_\phi \sqsubseteq_{\mathbb{T}} D_{\phi,T}$
2. $(W \llbracket \pi^{-1} \rrbracket \parallel D_{\phi,T} \llbracket \pi \rrbracket =_{\mathbb{F}} W$

We show that $D_{\phi,T}$ is deterministic and that the following conditions hold, in order to be able to apply Lemma 5.3:

1. $D_{\phi,T}$ is deterministic
2. $A_\phi \sqsubseteq_{\mathbb{T}} D_{\phi,T}$
3. $\forall i \in \mathbb{T}(D_{\phi,T}) \cap \Sigma^* : (T \cup \mathcal{O})^\vee \subseteq \pi(\text{initials}(D_{\phi,T} \setminus i))$

Let T be the set of tasks that W engages in.

1. The process $D_{\phi,T}$ is deterministic because $D_{\phi,T} = E_{\phi,T}(\langle \rangle)$ in the definition of $E_{\phi,T}$, the guards of each rule are disjoint.
2. This holds by Lemma 5.5.

3. By Lemma 5.4 holds $\pi(\text{initials}(D_{\phi,T} \setminus i)) = \pi(\text{initials}(E_{\phi,T}(\langle \rangle) \setminus i)) = \pi(\text{initials}(E_{\phi,T}(i)))$. Let $i \in \mathsf{T}(D_{\phi,T}) \cap \Sigma^*$. From the definition of $E_{\phi,T}(i)$ and π follows $O^\vee \subseteq \pi(\text{initials}(D_{\phi,T} \setminus i))$. Furthermore, from Lemma 5.6 follows $\exists V : V \models (\phi, T, \emptyset, i)$. This implies that also $T \subseteq \pi(\text{initials}(D_{\phi,T} \setminus i))$.

The proposition follows from the application of Lemma 5.3. \square

5.3 Computing a Satisfying Task-User Assignment

We show how for a given authorization policy $\phi = (UT, S, B)$, a set of tasks T , and task-user assignments \bar{V} and \hat{V} , we can find a task-user assignment V such that $V \models (\phi, T, \bar{V}, \hat{V})$, using a SAT solver. For this, we refine in two steps the conditions of the definition of $V \models (\phi, T, \bar{V}, \hat{V})$, into a form that can be directly translated into a logical formula. We prove that from a solution to the formula we can construct a task-user assignment V such that $V \models (\phi, T, \bar{V}, \hat{V})$.

Definition 5.7. A relation $V \subseteq \mathcal{T} \times \mathcal{U}$ strictly satisfies an authorization policy $\phi = (UT, S, B)$, a set of tasks T , and task-user assignments \bar{V} and \hat{V} , written $V \models_{\text{strict}}(\phi, T, \bar{V}, \hat{V})$, if it fulfills the constraints

1. $\bar{V} \subseteq V \subseteq \hat{V} \cap UT^{-1}$
2. $\forall t_1 \in T, t_2 \in T : t_1 =_\phi t_2 \Rightarrow (\forall u, u' \in \mathcal{U} : u \in V\{t_1\} \wedge u' \in V\{t_2\} \Rightarrow u = u')$
3. $\forall t_1 \in T, t_2 \in T : t_1 \neq_\phi t_2 \Rightarrow (\forall u, u' \in \mathcal{U} : u \in V\{t_1\} \wedge u' \in V\{t_2\} \Rightarrow u \neq u')$
4. $\forall t \in T \exists u \in \mathcal{U} : t.u \in V$

Lemma 5.7. Let T be a set of tasks and $\phi = (UT, S, B)$ an authorization policy. Let V and \bar{V} be task-user assignments. Then the following holds:

$$V \models_{\text{strict}}(\phi, T, \bar{V}, \hat{V}) \Rightarrow V \models (\phi, T, \bar{V}, \hat{V})$$

Proof. Let $T \subseteq \mathcal{T}$ and $\phi = (UT, S, B)$. Let $V \subseteq \mathcal{T} \times \mathcal{U}$. Assume the four constraints as given in D5.7 hold. We prove the three conditions of D5.3.

Condition 1: This follows directly from constraint 1.

Condition 2: Let $i \in \mathsf{T}(A_\phi)$. Assume $V \subseteq \chi(A_\phi \setminus i)$. Let $e \in (V \cup \mathcal{O})$. Therefore $e \in V \vee e \in \mathcal{O}$.

