PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Automatic Techniques for the Synthesis and Assisted Deployment of Security Policies in Workflow-based Applications

Daniel Ricardo dos Santos

Advisor:

Dr. Silvio Ranise

Fondazione Bruno Kessler (FBK)

Co-Advisors:

Dr. Luca Compagna and Dr. Serena Ponta

SAP Labs France

March 2017

To my family, who support me in every moment of my life; *amare et sapere vix deo conceditur.*

Abstract

Workflows specify a collection of tasks that must be executed under the responsibility or supervision of human users. Workflow management systems and workflow-driven applications need to enforce security policies in the form of access control, specifying which users can execute which tasks, and authorization constraints, such as Separation/Binding of Duty, further restricting the execution of tasks at run-time. Enforcing these policies is crucial to avoid frauds and malicious use, but it may lead to situations where a workflow instance cannot be completed without the violation of the policy. The Workflow Satisfiability Problem (WSP) asks whether there exists an assignment of users to tasks in a workflow such that every task is executed and the policy is not violated. The run-time version of this problem amounts to answering user requests to execute tasks positively if the policy is respected and the workflow instance is guaranteed to terminate. The WSP is inherently hard, but solutions to this problem have a practical application in reconciling business compliance (stating that workflow instances should follow the specified policies) and business continuity (stating that workflow instances should be deadlock-free). Related problems, such as finding execution scenarios that not only satisfy a workflow but also satisfy other properties (e.g., that a workflow instance is still satisfiable even in the absence of users), can be solved at deployment-time to help users design policies and reuse available workflow models.

The main contributions of this thesis are three:

1. *We present a technique to synthesize monitors capable of solving the run-time version of the WSP, i.e., capable of answering user requests to execute tasks in such a way that the policy is not violated and the workflow instance is guaranteed to terminate. The technique is extended to modular workflow specifications, using components and gluing assertions. This allows us to compose synthesized monitors, reuse workflow models, and synthesize monitors for large models.*

2. *We introduce and present techniques to solve a new class of problems called Scenario Finding Problems, i.e., finding execution scenarios that satisfy properties of interest to users. Solutions to these problems can assist customers during the deployment of reusable workflow models with custom authorization policies.*

3. *We implement the proposed techniques in two tools.* CERBERUS *integrates monitor synthesis, scenario finding, and run-time enforcement into workflow management systems.* AEGIS *recovers workflow models from web applications using process mining, synthesizes monitors, and invokes them at run-time by using a reverse proxy. An extensive experimental evaluation shows the practical applicability of the proposed approaches on realistic and synthetic (for scalability) problem instances.*

Keywords
Business Process, Workflow, Authorization, Workflow Satisfiability, Workflow Resiliency

Contents

List of Tables

List of Figures

Abbreviations

ARBAC Administrative RBAC

BoD Binding of Duty

BP Business Process

BPEL Business Process Execution Language

BPM Business Process Management

BPMN Business Process Model and Notation

EPC Event Process Chains

ERP Enterprise Resource Planning

FPT Fixed-Parameter Tractable

GRC Governance, Risk, and Compliance

GUI Graphical User Interface

LTL Linear Temporal Logic

MDP Markov Decision Process

MDW Moderate Discussion Workflow

OMG Object Management Group

RBAC Role-Based Access Control

RCP Resiliency Checking Problem

REST Representational State Transfer

SMT Satisfiability Modulo Theories

SOA	Service-Oriented Architecture
SoD	Separation of Duty
SoDA	Separation of Duties Algebra
TRW	Trip Request Workflow
WFMS	Workflow Management System
WSP	Workflow Satisfiability Problem
YAWL	Yet Another Workflow Language

Chapter 1

Introduction

A workflow specifies a collection of tasks, whose execution is initiated by humans or software agents executing on their behalf, and the constraints on the order of execution of those tasks. Workflows represent a repeatable and structured collection of tasks designed to achieve a desired goal. They are used to model Business Processes (BPs) that capture the activities that must be performed in a business setting to provide a service or product. Business Process Management (BPM) includes the identification, execution, monitoring, and improvement of business processes over time [158]. BPM encompasses tools such as Workflow Management Systems (WFMS), used to handle the execution of workflows modeled in standard languages, e.g., Business Process Model and Notation (BPMN); and Process Mining, used to discover business processes from event log data [144].

Workflow enactment services provide the run-time environment that controls and executes workflows. BPM was originally a business discipline focused on modeling rather than the automated enactment of processes. The notion of Service-Oriented Architecture (SOA) helps bridge the gap between modeled processes and executable artifacts by encapsulating functionality provided by applications within web services. SOA can be used to realize process enactment with processes implementing new services or using existing ones. Business Process Execution Language (BPEL) [115] is an executable language for specifying actions within business processes as web services. Processes in BPEL export and import information by using web service interfaces that can be invoked using standards such as HTTP, WSDL and UDDI [145].

More recent trends in SOA include the development of microservices to build systems that evolve incrementally and are continuously deployed, as well as the use of Representational State Transfer (REST) web services. These services allow requesting systems to access and manipulate textual representations of web resources using a predefined set of stateless operations, called an API. These trends have led to the development of a new generation of "agile" BPM tools that allow the direct execution of a subset of BPMN through the invocation of REST APIs (see, e.g., [90, 89]). On the infrastructure side, BPM is moving to the cloud due to advantages such as elasticity [135]. Workflow as a Service (WFaaS) [120] and Business Process as a Service (BPaaS) [153] platforms allow customers to outsource the modeling and execution of processes. Cloud-based run-time platforms take advantage of RESTful BPM to manage many process instances and provide scalability and dependability not supported by traditional BPM software [63]. One example of a cloud-only BPM tool is Amazon SWF[1], whereas tools such as jBPM[2] and Activiti[3] can be deployed on-premise or on cloud. The three preceding tools expose REST APIs to manage the execution of workflows.

The aforementioned different technologies show that there are several ways to implement BPM. Both functional and non-functional requirements need to be considered when implementing workflow-based applications [145]. Workflow patterns [148, 128, 127] can be used to elicit

[1]`http://aws.amazon.com/swf/`
[2]`http://www.jbpm.org/`
[3]`http://www.activiti.org/`

functional requirements, whereas architectural choices are mostly driven by non-functional requirements related to cost, performance, and security [145].

Security-related dependencies are specified in workflows as authorization policies and additional constraints on the execution of the various tasks. Authorization policies specify that, in an organization, a workflow task is executed by a user who should be entitled to do so; e.g., the teller of a bank may create a loan request whereas only a manager may accept it. Additional authorization constraints are usually imposed on task execution, such as Separation of Duty (SoD) or Binding of Duty (BoD) whereby two distinct users or the same user, respectively, must execute two tasks. In this thesis, following [5], we call "security-sensitive" this kind of workflows. Authorization policies and constraints are crucial to ensure the security of workflow systems and to avoid errors and frauds. A recent example of fraud that could be avoided with the proper enforcement of authorization constraints is the Société Génerale scandal, caused by shuffling transactions, that led to a loss of almost 5 billion euro [28, 59].