1. Case $e \in \mathcal{O}$: By L3.5, $\chi(A_\phi \setminus i) \subseteq \chi(A_\phi \setminus i^\wedge \langle e \rangle)$ and therefore $V \subseteq \chi(A_\phi \setminus i^\wedge \langle e \rangle)$.
2. Case $e \notin \mathcal{O}$: Here $e \in V$. Then $e = t_1.u_1$ for some $t_1 \in \mathcal{T}$ and some $u_1 \in \mathcal{U}$. We need to show that $V \subseteq \chi(A_{UT} \setminus i^\wedge \langle e \rangle)$ and $\forall s \in S : V \subseteq \chi(A_s(\mathcal{U}, \mathcal{U}) \setminus i^\wedge \langle e \rangle)$ and $\forall b \in B : V \subseteq \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle e \rangle)$. Let $t_2.u_2 \in V$.
 - (a) $\chi(A_{UT} \setminus i^\wedge \langle e \rangle) = \chi(A_{UT}) = UT^{-1}$ and therefore $V \subseteq \chi(A_{UT} \setminus i^\wedge \langle e \rangle)$.
 - (b) Let $s \in S$.
 - i. Case $t_1 \neq_s t_2$: Here $t_1 \neq_\phi t_2$. By constraint 3, $u_1 \neq u_2$. By L3.2/3, $t_2.u_2 \in \chi(A_s(\mathcal{U}, \mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$.
 - ii. Case $\neg(t_1 \neq_s t_2)$: By L3.2/3, $t_2.u_2 \in \chi(A_s(\mathcal{U}, \mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$.
 - (c) Let $b \in B$.
 - i. Case $t_1 =_b t_2$: Here $t_1 =_\phi t_2$. Therefore by constraint 2, $u_1 = u_2$. By L3.3/3, $t_2.u_2 \subseteq \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$.
 - ii. Case $\neg(t_1 =_b t_2)$: By L3.3/3, $t_2.u_2 \subseteq \chi(A_b(\mathcal{U}) \setminus i^\wedge \langle t_1.u_1 \rangle)$.

Condition 3: This follows from constraint 4.

As conditions 1-3 are fulfilled, $V \models (\phi, T, \bar{V}, \hat{V})$ holds. \square

A relation $V \subseteq \mathcal{T} \times \mathcal{U}$ that strictly satisfies an authorization policy $\phi = (UT, S, B)$ and a set of tasks T can be found as follows using a SAT solver.

Lemma 5.8. *Let $\phi = (UT, S, B)$ be an authorization policy. Let T be a set of tasks. Let \bar{V} and \hat{V} be task-user assignments. We define the set of variables $A = \{a_{t,u} \mid t \in T, u \in \mathcal{U}\}$. Let $\alpha : A \rightarrow \{\top, \perp\}$, i.e. an assignment of each variable to a truth value. Furthermore, we define the relation $V_\alpha = \{(t, u) \mid t \in T, u \in \mathcal{U}, \alpha(a_{t,u}) = \top\}$.*

1. $\forall t \in T, u \in \mathcal{U} : \text{if } (t, u) \notin \hat{V} \cap UT^{-1} \text{ then } (\neg\alpha(a_{t,u}))$
2. $\forall t \in T, u \in \mathcal{U} : \text{if } (t, u) \in \bar{V} \text{ then } (\alpha(a_{t,u}))$
3. $\forall t_1 \in T, t_2 \in T : \text{if } t_1 =_\phi t_2 \text{ then } \forall u \in \mathcal{U} : (\neg\alpha(a_{t_1,u}) \vee \alpha(a_{t_2,u})) \wedge (\alpha(a_{t_1,u}) \vee \neg\alpha(a_{t_2,u}))$
4. $\forall t_1 \in T : \text{if } \exists t_2 \in T : t_1 =_\phi t_2 \text{ then } \forall u_1 \in \mathcal{U}, u_2 \in \mathcal{U} : \text{if } u_1 \neq u_2 \text{ then } (\neg\alpha(a_{t_1,u_1}) \vee \neg\alpha(a_{t_1,u_2}))$
5. $\forall t_1 \in T, t_2 \in T : \text{if } t_1 \neq_\phi t_2 \text{ then } \forall u \in \mathcal{U} : (\neg\alpha(a_{t_1,u}) \vee \neg\alpha(a_{t_2,u}))$
6. $\forall t \in T : (\bigvee_{u \in \mathcal{U}} \alpha(a_{t,u}))$

If the above conditions on α are fulfilled, then $V_\alpha|_{\text{strict}}(\phi, T, \bar{V}, \hat{V})$.

Note that the conditions for V_α in Lemma 5.8 are such that for a given ϕ, T and \mathcal{U} , they can be expressed as a boolean formula in Conjunctive Normal Form (CNF). We demonstrate this using an example:

Example 5.3. Let $T = \{t_1, t_2\}$, $U = \{u_1, u_2\}$, $\bar{V} = \emptyset$, $\hat{V} = UT^{-1} = T \times U$, $S = \emptyset$, $B = (\{t_1, t_2\}, \emptyset)$, $\phi = (UT, S, B)$. For a satisfying assignment α for the CNF formula,

$$\begin{aligned} & (\neg a_{t_1, u_1} \vee a_{t_2, u_1}) \wedge (a_{t_1, u_1} \vee \neg a_{t_2, u_1}) \wedge (\neg a_{t_1, u_2} \vee a_{t_2, u_2}) \wedge (a_{t_1, u_2} \vee \neg a_{t_2, u_2}) && \text{[from condition 3]} \\ & \wedge (\neg a_{t_1, u_1} \vee \neg a_{t_1, u_2}) \wedge (\neg a_{t_2, u_1} \vee \neg a_{t_2, u_2}) && \text{[from condition 4]} \\ & \wedge (a_{t_1, u_1} \vee a_{t_1, u_2}) \wedge (a_{t_2, u_1} \vee a_{t_2, u_2}) && \text{[from condition 6]} \end{aligned}$$

for the relation $V_\alpha = \{(t, u) \mid t \in T, u \in \mathcal{U}, \alpha(a_{t,u}) = \top\}$ holds $V_\alpha|_{\text{strict}}(\phi, T, \bar{V}, \hat{V})$.

In general, the formula has a size of up to $2 \cdot |T| \cdot |\mathcal{U}|^2 + 6 \cdot |T|^2 \cdot |\mathcal{U}| + 2 \cdot |T| \cdot |\mathcal{U}|$ literals. We can find a satisfying assignment α using a SAT solver. If the SAT solver doesn't find a solution, we can conclude that none exists. If it finds a solution, we are sure that it satisfies the CNF formula, and therefore it satisfies the conditions of Lemma 5.8.

Proof of Lemma 5.8. Let $\phi = (UT, S, B)$ be an authorization policy. Let T be a set of tasks. Let \bar{V} and \hat{V} be task-user assignments. Let $a_{t,u} \in \{\top, \perp\}$ for each $t \in T, u \in \mathcal{U}$ be values that fulfill the five conditions of Lemma 5.8. Let $V_\alpha = \{(t, u) \mid t \in T, u \in \mathcal{U}, \alpha(a_{t,u}) = \top\}$. We prove the four conditions of Definition 5.7.

1. Let $(t, u) \in \bar{V}$. By condition 2, $(t, u) \in V_\alpha$ and therefore $\bar{V} \subseteq V_\alpha$.
Let $(t, u) \in V_\alpha$. By condition 1, $(t, u) \in \hat{V} \cap UT^{-1}$ and $V_\alpha \subseteq \hat{V} \cap UT^{-1}$.
2. Let $t_1 \in T, t_2 \in T$. Assume $t_1 =_\phi t_2$. Let $u_1 \in V_\alpha\{t_1\}$, $u_2 \in V_\alpha\{t_2\}$. By condition 3, $u_1 \in V_\alpha\{t_2\}$. Proof by contradiction: Assume $u_1 \neq u_2$. Therefore by condition 4, either $u \notin V_\alpha\{t_1\}$ or $u_2 \notin V_\alpha\{t_2\}$, which contradicts our assumptions about u_1 and u_2 . Therefore $u_1 = u_2$.
3. Let $t_1 \in T, t_2 \in T$. Assume $t_1 \neq_\phi t_2$. Let $u_1 \in V_\alpha\{t_1\}$, $u_2 \in V_\alpha\{t_2\}$. Then $\alpha(a_{t_1, u_1}) = \top$. Therefore, by condition 5, $\alpha(a_{t_2, u_2})$ must be \perp . Therefore $u_1 \neq u_2$.
4. Let $t \in T$. Proof by contradiction. Assume $\neg \exists u \in \mathcal{U} : (t, u) \in V_\alpha$. Then $\forall u \in \mathcal{U} : \alpha(a_{t,u}) = \perp$. This contradicts condition 6.

□

5.4 Evaluation

For a policy ϕ and a workflow W , we showed how to construct an enforcement process *if* a task-user assignment V exists such that $V \models (\phi, T)$ holds, for the tasks T in W . The enforcement processes constructed with this approach are independent of the control flow of W , and only depend on the the policy ϕ and the set of tasks T . As a consequence of this, for some workflows our constructions are not possible, because no V exists such that $V \models (\phi, T)$ holds. However, for such a workflow some enforcement process for ϕ may still exist, which becomes apparent when one takes the control flow of the workflow into account.