While the enforcement of authorization policies and constraints is fundamental for security [92], it may also lead to situations where a workflow instance cannot be completed because no task can be executed without violating either the authorization policy or the constraints. These deadlock situations emphasize the conflict between security and usability and, in the case of business processes, the conflict between business compliance and business continuity. Business compliance states that the business processes must follow the modeled workflows, respecting control-flow constraints, authorization policies and authorization constraints. Business continuity, on the other hand, states that the business must not stop even in adverse conditions, e.g., in the absence of authorized users.

These conflicts may be resolved by an administrator granting additional permissions to a user, which violates the original intended policy and therefore hurts compliance. An alternative is that an administrator can cancel the execution of a business process instance, which violates business continuity. An ideal solution is to avoid that any instance execution ever reaches a situation where this choice must be made.

The *Workflow Satisfiability Problem (WSP)* consists of checking if there exists an assignment of users to tasks such that a security-sensitive workflow successfully terminates while satisfying all authorization constraints. Such a problem has been studied in several papers; see, e.g., [14, 34, 154, 41, 36, 29, 30, 35, 42, 12]. The run-time version of the WSP consists of answering sequences of user requests at execution time and ensuring successful termination together with the satisfaction of authorization policies and constraints. This problem has received less attention and only an approximate solution is available [12, 10]. Besides the WSP, other related problems have been studied in the literature. *Workflow Resiliency* [154] amounts to checking if a workflow can still be satisfied even in the absence of a certain number of users, while *Workflow Feasibility* [85] concerns the question of whether there is a possible configuration of the authorization policy (considering, e.g., delegation or administrative policies) in which the workflow is satisfiable.

These problems have been studied for workflows running in a WFMS, which usually provides control-flow enforcement and basic authorization enforcement, although support for authorization constraints is rare. However, nowadays, web applications are one of the most widely used platforms to implement business processes and deliver services to customers. Web applications implementing workflows are called workflow-driven and both control-flow and authorization enforcement are often implemented as adhoc solutions for these applications.

In this thesis, we describe a new solution to the run-time WSP using a monitor synthesis technique. The idea underlying our approach can be summarized as follows. Model checking is a technique for determining whether a (formal) model of a system satisfies a given property. If the property is false in the model, model checkers typically produce a counterexample scenario, which is then used by developers to fix bugs in the design of the system. To solve the WSP, we use the capability of model checkers to return counterexamples as follows. We formally represent security-sensitive workflows as symbolic transition systems. A symbolic model checker is then asked to find a counterexample to the property that the system is not terminating. Indeed, the returned counterexample (if any) is precisely an execution scenario solving the WSP. Since we are interested in finding all execution scenarios, we modify the model checker in order to compute all counterexamples, not just one. We represent a set of counterexample scenarios by using a reachability graph, i.e., a directed graph whose edges are labeled by task-user pairs in which users are symbolically represented by variables and whose nodes are labeled by a symbolic representation (namely, a formula of first-order logic) of the set of states from which it is possible to reach a state in which the workflow successfully terminates. The graph allows us to compactly encode all possible interleavings of tasks in a workflow. From the set of formulae labeling the nodes in the reachability graph we derive a monitor capable of answering positively to user requests to execute tasks at run-time iff the user is authorized to do so by the policy, there is no violation of authorization constraints and the workflow can still terminate (i.e., the request is part of one of the scenarios computed in the reachability graph).

The crux of our approach is that the model of the security-sensitive workflow in input to the symbolic model checker only contains the constraints on the control-flow and the authorization constraints, while it abstracts away from the authorization policy. In this way, the reachability graphs computed by the model checker can then be refined with respect to an authorization policy associated to a particular deployment context.

This technique is reused and extended to solve related problems, e.g., finding execution scenarios at deployment-time that satisfy certain authorization policies or additional constraints. Monitor synthesis and run-time enforcement are applied to workflow management systems, implemented as a tool called Cerberus, and to workflow-driven web applications, implemented as a tool called Aegis. The technique is the first precise and efficient solution to the run-time WSP (as opposed to approximate solutions, e.g., [12] that is too restrictive). Experimental results indicate good performance, with requests answered in less than 1 second for workflows of up to 200 tasks in Cerberus and overheads under 100 ms for real-world applications in Aegis (details in Chapters 6 and 7).

The work reported in this thesis was done as part of the SECENTIS[4] project, a European Industrial Doctorate program[5] with the participation of Fondazione Bruno Kessler, SAP Labs France, and the University of Trento. Due to the nature of the project, much of the research done in this thesis was driven and influenced by industrial needs.

1.1 Objectives and research challenges

This thesis has three main goals: to develop a novel solution to the run-time WSP that can be applied in an industrial setting; to foster the reuse of workflow models by providing tools to assist customers in deploying these models; and to show how to integrate the developed solutions into workflow management systems and workflow-driven web applications.

To achieve these goals, we must answer the following non-trivial research questions (**Q**), each with an associated challenge (**C**).

[4]`www.secentis.eu`

[5]Funded by the European Union under grant 317387 (FP7-PEOPLE-2012-ITN).

Q1 How can we best specify security-sensitive workflow systems in order to allow automatic synthesis of run-time monitors?

C1 The WSP is known to be NP-hard already in the presence of one SoD constraint [154]. Although there are current solutions to the problem, they are either not focused on its run-time version (see, e.g., [78]) or not precise [12].

Q2 How can we scale monitor synthesis and allow the reuse of synthesized monitors?

C2 Synthesis techniques based on formal methods are known to suffer from the state-space explosion problem that severely limits their scalability [118]. The resulting artifacts from such techniques cannot be modularly composed in the interest of both reuse and scalability.

Q3 How can we ensure customers that they can reuse workflow models with the authorization policy in their organization and how to ensure this deployment is resilient?

C3 It is not obvious which problems a customer may face when trying to reuse a workflow model. The associated computational problems are at least as complex as the WSP.

Q4 How to integrate monitor synthesis and run-time enforcement with an industrial workflow management system?

C4 Support for modeling and enforcement of authorization constraints is not built into industrial workflow management systems. These constraints are often specified separately and handled by auditing software, whose main goal is to detect problems a posteriori.

Q5 How to extend the applicability of monitor synthesis and run-time enforcement to workflow-driven web applications?

C5 Web applications are usually not developed from a workflow specification. The absence of a model and the stateless nature of HTTP leads to bugs and vulnerabilities in control-flow and authorization enforcement for web applications.

1.2 Contributions

To answer the five questions above and overcome the associated challenges, this thesis makes the following main contributions.

1. The specification of security-sensitive workflow systems using transition systems and an automated technique to synthesize run-time monitors capable of ensuring the successful termination of workflows while enforcing authorization policies and SoD/BoD constraints, thus solving the run-time version of the WSP (Chapter 3). This contribution provides answers to **Q1** and **C1**.
2. A refined version of the same specification and synthesis technique that is focused on modularity and composability, allowing scalability and reuse of synthesized monitors (Chapter 4). This contribution provides answers to **Q2** and **C2**.
3. The definition and solution of Scenario Finding Problems. Each of these problems represents situations that a customer may want to know before deploying a reusable workflow model (Chapter 5). This contribution provides answers to **Q3** and **C3**.
4. The architecture, implementation, and evaluation of the integration of the monitor synthesis technique with a state-of-the-art workflow management system based on the SAP HANA[6] in-memory database (Chapter 6). This contribution provides answers to **Q4** and **C4**.
5. The use of process mining to obtain workflow models from web application execution traces that can be fed to the same synthesis tech-

[6]`http://hana.sap.com/`

nique to generate monitors capable of enforcing security policies in workflow-driven web applications (Chapter 7). This contribution provides answers to **Q5** and **C5**.