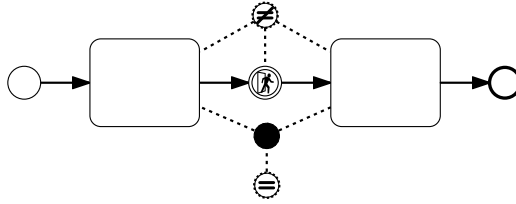


Figure 5.3: Workflow where our constructions fail

Example 5.4. Take the workflow depicted in Figure 5.3. The SoD constraint states that both tasks have to be executed by different users, whereas the BoD constraint states that they have to be executed by the same user. Therefore no satisfying $V \subseteq \mathcal{T} \times \mathcal{U}$ exists. However, if one takes into account the control flow of the workflow, it becomes clear that the SoD constraint does not have any effect, since it is released before it could constrain any task execution.

In future work, one could adapt the definition of $V \models (\phi, T, \bar{V}, i)$, so that it takes into account the actual set of traces of the workflow. Model-checking methods, for example using Roscoe’s work on model-checking CSP [Ros94], could be used for analyzing the traces. We expect that the lemmas and proofs that we provide in this work can be useful as a template for proving the correctness of a modified dynamic approximative enforcement process.

Note that we did not evaluate the performance of this approach. It is clear that when naively implemented, our approximate dynamic enforcement process needs to solve an NP-complete problem for every authorization decision. However, for a given workflow and policy these problems are all very similar. Therefore, it is likely that some assignments will be solutions for more than one problem. Solutions should therefore be cached. It can be checked in polynomial time whether a cached solution solves the problem at hand. Therefore, we expect that many of the problems can be solved efficiently. We propose a thorough performance analysis as future work.

Chapter 6

Tool Support

6.1 Purpose

In Chapter 4 we extended BPMN 2.0 to support our constraint model. In order to show the applicability of this approach, we implemented the extension in a BPMN 2.0 modeling tool. In future work, this implementation can be used for further validation, for example by conducting a case study. Another purpose of the implementation is to illustrate our enforcement processes with actual workflow models.

6.2 Choice of Implementation

We implemented our extension in the web-based modeling system *Oryx* [DOW08]. Oryx was chosen for four reasons:

- It is **open-source**: This means it is possible to extend or modify it in any way that is necessary.
- It is **mature**: The architecture and most of the code serve as a basis for the commercial modeling tool and company *Signavio* [Sig], as well as for the open-source workflow engine *Activiti* [Act].
- It is **well-documented**: Oryx has been developed in the form of several Bachelor theses, which provide a detailed description of the concepts and their implementations.
- It is **web-based**: Web-based tools can be used without installing any software. This makes it convenient for demonstration purposes.

6.3 Oryx Architecture

Oryx uses a client-server architecture based on standard Web technologies. The server part uses a Java EE Web Server (Tomcat Version 6 [Tom]), with a PostgreSQL [Pos] database backend. The server component is mostly implemented in Java, but also uses stored procedures in the database written in Python [Foub], and XSLT [Wor99] to carry out XML data transformations. The client part relies heavily on JavaScript [Foua] and asynchronous requests to the server (commonly called “Ajax”), as well as SVG [Wor11] for the graphical representation of models, in addition to the basic Web technologies. JSON [JSO], a subset of JavaScript for serializing data structures, is used by Oryx as its main format for configuration and for transmitting data between the client and the server. The client component of Oryx has been developed to run in Mozilla Firefox [Fir], but future versions of other browsers may as well be supported, if their support for the aforementioned standards improves.

6.4 Oryx Editor Principles

Oryx is a tool for editing *diagrams*. A diagram is a hierarchical structure of *shapes*: Each shape can have several *child shapes*. The root of the structure is the *canvas* shape. Except for the root, each shape has exactly one *parent shape*. A shape is associated with a *stencil*, which describes the type of the shape. A modeling language implementation for Oryx defines a stencil for each element of the language. The stencil defines a graphical representation and a set of properties. A *stencil set* describes how shapes can be arranged in the diagram, depending on their associated stencils. Shapes are ordered by their *z-order*. The z-order determines the order in which the shapes are painted when the diagram is displayed. A shape is painted on top of another shape if its z-order is lower.

6.5 Oryx Extension Mechanisms

Oryx is implemented as a small core and a rich set of extension plugins. As such, it is built to be extensible. New modeling languages can be added to Oryx by creating new stencil sets. A stencil set consists of a set of SVG graphics with Oryx-specific annotations, a set of PNG [PNG] icons, and a JSON configuration file. The configuration file describes the language in terms of which types of objects exist, what properties they have, how they may be connected or combined, and finally how they are represented graphically. It is also possible to add an extension to an existing stencil set.

Two kinds of plugins can be added to Oryx:

- *Client plugins* can add new elements to the user interface, e.g. a button, as well as any additional client-side processing, which includes interaction with the server. They are implemented in JavaScript as classes that inherit from the abstract plugin class provided by Oryx. Plugins can offer functionality to other plugins. This works through a publish-subscribe mechanism: A plugin can define *events*, to which any plugin can *register*. Any plugin can *raise* any event. When an event is raised, all plugins which have registered to the event get notified. They also receive the data that has been submitted with the event. Events are the only direct mechanism for communication between plugins.
- *Server plugins* extend the server. They are Java web applications based on Servlets [Ser]. Servlets are Java classes that implement the interface `javax.servlet.Servlet`. They can be bundled into *web application archives* and deployed in a web server which complies with the Java Servlet specification. When deployed in a web server, a servlet gets assigned a URL pattern, and is called by the web server to handle any requests that match this pattern. More specifically, the web server calls the “doGet” or “doPost” method on the servlet, depending on the type of the HTTP request.

6.6 Our Extension

Our extension to Oryx consists of three client-side and three server-side components. The server-side components are:

- **Assign**, a servlet that offers a method for finding a relation $V \subseteq \mathcal{T} \times \mathcal{U}$ such that $V | (\phi, T, \bar{V}, \hat{V})$, for a policy ϕ , a set of tasks T and a minimal and a maximal assignment \bar{V} and \hat{V} . However, instead of expecting $\phi, T, \bar{V}, \hat{V}$ as its input, it expects a BPMN 2.0 diagram with BPMNAuth annotations (as introduced in Chapter 4), and a task-user relation UT . It extracts the set of tasks T , as well as the set of BoD constraints B and the set of SoD constraints S from the diagram (as described in Section 4.1.5). This servlet uses the SAT solver SAT4J [Ber] for finding the relation V .

- **Check**, a servlet that offers a method for producing information about a BPMN 2.0 diagram with BPMNAuth annotations: It checks that the constraints are correctly defined and that they contain no contradictions. Additionally, if it is also provided with a task-user relation UT , it computes an assignment V (using the SAT solver SAT4J) such that $V \models (\phi, T)$, where $\phi = (UT, S, B)$, and S, B, T are extracted from the diagram. The result is returned in the form of a function from a diagram element to a message. A message can be a textual error message or a list of users.

The purpose of this functionality is to help the user of the editor locate problems in the model. Simply checking whether a satisfying assignment exists is not very helpful if the result is simply “No”. This servlet returns more detailed information about problems. For example, if there is a conflict between constraints, it can indicate which constraints are involved in the conflict.

- **Store**, a component that enables the server to deliver task-user relations and lists of users to the client. Currently, it only supports delivering data from files that are deployed on the server as a part of the web application archive, and data that has been sent to the server by a client application. For an enterprise setting, this component could be extended to deliver to the modeling environment the enterprise’s list of users and their authorizations.