1.3 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 discusses the background required for the rest of this work and describes the main related work.

Part I (Fundamentals) contains Chapters 3 to 5, describing the main theoretical contributions. Chapter 3 presents the technique to automatically synthesize run-time monitors. Chapter 4 shows an extension of the technique for modular specifications of workflows, improving its scalability and allowing users to exploit the reuse of workflow specifications. Chapter 5 introduces and solves Scenario Finding Problems, whose solutions assist users during the deployment of workflows with specific authorization policies.

Part II (Applications) contains Chapters 6 and 7, describing two valuable applications of the solutions presented in the previous Chapters. Chapter 6 presents the implementation of a tool integrating our work into the SAP HANA Operational Process Intelligence platform for business process modeling and enactment. Chapter 7 discusses the use of the monitor synthesis technique for workflow-driven web applications.

Part III (Discussion) contains Chapters 8 and 9, discussing the impact and results of the thesis. Chapter 8 considers the industrial impact of this work. Chapter 9 concludes this thesis and presents future lines of research.

1.4 List of publications

Parts of this thesis were published as the following papers (in reverse chronological order).

[1] Chapter 7: L. Compagna, D.R. dos Santos, S.E. Ponta, and S. Ranise. Aegis: Automatic enforcement of security policies in workflow-driven web applications. In *Proc. of CODASPY*. ACM, 2017

[2] Chapter 4: D.R. dos Santos, S. Ranise, and S.E. Ponta. Modular Synthesis of Enforcement Mechanisms for the Workflow Satisfiability Problem: Scalability and Reusability. In *Proc. of SACMAT*. ACM, 2016

[3] Chapter 6: L. Compagna, D.R. dos Santos, S.E. Ponta, and S. Ranise. Cerberus: Automated Synthesis of Enforcement Mechanisms for Security-sensitive Business Processes. In *Proc. of TACAS*. Springer, 2016

[4] Chapter 5: D.R. dos Santos, S. Ranise, L. Compagna, and S.E. Ponta. Assisting the Deployment of Security-Sensitive Workflows by Finding Execution Scenarios. In *Proc. of DBSec*. Springer, 2015 **(Best Student Paper Award)**

[5] Chapter 3: C. Bertolissi, D.R. dos Santos, and S. Ranise. Automated Synthesis of Run-time Monitors to Enforce Authorization Policies in Business Processes. In *Proc. of ASIACCS*. ACM, 2015

[6] Chapter 8: S. Dashevskyi, D.R. dos Santos, F. Massacci, and A. Sabetta. TESTREX: a Testbed for Repeatable Exploits. In *Proc. of CSET*. USENIX, 2014

This work also resulted in the filing of two patents, which are currently under evaluation in the United States Patent and Trademark Office (USPTO):

[7] S.E. Ponta, L. Compagna, D.R. dos Santos, and S. Ranise. Secure and Compliant Execution of Processes. Filed in the USPTO on 13/04/2016

[8] A. Sabetta, L. Compagna, S.E. Ponta, S. Dashevskyi, D.R. dos Santos, and F. Massacci. Multi-context Exploit Test Management. Filed in the USPTO on 22/04/2015

Chapter 2

State of the art

A workflow specification spans at least three perspectives: control-flow, data-flow, and authorization (also called the resource perspective) [148]. Control-flow constrains the execution order of the tasks (e.g., sequential, parallel, or alternative execution); the data-flow defines the various data objects consumed or produced by these tasks; and the authorization specifies the organizational actors responsible for the execution of the tasks in the form of authorization policies and constraints. These three dimensions are interconnected, as each one of them influences the others. The set of behaviors (i.e., possible executions of the workflow) allowed by the control-flow is further constrained by conditions on the data, as well as by user assignments and constraints in the authorization perspective.

Consider a simple Loan Origination Process with fours tasks: *Request Loan* ($t1$), *Evaluate Internal Credit Rating* ($t2$), *Evaluate External Credit Rating* ($t3$), and *Approve Loan* ($t4$). If task $t1$ has to be executed first, followed by $t2$ and $t3$ (in any order), followed by $t4$, then the behaviors $t1, t2, t3, t4$ and $t1, t3, t2, t4$ are allowed, whereas, e.g., $t1, t4, t3, t2$ is not (where $t1, \dots, tn$ represents a sequence of n tasks executed in order, i.e., t_{i+i} is executed after t_i). Now imagine that $t3$ is only executed for loans of more than 10k Euro, then behavior $t1, t2, t4$ becomes allowed, but only for some instances (those where the data object "loan amount" is less than 10k). If the organization running this workflow has two users $u1$ and $u2$ and there is a SoD constraint between $t1$ and $t4$ (so that the same user cannot request and approve a loan), then any behavior containing, e.g., $t1(u1)$ and $t4(u1)$ is not allowed (where $t(u)$ means that user u executes task t).

Given the conflicting goals of business compliance and business continuity presented in Chapter 1, finding good (or even optimal) trade-offs has been a topic of research in the business process management and security communities. The problems raised by these opposing views are further complicated by the interplay between the three perspectives (control-flow, data-flow, and authorization) introduced above. To better understand the problems and solutions that have been identified in the literature, we present an overview of the state of the art in four related fields, namely workflow modeling (Section 2.1), workflow satisfiability (Section 2.2), workflow resiliency (Section 2.3), and workflow-driven web applications (Section 2.4).

Notice that a common practice in the analysis of workflow satisfiability and resiliency is to abstract away from parts of a workflow specification. For instance, no related work takes into account the data-flow (some completely disregard it, e.g. [35], and some model it with non-deterministic decisions, e.g., [12]). It is also usual practice to limit the allowed control-flow constructs and supported authorization constraints.

2.1 Workflow modeling

There are many ways to model workflows, e.g., Petri nets [111], Event Process Chains (EPC), and Yet Another Workflow Language (YAWL) [158]. BPMN is a *de facto* standard in workflow modeling. It is maintained by the Object Management Group (OMG) [116] and is currently the language of choice in most BPM applications. BPMN provides a graphical notation for modeling business processes with the goal of supporting BPM for technical and business users, since it is intuitive and can be used to represent complex process semantics.

BPMN models are diagrams constructed from a set of graphical elements. Most realistic use cases are constrained to a subset of the elements found in the language. Muehlen and Recker [110] observed the frequency of elements in real-world scenarios. The most widely used (more than 50% of models) are Sequence Flow, Task, End Event, Start Event, Pool, and Exclusive Gateway; followed by Start Message, Text Annotation, Message Flow, Parallel Fork/Join, Lanes and Gateway (used in around 25% of models). We now describe the workflow modeling elements that are used throughout this thesis.

Tasks are depicted as rounded boxes and represent any kind of work that must be performed in a workflow. They are categorized based on the agent who performs them. Human tasks are performed by human users, whereas system tasks are performed automatically by the WFMS (e.g., stored procedures or invocation of external applications).

Sub-processes are depicted as rounded boxes with a "+" sign at the bottom. They represent special activities whose internal details are abstracted in a high-level view.

Events are depicted as circles and their occurrence affects the flow of a process. We use only Start and End events, which mark the beginning and the end of the execution of a process, respectively.

Sequence Flows are depicted as solid arrows and constrain the order of execution of activities. Arrows connect a *source*, which must be executed first, to a *target*, which must be executed afterwards.

Gateways are depicted as diamonds and control the divergence and convergence of Sequence Flows in a process, determining branching, forking, merging, and joining of paths. We use two kinds of gateways. **Parallel** gateways create and synchronize parallel flows, where the activities may be performed in any possible ordering. **Exclusive** gateways create and merge alternative flows. Only one of the alternative flows is taken during process execution and only the activities inside that flow will be executed. Exclusive gateways have conditions associated to each of the paths that are evaluated at run-time.

Data Objects represent the data consumed or produced by an activity. They are associated to activities by data associations (depicted as dashed arrows), whose direction indicates if the object is an input or an output.

Security requirements and authorization constraints are not supported by BPMN. There have been proposals of BPMN extensions to model security elements, e.g., SecureBPMN [22] and SecBPMN [129], but they are not yet widely adopted. In our graphical depictions of workflows throughout the thesis, we adopt a variation of those proposals to model authorization constraints. We use dashed lines connecting pairs of tasks and labeled by icons representing each constraint (e.g., "$\neq$" for SoD and "$=$" for BoD).

2.2 Workflow satisfiability

To position our work with respect to the state of the art in workflow satisfiability, we must introduce some concepts (Section 2.2.1), then comment on the related approaches (Section 2.2.2) and finally compare them with our technique (Section 2.2.3).

2.2.1 Problem formulations

Different formulations of the WSP are concerned with three dimensions: control-flow models, supported authorization policies and constraints, and problem setting.

Control-flow

There are three basic categories of control-flow support: linear workflows, which only admit a sequential execution of tasks (e.g., [14]); partial orders, which also allow parallel executions (e.g., [35]); and others (e.g., CSP [77] and Petri nets [111]), which add support for conditional branches and loops (e.g., [12]).

It is known that a family of partial orders is needed to characterize one Petri net [84], which means that modeling the control-flow with Petri nets has the advantage of compactly representing a workflow that has to be specified as potentially many partial orders. It is also always possible to obtain a safe Petri net from a CSP process [159].

As described in [160], conditional execution can lead to execution paths of different lengths, which means that WSP solutions that try to assign users to every task in a workflow cannot be immediately applied. Yang et al. [164] defined several formulations of the WSP, considering different control-flow patterns, and showed that, in general, the problem is intractable.

Authorization

A plan $\pi : T \to U$, where T is the finite set of tasks in the workflow and U is a finite set of users, is an assignment of tasks to users representing a workflow execution where $(t, u) \in \pi$ means that user u takes the responsibility of executing task t. An authorization constraint $c \in C$ can be seen as a pair (T', Θ), where $T' \subseteq T$ is called the scope of c and Θ is a set of functions $\theta : T' \to U$ [35]. The functions in Θ specify the assignments of tasks to users that satisfy the constraint.

Instead of enumerating every function $\theta \in \Theta$, it is common to define Θ implicitly by using a specification device. Several classes of authorization constraints for workflows have been identified in the literature. They can all be used, with some ingenuity, to define the functions $\theta \in \Theta$, so they can be recast in the form (T', Θ) shown above [29].

Counting constraints are of the form (t_l, t_r, T'), where $1 \leq t_l \leq t_r \leq k$. A plan satisfies a counting constraint if a user performs either no tasks in T' or between t_l and t_r tasks. One example of counting constraint is $(1, 2, \{t1, t2, t3\})$, which is satisfied if a user $u1$ executes 0, 1 or 2 tasks among those in $\{t1, t2, t3\}$.

Entailment constraints are of the form (T_1, T_2, ρ), where $T_1 \cup T_2 = T'$ and $\rho \subseteq U \times U$. A plan satisfies an entailment constraint iff there exist $t_1 \in T_1$ and $t_2 \in T_2$ such that $(\pi(t_1), \pi(t_2)) \in \rho$.

Entailment constraints can be further subdivided in three types. In Type 1 constraints, both sets T_1 and T_2 are singletons. In Type 2 constraints, at least one of the sets must be a singleton, whereas in Type 3 there are no restrictions on the cardinality of sets.

Examples of Type 1, 2 and 3 constraints are $(\{t1\}, \{t2\}, \neq\})$, $(\{t1, t2\}, \{t3\}, \neq\})$, and $(\{t1, t2\}, \{t3, t4\}, \neq\})$, respectively. The first constraint is satisfied if a user $u1$ executes $t1$ and $u2$ executes $t2$ (because $u1 \neq u2$). The second and third constraints are satisfied if $u1$ executes $t1$ and $u2$ executes $t3$. Those are examples of SoD constraints, BoD constraints can be similarly defined by using $=$ instead of $\neq$.

A special class of Type 1 constraints are equivalence-based constraints, of the form $(t_1, t_2, \sim)$, where $\sim$ is an equivalence relation on U. A plan satisfies this kind of constraint if the user who executes t_1 and the user who executes t_2 belong to the same equivalence class, e.g., same role (or to different classes for $\not\sim$ constraints).

Two generalizations of the previous classes are user-independent constraints and class-independent constraints [35].

User-independent constraints c are those where given a plan π that satisfies c and any permutation $\phi : U \to U$, the plan $\pi' = \phi(\pi(s))$ also satisfies c [29]. I.e., user-independent constraints are those whose satisfaction does not depend on the individual identities of users. The SoD constraints presented so far are user-independent, whereas a constraint requiring a specific user to perform at least one task in a set is not user-independent [30].

Class-independent constraints are those whose satisfaction depends only on the equivalence classes that users belong to [35]. Formally, let c be a constraint, $\sim$ be an equivalence relation on U, $U^\sim$ be the set of equivalence classes induced by $\sim$, and $u^\sim \in U^\sim$ be the equivalence class containing u. Then, for any plan π, we can define a function $\pi^\sim : T \to U^\sim$ as $\pi^\sim(t) = (\pi(t))^\sim$. Finally, c is class-independent for $\sim$ if for any function θ, $\theta^\sim \in \Theta$ implies $\theta \in \Theta$, and for any permutation $\phi : U^\sim \to U^\sim$, $\theta^\sim \in \Theta^\sim$ implies $\phi \circ \theta^\sim \in \Theta^\sim$ [35].

One example of class-independent constraint is $(\{t1\}, \{t2\}, \sim)$, where the classes induced by $\sim$ corresponds to departments of a company. This constraint is satisfied if $u(t1) \sim u(t2)$, i.e., the user executing $t1$ and the user executing $t2$ are in the same department. Indeed, every equivalence constraint $(t_1, t_2, \sim)$ (or $(t_1, t_2, \not\sim)$) is class-independent and every user-independent constraint is class-independent with respect to the identity relation [35].