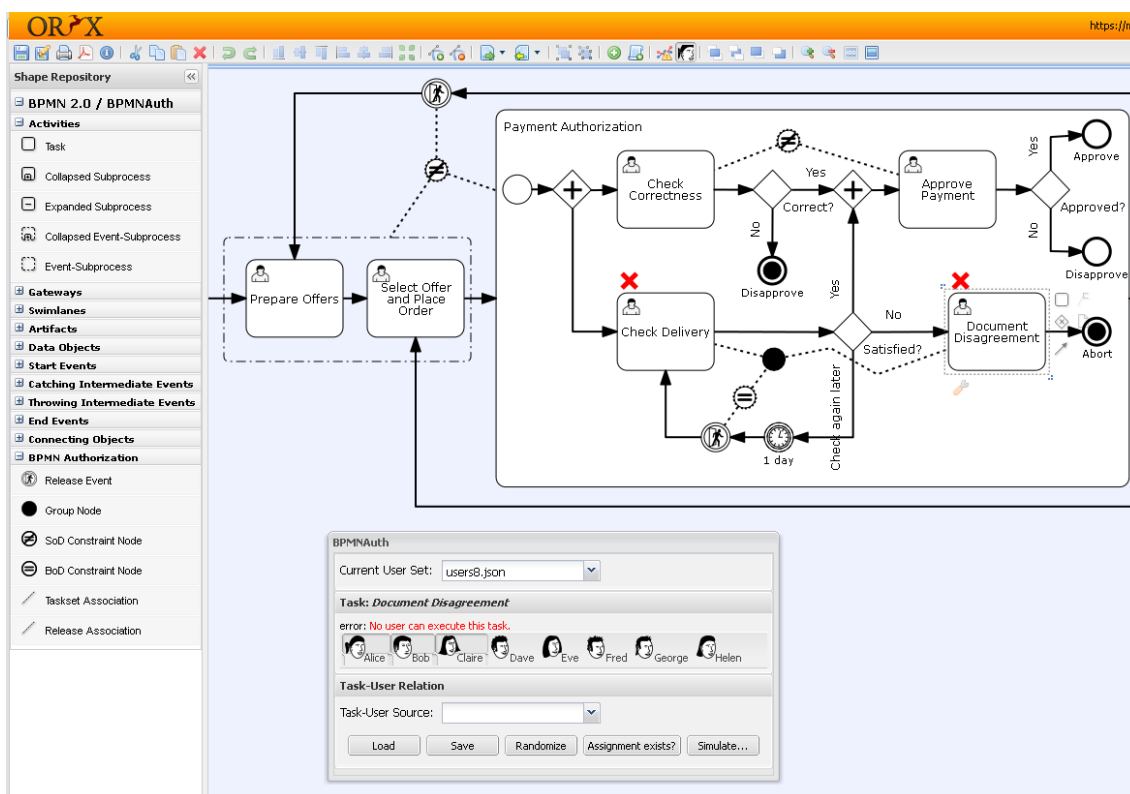


Figure 6.1: (Screenshot) A process diagram opened in Oryx, with BPMNAuth annotations, and with the task-user assignment dialog.

The client-side components integrate the BPMNAuth extension for BPMN 2.0 into the Oryx editor. They also provide an interface for the server components. The client-side components are the following:

- The **BPMNAuth** stencil set, which extends the BPMN 2.0 stencil set of Oryx with the elements of our extension for BPMN 2.0.

- **BPMNAuth**, an Oryx client plugin that is an interface to the **Check** and **Assign** servlets. Through events, it offers to the other plugins the functionality to send the currently active workflow to the **Check** servlet and display the returned messages as symbols in the diagram. It also offers sending the currently active workflow to the **Assign** servlet, along with optional parameters, and pass the returned task-user relation to the calling plugin. Furthermore, it changes in two ways how the editor handles group shapes:
 - Reordering: Originally, group shapes which are placed over existing shapes “cover” the existing shapes: Clicking on one of the existing shapes selects the group shape. Our plugin changes this behavior: Upon selection, the group shape is moved to the back of the z-order, i.e. it is placed behind all other elements. This makes those elements accessible which were previously covered.
 - Reassigning of parent: Originally, shapes that were placed inside a group shape were added as child shapes of the group shape. As a consequence, if the group shape was moved, the child shapes moved along. The plugin changes this behavior. Whenever a group shape is moved, it assigns the children of the group shape to the group shape’s parent. Therefore they no longer move along.

We feel that these changes make the group element more intuitive and easier to use.

- **BPMNAuthTaskUser**, an Oryx client plugin that offers a dialog for specifying a task-user assignment. It has a list of users that is currently loaded, and displays a toggle button for each user in this list. When the shape of a task t is selected in the diagram, $(u, t) \in UT$ holds if and only if the button for the user u is pressed. The plugin uses the **Store** component to offer loading of different lists of users, and storing and retrieving task-user assignments.
- **BPMNAuthSimulation**, an Oryx client plugin that is a partial implementation of a mechanism for simulating executions of the dynamic approximative enforcement process. It can determine which users can execute which tasks initially, using the **Assign** servlet. It is future work to add a simulation of the authorization process A_ϕ , which would allow simulating executions.

Chapter 7

Related Work

Basin, Burri and Karjoth [BBK11] introduce the concept of “release” to scope SoD and BoD constraints, and describe this approach using a formal model in CSP, and an extension to BPMN. Also, they define the notion of an enforcement process that ensures the absence of both constraint violations and obstructions. They formulate the existence of an enforcement process as a decision problem, which they show to be NP-hard, but decidable in the case that the workflow and set of users are finite. For this problem they provide an approximative solution, by describing an algorithm to construct an enforcement process, which may not provide a solution in all cases where one exists. Furthermore, they describe a variant of this algorithm which runs in polynomial time if the set of users is large and their static authorizations are well-distributed. Their work forms the basis of this thesis. We use their formal model and refine and implement their extension for BPMN. We provide a new construction of an enforcement process which is less restrictive.

Wolter, Miseldine and Meinel [WMM09, WM10] present an extension to BPMN for capturing authorization requirements. It supports SoD, BoD, and also more general forms of constraints. The constraints are modeled using artifacts, similar to our approach. However, constraints are always defined with the current process instance as the scope. Relaxing the constraint through a releasing mechanism is not considered. They also describe how to show absence of deadlocks and security property violations for authorization-constrained workflows using the model checker SPIN. This is a very powerful method that can accurately verify most properties. However, the approach fails for workflows which contain arbitrary cycles.

The constraint model used in the BBK11 framework is based on the model of Crampton [Cra05]. His notion of a “completion compliant reference monitor” corresponds to an enforcement process. He writes:

“In general, given a request by u to execute task t in a partially completed instance of a workflow, there are three questions a reference monitor could consider:

- Is u authorized to perform t ?
- Are all constraints in which t is [...], satisfied for this instance if u performs t ?
- Can the workflow complete if u performs t ?