Other approaches to authorization constraint specification include Bertino et al.'s [14] constraint specification language and Li and Wang's Separation of Duties Algebra (SoDA) [95]. The first is based on rules built on pre-defined logic predicates. The resulting set of rules, called constraint base, is a stratified normal program. The second is an algebra for high-level policies that allows to express and formalize policies based on users' attributes and the number of users executing tasks. The policies are enforced by low-level mechanisms such as static and dynamic separation of duties in Role-Based Access Control (RBAC) [132].

Unlike for control-flow, it is not easy to classify authorization constraints in terms of expressiveness, partly because there are many different frameworks to express them. For instance, entailment constraints of Type 3 clearly include those of Types 1 and 2, but counting constraints can also be used to express some forms of SoD [160], so entailment and counting constraints are not disjoint (i.e., in some cases, it is possible to express the same set of behaviors using a counting constraint or an entailment one). Also, clearly user-independent and class-independent constraints subsume parts of the other classes, but it is not clear which parts.

Figure 2.1 shows an attempt to systematically classify some classes of authorization constraints for workflow systems presented in the literature. The Figure shows the sets *Ent.* of entailment constraints (the subsets of constraints of Types 1, 2, and 3 are not shown to keep the figure readable), *Count.* of counting constraints, *Eq.* of equivalence constraints, *CI* of class-independent constraints and *UI* of user-independent constraints. Naturally, $Eq. \subset Ent.$ and $CI. \subset Ent.$, since an equivalence relation is an instance of a binary relation. The facts $UI \subset CI$ and $Eq. \subset CI$ were shown by Crampton et al. [35].

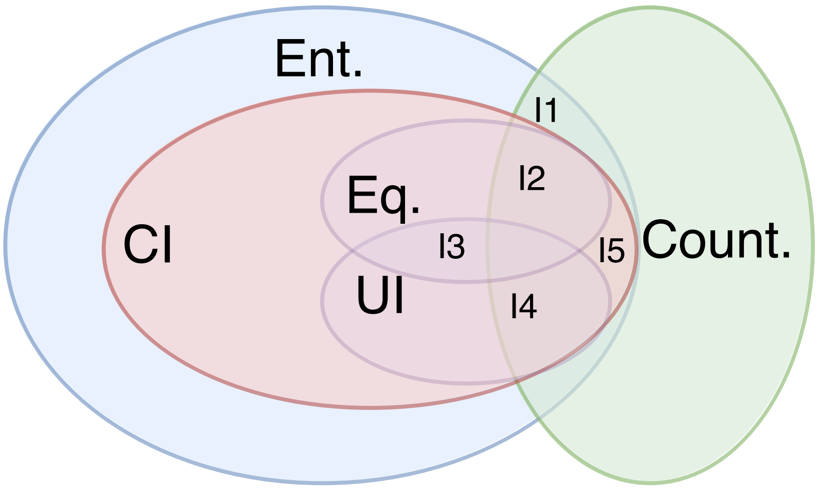


Figure 2.1: Constraint classes

The Figure also shows the following intersections: $I1 = Ent. \cap Count.$, $I2 = Eq. \cap Count.$, $I3 = Eq. \cap UI$, $I4 = Count. \cap UI$, $I5 = Count. \cap CI$. We can show that these intersections are non-empty by using SoD and BoD constraints as examples. $I1$ and $I2$ are non-empty because SoD and BoD can be specified using entailment: $(t1, t2, \neq)$ and $(t1, 2, =)$, resp.; counting: $(1, 1, \{t1, t2\})$ and $(2, 2, \{t1, t2\})$, resp.; or equivalence, since $=$ is an equivalence relation. $I3$, $I4$, and $I5$ are non-empty because both constraints are user-independent [30], which also makes them class-independent [35].

To the best of our knowledge, there has never been a comparison between the expressive power of other frameworks, e.g., SoDA and the constraint classes defined by Crampton et al. In any case, the most widely adopted kinds of constraints in practice are simple forms of SoD and BoD, and we will focus mostly on these.

Problem setting

Different solutions to the WSP consider at least two distinguishing characteristics: (i) is the order of the tasks considered? and (ii) is satisfiability checked at design-time (before the execution of any instance of the workflow) or at run-time (during execution), so that a satisfying execution is ensured?

The separation between ordered and unordered WSP was presented in [36]. The unordered WSP admits as solution a plan π assigning users to tasks in such a way that all tasks have an assigned user and all constraints are satisfied. The ordered version admits as solution a plan π with an execution schedule σ, which is a tuple $(t_1, \ldots, t_k)$ such that $1 \leq i < j \leq k, t_i \neq t_j$, i.e., the assignment must respect the ordering of tasks defined by the control-flow. The ordered and unordered versions of the WSP are only equivalent for the class of well-formed workflows [36], i.e., workflows with the following property: for all $t_i || t_j$ (tasks that can be executed in any order), $(t_i, t_j, \rho) \in C$ if and only if $(t_i, t_j, \tilde{\rho}) \in C$, where $\tilde{\rho}$ is defined as $\{(u, u') \in U \times U : (u', u) \in \rho\}$ and C is a set of entailment constraints.

A classification of WSP approaches in the design-time/run-time dimension was done in a recent survey [78]. Design-time techniques ensure the existence of at least one satisfying assignment, whereas run-time techniques enforce that a workflow instance follows a satisfying execution. As shown in Chapter 5, it is possible to use, at run-time, an algorithm that statically solves the WSP, but this is very inefficient, as it entails solving a new instance of the problem for each user request.

2.2.2 Related approaches

Table 2.1 presents a comparison of related approaches on the WSP (column 'Paper') in terms of control-flow models (column 'Control-flow'), authorization constraints (column 'Constraints'), and problem setting (ordered/unordered and execution time in columns 'Ordered' and 'Time', respectively), as described in the previous Section.

Initial works

The seminal work of Bertino et al. [14] described the specification and enforcement of authorization constraints in workflow management systems, presenting constraints as clauses in a logic program and an exponential algorithm for assigning users and roles to tasks without violating them, but considering only linear workflows. Crampton [34] extended these ideas by considering workflows as partial orders, defining Type 1 constraints, and developing an algorithm to determine whether there exists an assignment of users to tasks that satisfies the constraints.

Table 2.1: Comparison of related work

Paper	Control-flow	Constraints	Ordered	Time
[14]	Linear order	Constraint Specification Language	Yes	Design-time
[34]	P. order	Type 1	Yes	Design-time
[154]	P. order	Type 2	No	Design-time
[41]	P. order	Type 3 + Counting + Equivalence	No	Design-time
[36]	P. order	Type 3	No	Design-time
[29]	P. order	User-independent + Equivalence	No	Design-time
[30]	P. order	User-independent + Counting	No	Design-time
[35]	P. order	Class-independent	No	Design-time
[42]	P. order	Type 1	Yes	Design-time
[12]	CSP	SoD + BoD	Yes	Run-time
This work	1-safe PN	First-order logic	Yes	Run-time

Wang and Li [154] introduced the unordered version of the WSP, showed that it is NP-complete and that this intractability is inherent in authorization systems supporting simple constraints. They reduced the problem to SAT, which allows the use of off-the-shelf solvers, and showed that, with only equality and inequality relations (BoD/SoD), the WSP is Fixed-Parameter Tractable (FPT) in the number of tasks (since the number of tasks is typically smaller than the number of users)[1]. Wang and Li's FPT proof motivated many later works by Crampton et al., all considering the unordered version of the WSP for workflows specified as partial orders.

Crampton et al.

Crampton et al. [41] improved the complexity bounds for the WSP and showed that it remains FPT with counting and equivalence constraints. Later [36], they used the notion of constraint expressions (logical combinations of constraints) to support conditional workflows and Type 3 constraints by essentially splitting one instance of the problem into several instances, e.g., an instance of WSP for SoD/BoD constraints of Type 3 can be transformed into multiple instances of the WSP with SoD/BoD constraints of Type 1, and an instance of the WSP for a conditional workflow can be solved as many instances for parallel workflows. They also showed that the ordered version of the WSP is FPT for constraints of Type 1.

Cohen et al. [29] solved the WSP using techniques for the Constraint Satisfaction Problem, which allowed the authors to devise a general algo-

[1]FPT is a parameterized complexity class which contains the problems that can be solved in time $f(k) \cdot n^a$ for some computable function f, parameter k, and constant a [56]. Many hard problems become less complex if some natural parameter of the instance is bounded. An example is the satisfiability problem parameterized by the number of variables: a given formula of size n with k variables can be checked by brute force in time $O(2^k n)$. The WSP is FPT when parameterized by the number of tasks (i.e., $k = |T|$).

rithm that works for several families of constraints. Their solution builds executions incrementally, discarding partial executions that can never satisfy the constraints. The authors showed that their algorithm is optimal for user-independent constraints. Cohen et al. [30] demonstrated the practicality of the previously designed algorithm by adapting it to the class of user-independent counting constraints and showing its superiority when compared with the classical SAT reduction of the problem.

Crampton et al. [35] extended the notion of user-independent constraints to that of class-independent constraints, showed that the WSP remains FPT in this case and provided an algorithm to solve it. Crampton et al. [35] and Cohen et al. [30] experimentally compared the results of FPT algorithms against those of a SAT solver on workflows of up to 30 tasks and concluded that FPT algorithms are better because those based on the SAT solver run out of memory.

Crampton et al. [42] used model checking on an NP-complete fragment of Linear Temporal Logic (LTL), called LTL(F), to decide the satisfiability of workflow instances. The authors presented three encodings in LTL(F) that can compute a set of solutions. The slowest encoding considers the ordered WSP, while the other two consider unordered versions. They experimented with workflows of up to 220 tasks. The synthesis of monitors was left as future work.

Basin et al.

Basin et al. [11] considered the problem of choosing authorization policies that allow a successful workflow execution and an optimal balance between system protection and user empowerment. They treated the problem as an optimization problem (finding the cost-minimizing authorization policy that allows a successful workflow execution) and showed that, in the role-based case, it is NP-complete. They generalized the decision problem of whether a given authorization policy allows a successful workflow execution to the notion of an optimal authorization policy that satisfies this property.

In a following work [10], the authors used SoDA to enforce SoD constraints in a dynamic, service-oriented enterprise environment. They generalized SoDA's semantics to workflow traces that satisfy a term and refined it for control-flow and role-based authorizations. Their formalization, based on CSP, is the base for provisioning SoD as a Service, with an implementation using a workflow engine and a SoD enforcement monitor. Finally, in [12], they used CSP to model workflows in two levels: control-flow and task execution, allowing them to synthesize monitors that enforce at run-time obstruction-free, or satisfying, workflow executions.

2.2.3 Comparison

Overview of our approach

Our approach to solve the WSP, described in Chapter 3, works by synthesizing run-time monitors capable of ensuring that all executions terminate and authorization constraints in a workflow are satisfied. This is done in two steps.

We first construct a symbolic transition system S whose executions correspond to those of the security-sensitive workflow. Transitions in the system are composed of enabling conditions and effects. Control-flow and authorization constraints are encoded in the conditions of transitions in S by using predicates on state variables. Authorization constraints (such as SoD and BoD) are specified by using history functions $h_{ti}(u)$ that evolve with the execution of a workflow instance to keep track of which user u has executed which task t_i. Authorization policies are encoded by using functions $a_{ti}(u)$ that return *True* if the user u is authorized to execute task t_i according to the policy (they are left unspecified in the first step, so that we can accommodate different policies at run-time). Then, we use a symbolic model checker to explore all terminating executions of the workflow which satisfy both the control-flow and the authorization constraints. The model checker returns a symbolic representation R of the set of all reachable states encountered during the exploration of the terminating executions of S. We use first-order logic [58] to symbolically represent S and R. In the second step, we derive a Datalog [26] program M from the formulae R, representing the set of states reachable in the terminating executions of S and the policy P specifying which user can perform which task. The Datalog program M derived in this way is the monitor capable of guaranteeing that any request of a user to execute a task is permitted by P, satisfies the authorization constraints, and the workflow instance can terminate.

Splitting the problem in two steps allows us to pre-compute a significant part of the solution and synthesize monitors that are parametric with respect to the authorization policies. This is important because it fosters the reuse of workflow models across different organizations and is more efficient when there is a policy change inside the same organization.

In Chapter 4 we describe an extension of this technique based on a refinement of the transition systems used to specify security-sensitive workflows. The refined transition systems are associated to a suitable notion of interface, forming a so-called security-sensitive component. We then show how to synthesize monitors for components and how to combine these monitors in a principled way, by using gluing assertions that specify how the control-flow and authorization constraints are transferred from one component to another. The intuition is that monitors for components are computed by considering any possible values for the variables in their interfaces. The additional constraints in the gluing assertions define a subset of these values.

Since the synthesized monitors are modular, we can treat very large workflow instances as a composition of simpler parts, again fostering reuse and allowing the scalability of the synthesis technique.

Authorization constraints

Although in the remainder of this thesis we present only SoD and BoD constraints as examples (because the history functions used in the conditions of transitions are a natural representation of user-independent constraints), the use of first-order logic formulae allows us to support a variety of authorization constraints.

As introduced above, we derive a symbolic transition system S from a workflow W such that the transitions in S encode a condition and an effect of the execution of each task in W. Both conditions and effects are composed of a control-flow and an authorization part. We ignore the control-flow for now, as it reuses well-known symbolic encodings of Petri nets (see, e.g., [133]), and give a hint on how we encode authorization constraints (both are detailed in the next Chapters).