[...] We say a reference monitor is [...] *completion compliant* if it guarantees that the answers to each of the three questions is yes.”

His algorithm for a completion compliant reference monitor corresponds to our dynamic approximative enforcement process. For example, the following observation also applies to our process:

“We now make the crucial observation that a partially completed instance of a workflow can be regarded as a workflow specification in which completed tasks are assigned to a single user, namely the user that executed the task.”

However, because he does not consider loops in workflows and no mechanism for releasing constraints, the following differences exist: Firstly, because of loops there might be more than

one user who have executed a particular task. Secondly, because of releasing not *all* completed task-executions are relevant for deciding whether an unterminated trace i can still be completed. Rather, it is the state of the authorization process A_ϕ after the trace i .

Nevertheless, Crampton's model is more general in the sense that for two tasks t_1 and t_2 , an arbitrary relation ρ on users can be used to constrain which task-executions are possible: A constraint (t_1, t_2, ρ) mandates that if $t_1.u_1$ and $t_2.u_2$ happen in a workflow instance, then $u_1 \rho u_2$ has to hold. We expect that this capability can be added as an extension to the [BBK11] model. However, it remains open whether this is useful in any practical applications.

Bertino, Ferrari and Atluri [BFA99] were the first to study and propose a method for enforcing security constraints such as SoD on workflows. Their method is also based on task-user assignments that satisfy the constraints. Their workflow model is however limited to sequences of tasks, with no consideration of loops and parallel executions.

Chapter 8

Conclusions and Future Work

We have shown how to extend the industry standard for modeling workflows, BPMN, with the authorization constraints introduced in the BBK11 paper. We have also described our extension of the modeling tool Oryx, which adds support for the BBK11 framework. Furthermore, we have described a new algorithm for enforcing constraints, which is more general than the one introduced in the paper. It achieves this by taking into account the release mechanism. We have also proved the correctness of this algorithm, and pointed out its relation to existing approaches.

The following are opportunities for future research:

- Based on our implementation in Oryx, a case study involving business process and security modeling practitioners from industry can be conducted, to validate the applicability of the constraint model in practice. For this, the client plugin for the simulation of workflow executions should be extended, so that it can be used to demonstrate the constraint semantics.
- The dynamic approximative enforcement process for enforcing policies on workflows can be modified to take into account the traces of the workflows, and therefore allow more of the traces which satisfy the policy.
- A thorough performance analysis of our dynamic approximative enforcement process should be conducted. The current implementation in Oryx can be adapted for performance testing.
- In the BBK11 framework, task executions are considered atomic. The implications of this should be investigated. For example, consider the case that a task t_1 is started by some user u_1 , but then aborted and instead completed by a user u_2 . If t_1 and another task t_2 are separated by an SoD constraint, the question arises whether u_1 should be authorized to execute t_2 .
- The constraint model can be adapted to support using arbitrary relations on users to specify constraints, as in [Cra05].
- The constraint model should be adapted so that the permissions which are now considered static can be changed during the execution of a workflow instance, like it is possible in practice.

In conclusion, we have provided three contributions that build on the approach from the BBK11 paper. We have focused on steps towards making the approach applicable in practice. This work enables the conduction of case studies for proving the practical relevance of the framework. Such studies may finally lead to the implementation of concepts from the framework within commercial BPM systems. We hope that this thesis can thereby contribute to advancing the practice of security modeling and enforcement on workflows.

Bibliography

- [Act] Activiti: Team. <http://www.activiti.org/team.html>, accessed 30.7.2011.
- [BBK11] David Basin, Samuel J. Burri, and Günter Karjoth. Obstruction-free Authorization Enforcement: Aligning Security and Business Objectives. *Proc. of the 24th IEEE Computer Security Foundations Symposium (CSF '11)*, 2011.
- [Ber] Daniel Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform. <http://www.sat4j.org/>, accessed 31.07.2011.
- [BFA99] 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):65–104, 1999.
- [BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 2005.
- [Cra05] J. Crampton. A Reference Monitor for Workflow Systems with Constrained Task Execution. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT '05)*, pages 38–47, 2005.
- [DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx - An Open Modeling Platform for the BPM Community. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 382–385. Springer Berlin / Heidelberg, 2008.
- [Exp09] Expert Group on E-Invoicing. Final Report of the Expert Group on e-Invoicing. http://ec.europa.eu/internal_market/consultations/docs/2009/e-invoicing/report_en.pdf, November 2009.
- [Fir] Mozilla Firefox Web Browser. <http://www.mozilla.com/firefox/>, accessed 30.07.2011.
- [Foua] Mozilla Foundation. JavaScript. <https://developer.mozilla.org/en/JavaScript>, accessed 30.07.2011.
- [Foub] Python Software Foundation. About Python. <http://www.python.org/about/>, accessed 30.07.2011.
- [FSG⁺01] 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):224–274, 2001.
- [HM04] E. Harold and W. Means. *XML in a Nutshell*. O'Reilly, 2004.
- [Ins05] IT Governance Institute. Control Objectives for Information and Related Technology (COBIT) 4.1, 2005.
- [JSO] Introducing JSON. <http://json.org/>, accessed 30.07.2011.