We illustrate the main ideas of our symbolic encoding by using a simple workflow W containing tasks $t1$ and $t2$ and a SoD constraint $(t1, t2, \neq)$. The constraint can be specified as a condition that must hold in every state in the execution of W:

$$\forall u. \neg(h_{t1}(u) \wedge h_{t2}(u)) \Leftrightarrow \forall u. (\neg h_{t1}(u) \vee \neg h_{t2}(u)) \,. \tag{2.1}$$

We can add this constraint to the conditions of every transition in S. For instance, transition $t1$ is

$$\exists z. CF_C \wedge a_{t1}(z) \wedge \forall u. (\neg h_{t1}(u) \vee \neg h_{t2}(u)) \rightarrow CF_E \wedge h_{t1}(z) := T \tag{2.2}$$

where CF_C and CF_E are the symbolic representations of conditions and effects, respectively, on control-flow. We can eliminate the universal quantifier by only considering finitely many instances of u (exactly those that are existentially quantified). For the formal result guaranteeing this, see [2]. From now on, for simplicity, we also omit the existential quantifier, which is implicit for the users in every transition. Then (2.2) becomes

$$CF_C \wedge a_{t1}(z) \wedge (\neg h_{t1}(z) \vee \neg h_{t2}(z)) \rightarrow CF_E \wedge h_{t1}(z) := T \tag{2.3}$$

which can be rewritten as two transitions by using simple logical transformations

$$CF_C \wedge a_{t1}(z) \wedge \neg h_{t1}(z) \rightarrow CF_E \wedge h_{t1}(z) := T \tag{2.4}$$
$$CF_C \wedge a_{t1}(z) \wedge \neg h_{t2}(z) \rightarrow CF_E \wedge h_{t1}(z) := T \tag{2.5}$$

It is not difficult to show that (2.4) is redundant. Intuitively, if $t1$ has never been executed, then $\forall u. \neg h_{t1}(u)$ trivially holds and the set of states generated by (2.4) is contained in the set of states generated by (2.5).

Transition $t2$ is similar, but substituting a_{t1} by a_{t2} and showing that the version with condition $\neg h_{t2}$ is redundant. It is possible to optimize the encoding provided that some additional constraints hold. For instance, if we consider a control-flow constraint that specifies that $t1$ must be executed before $t2$, we can eliminate $\neg h_{t2}$ from the condition of $t1$.

It is possible to support other authorization constraints by devising logical constraints such as (2.1). For entailment constraints (T_1, T_2, ρ), we can use

$$\forall u_1, u_2. \bigvee_{t_1 \in T_1} \bigvee_{t_2 \in T_2} h_{t1}(u_1) \wedge h_{t2}(u_2) \Rightarrow \rho(u_1, u_2) \tag{2.6}$$

when ρ is definable in a "relatively simple" theory of users, which satisfies the assumptions for the termination of fix-point computation (for instance, $\neq$ for SoD and $=$ for BoD). The formalization of this kind of theory is given in [2]. For counting constraints (t_l, t_r, T'), we can use

$$\forall u_{T'}. \left(\bigwedge_{t' \in T'} h_{t'}(u_{t'}) \Rightarrow \mathsf{AtMost}(U_{T'}, t_r) \wedge \mathsf{AtLeast}(U_{T'}, t_l) \right) \tag{2.7}$$

where $U_{T'} = \{u_{t'} | t' \in T'\}$.

Differences

The works by Basin et al. [10, 12] are the closest to ours, since they also consider the WSP at run-time and implement a monitor. However, the monitor in [10] only verifies if a trace of a workflow satisfies a SoDA term with respect to the past, being incapable of checking whether there is a future trace that can be concatenated in order to satisfy the workflow. On the other hand, the monitor in [12] enforces obstruction-freedom (which is equivalent to solving the WSP) but in an approximated way and may be too restrictive. The authors call their monitors Enforcement Processes and the problem of deciding the existence of such a monitor for a constrained workflow is called Enforcement Process Existence (EPE). Their naive solution to the EPE is double exponential in the number of users and constraints because it depends on checking failure-equivalence in CSP [126]. They present two approximate solution, one is exponential and one is polynomial. The approximations are based on solutions to the graph coloring problem [62] and are overly restrictive because they may return 'No' even if an enforcement process does exist for the constrained workflow taken as input (although they make no approximations in the other direction, i.e., if there does not exist an enforcement process, the procedures always return 'No'). They also implemented, as we do (Chapter 6), tool support for the specification and enforcement of constraints, but the evaluation was limited to a few workflows used as examples.

Ours is the first work to provide a precise enforcement for the run-time version of the WSP. As shown in Table 2.1, it is also one of only two to consider control-flows more complex than partial orders and among a few to consider the ordered version of the problem. Most existing approaches to solve the WSP [154, 34, 36, 41, 42, 29, 30, 35] model workflows as partially ordered sets, which does not allow for conditional execution branches. This limitation can be overcome by treating a workflow model with conditional branches as many deterministic workflows [36], however that means that one instance of the problem has to be solved in multiple steps. We take as input a BPMN model that is translated to a Petri net and later to a transition system, which allows us to support conditional workflows by using many goal formulas. Other, more complex, control-flow patterns can also be handled more naturally this way [148, 51].

2.3 Workflow resiliency

Li et al. [96] introduced the notion of resiliency policies for access control systems, i.e., policies that require the system to be resilient to the absence of users. They defined the Resiliency Checking Problem (RCP), which amounts to checking if an access control state satisfies a given resiliency policy. Wang and Li [154] then studied resiliency in workflow systems and its relation to the WSP, defining three levels of resiliency based on when the users are allowed to be absent and whether they are allowed to return. In *static* resiliency, a number of users may be absent before a workflow instance execution; in *decremental* resiliency, users may be absent before or during a workflow instance execution, but absent users do not become available again; and in *dynamic* resiliency, users may become absent and available again. They showed that checking static workflow resiliency is in NP, while checking decremental and dynamic resiliency is in PSPACE. The authors observed that there are other possible formulations of resiliency that can be of interest.

Mace et al. [101] defined *quantitative* workflow resiliency, in which a user wants to know how likely a workflow instance is to terminate given a user availability model. The authors solve the problem by finding optimal plans for Markov Decision Process (MDP). The same authors [104] showed that alternative executions may lead to different resiliency values for each path, and defined resiliency variance as a metric to indicate volatility, claiming that a higher variance increases the likelihood of workflow failure. User availability models were discussed in more details in [102], categorized into non-deterministic, probabilistic, and bounded, with several encodings for the PRISM probabilistic model checker[2]. The same group studied the impact of policy design (adding or removing authorization constraints) on workflow resiliency computation time [103]. They were able to compute sets of security constraints that can be added to a policy in order to reduce computation time while maintaining resiliency. The authors then developed WRAD [105], a tool for workflow resiliency analysis and design, which automatically encodes workflows into PRISM, evaluates their resiliency and computes optimal changes for security constraints to ensure a resiliency threshold.

Crampton et al. [38] studied the Bi-Objective WSP, which is the problem of minimizing two weight functions associated to a valid plan, one representing the violation of constraints and one representing the violation of the authorization policy. This problem has a set of incomparable solutions (a Pareto front), allowing the user to choose the most suitable. The authors also related this problem to workflow resiliency, claiming that Mace et al.'s translation to MDP is not necessary since the same metrics can be computed by constructing a graph where the nodes are partial valid plans, and the edges, connecting successive plans, are labeled with the probability of a user being available to execute the next task (checking every possible partial plan has exponential-time complexity). The Bi-Objective WSP is a generalization of the Valued WSP [37], which has as single objective minimizing the sum of both weights.