- [LP05] Christian Lahti and Roderick Peterson. *Sarbanes-Oxley Compliance Using COBIT and Open Source Tools*. Syngress, 2005.
- [MA11] Mark McDonald and Dave Aaron. Reimagining IT: The 2011 CIO Agenda. Technical report, Gartner, Inc., January 2011.
- [Obj10] Object Management Group. Diagram Definition (DD) 1.0 - Beta 1. <http://www.omg.org/spec/DD/1.0/Beta1/>, accessed 31.07.2011, 2010.
- [Obj11] Object Management Group (OMG). Business Process Model and Notation (BPMN), Version 2.0. OMG Standard, <http://www.omg.org/spec/BPMN/2.0/PDF>, January 2011.
- [PNG] Portable Network Graphics. <http://www.libpng.org/pub/png/>, accessed 30.07.2011.
- [Pos] About PostgreSQL. <http://www.postgresql.org/about/>, accessed 30.07.2011.
- [Ros94] A. W. Roscoe. *Model-Checking CSP*, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.
- [Ros05] A. W. Roscoe. *The Theory and Practice of Concurrency*. Pearson and Roscoe, <http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf>, Upper Saddle River, NJ, USA, 2005.
- [Ser] Java Servlet Technology. <http://www.oracle.com/technetwork/java/javaee/servlet/index.html>, accessed 30.07.2011.
- [Sig] About Signavio GmbH. <http://www.signavio.com/en/company/about-us.html>, accessed 30.07.2011.
- [SO002] Sarbanes-Oxley Act of 2002. <http://www.sec.gov/about/laws/soa2002.pdf>, January 2002.
- [Tom] Apache Tomcat 6.0 Documentation Index. <http://tomcat.apache.org/tomcat-6.0-doc/>, accessed 30.07.2011.
- [WG07] P. Y. H. Wong and J. Gibbons. A Process-Algebraic Approach to Workflow Specification and Refinement. In *In Proceedings of the 6th International Symposium on Software Composition (SC '07)*, pages 51–65, Braga, Portugal, March 2007.
- [WM10] Christian Wolter and Christoph Meinel. An Approach to Capture Authorisation Requirements in Business Processes. *Requirements Engineering*, 15:359–373, 2010. 10.1007/s00766-010-0103-y.
- [WMM09] Christian Wolter, Philip Miseldine, and Christoph Meinel. Verification of Business Process Entailment Constraints Using SPIN. In *Proceedings of the First International Symposium on Engineering Secure Software and Systems (ESSoS '09)*, pages 1–15, 2009.
- [Wor99] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. <http://www.w3.org/TR/xslt>, 1999.
- [Wor08] World Wide Web Consortium (W3C). XML 1.0 (Fifth Edition). <http://www.w3.org/TR/2008/REC-xml-20081126/>, accessed 29.07.2011, 2008.
- [Wor11] World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.1. <http://www.w3.org/TR/SVG11/>, accessed 30.07.2011, 2011.

Appendix A

Terminology

Table A.1 lists some of the terms that are used in more than one of the different formalisms, and the corresponding terms that we use in this document. For example, we use “workflow” to mean “business process”, in order to clearly set it apart from the term “process”, as used within CSP. However, we may use “event” both to refer to an event as defined in BPMN or CSP respectively. If it might not be clear from the context which of the two is meant, we use “BPMN event” or “CSP event”.

| Term | Domain | Description |
|-------------|--------|--|
| process | CSP | A discrete mathematical object, characterized as a set of traces or failures (by the denotational semantics) or as a transition system (by the operational semantics). |
| | BPMN | “A sequence or flow of Activities in an organization with the objective of carrying out work.” [Obj11] We call this a <i>workflow</i> to prevent confusion with CSP processes. |
| event | BPMN | A flow element that triggers some action or waits for a condition to occur. |
| | CSP | An element of a trace, or a label in the transition system of a process. |
| element | BPMN | A syntactical unit. Corresponds to a notational symbol, an abstract syntax class, and an XML element. |
| | XML | “Each XML document contains one or more <i>elements</i> , the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag.” [Wor08] |
| association | BPMN | A BPMN artifact that relates two BPMN elements. Expressed by a dotted line. |
| | UML | “A [UML] structured relationship that specifies that objects of one thing are connected to objects of another.” [BRJ05] |

Table A.1: Conflicting Terminology