Crampton et al. [39] reduced the RCP to the WSP and showed how to

[2]`http://www.prismmodelchecker.org/`

solve it using an FPT algorithm for the WSP. The RCP differs from workflow resiliency by considering three parameters: s users, forming d teams of size t, such that all teams are authorized to access the resources in a policy P. In contrast, k-resiliency for workflows just considers k absent users and whether the remaining users can execute all tasks. However, the RCP is always static, whereas workflow resiliency can be static, decremental, or dynamic. The basic solution in the original paper about RCP [96] is to enumerate all subsets of s users and check for satisfiability (using a procedure for $s = 0$ as a black-box), but there is a pruning strategy based on the redundancy of some subsets, to have a more efficient solution. Crampton et al. [39] solved the same RCP problem, using the same pruning strategy and their FPT algorithm as the black-box to decide satisfiability (by translating the resources in P to a workflow). The authors mention that this basic reduction cannot be applied directly to decremental or dynamic workflow resiliency, but they point to their work on Valued WSP [37] as a possibility to do it, by using weights to represent the availability of users.

Khan and Fong [85] defined the problem of workflow feasibility, when there are rules to update the authorization relation. A workflow is feasible if there is at least one reachable access control configuration where the workflow is satisfiable. They studied feasibility in relationship-based access control.

2.3.1 Comparison

Our solution to workflow resiliency, described in Chapter 5, relies on refining the reachability graphs computed by the model checker with a given authorization policy. The refinement is performed by a depth-first search of the graph to prune those executions that do not satisfy the authorization policy used in the deployment context under consideration. This is combined with a (heuristic) method to generate subsets of users not containing k users (by adapting the pruning strategy from [96]) in order to find scenarios guaranteeing the termination of a workflow despite the absence of k users.

Our approach is inspired by the solution to the RCP in [39], with the difference that we solve the resiliency problem for workflows and invoke a synthesized monitor for solving the WSP. Compared to other works, our technique has the advantage of reusing the heaviest part of the computation (i.e., generating the reachability graph) and only refining it during deployment. The algorithm presented as a solution is also capable of finding execution scenarios that satisfy other constraints, such as a particular user executing a task or using a minimal number of users.

We only consider static workflow resiliency, but the solution could be adapted for quantitative resiliency by assigning weights representing availability to the edges in the reachability graph, as hinted at in [39].

2.4 Workflow-driven web applications

Web applications can be designed using workflow modeling and there are frameworks that allow their declarative description (e.g., Spring Web Flow [151]). Web applications can also be modeled using page-flows (a graph where the nodes are pages and the edges are links), control-flow graphs [99], and automata [74, 48]. However, model-driven development of web applications is not common practice. This highlights the need for model inference techniques, which can be based on static analysis [24, 72] or dynamic analysis (e.g., web crawling [156] or symbolic execution [73]). Some of the main challenges for all analysis techniques are dynamic web pages, and the precision of recovered models, often requiring human post-processing [25]. Process mining was used to capture workflow models of web applications to optimize user interaction [122].

There are many white-box approaches to the enforcement of authorization, control-flow, and data-flow integrity in web applications. These approaches mitigate so-called business or logic vulnerabilities, such as missing authorization checks [139, 163]; workflow and state violations [9, 33, 61]; forceful browsing [19]; and parameter tampering [18, 7]. A survey on how to secure web applications against business and logic vulnerabilities can be found in [45].

A black-box approach to block request forgery was described in [82, 81], but it ignores data-flow and authorization. Black-box enforcement of control-flow and data-flow integrity was done in Ghostrail [20] by dynamically replicating on the server-side valid user clicks, form entries, links, and parameters. Ghostrail does not consider authorization, and needs a fresh replica for each user session, which is not scalable.

Web application workflow models were used to detect anomalous user behavior using Hidden Markov Models [98] and to find logic vulnerabilities by capturing execution traces, identifying behavioral patterns, and generating test cases [119].

BLOCK [97] and InteGuard [161] use a reverse proxy as an external monitor for web applications, construct control-flow and data-flow policies using invariants detected from network traces, and rely on manual identification of critical requests. BLOCK also extracts invariants from session information in PHP applications. InteGuard is tailored for multi-party application integration, where most of the steps in a trace are automatic (not human tasks) and workflows must be executed from beginning to end in one shot. Neither tool enforces authorization policies nor constraints. FlowWatcher [112] uses a similar approach of proxy monitoring, but enforces only authorization policies specified in a domain specific language.

The enforcement of authorization constraints for collaborative web applications was studied in [64, 66, 65], where the authors considered applications for form and document editing.

2.4.1 Comparison

Aegis is the approach described in Chapter 7 to synthesize run-time monitors for workflow-driven web applications that are capable of automatically (i) enforcing security policies composed of combinations of control- and data-flow integrity constraints, authorization policies, and authorization constraints; and (ii) solving the run-time version of the WSP.

Aegis is based on the monitor synthesis technique from Chapter 3. To synthesize a monitor, Aegis first infers, using process mining [144], workflow models of the target application from a set of HTTP traces representing user actions. Traces must be manually edited to contain only actions that should be enforced by the monitor. Inferred models are Petri nets labeled with HTTP requests representing tasks and annotated with data-flow properties obtained by using heuristics based on differential analysis [161, 140]. These Petri nets can be refined by a human user who specifies authorization constraints and an authorization policy. A monitor is then generated from a model by invoking the synthesis tool. At run-time, a reverse proxy is used to (i) capture login actions to later establish the acting users, and (ii) capture incoming requests and query the monitor to determine whether to allow or deny the request. Aegis is completely black-box and can be used to add support for the enforcement of security-related properties or to mitigate logic vulnerabilities in web applications.

Aegis is similar to [97, 161] with respect to capturing execution traces and synthesizing policies that are enforced by a reverse proxy. It is also inspired by [64, 66, 65] in the enforcement of authorization constraints for collaborative scenarios. Nevertheless, our solution has some important differences. Aegis is the only to consider at the same time control-flow, data-flow, and authorization. It is also the first work to consider the WSP in the context of web applications.

Part I

Fundamentals

Chapter 3

Automatic synthesis of run-time enforcement monitors[1]

In this Chapter, we introduce our approach to synthesize run-time monitors for security-sensitive workflows. We first introduce the Trip Request Workflow (TRW) as an illustrative example that is used throughout the thesis.

Example 1 (Trip Request Workflow (TRW))**.** The workflow in Figure 3.1 is composed of five tasks—each one indicated by a box labeled by Request ($t1$), Car rental ($t2$), Hotel booking ($t3$), Flight reservation ($t4$), and Validation ($t5$)—whose execution is constrained as follows (cf. solid arrows and diamonds labeled with +): $t1$ must be executed first, then $t2$, $t3$ and $t4$ can be executed in any order, and when all have been performed, $t5$ can be executed, thereby terminating the workflow. Additionally, each task is executed under the responsibility of a user (indicated by the small icon inside the boxes corresponding to the various tasks) who has the right to execute it according to some authorization policy—not shown in Figure 3.1—and the five authorization constraints depicted as dashed lines labeled by the symbol $\neq$ for Separation of Duty (SoD). So, for example, the authorization constraint connecting the boxes of $t1$ and $t2$ requires the user executing $t2$

[1]Parts of this chapter were previously published in [15]

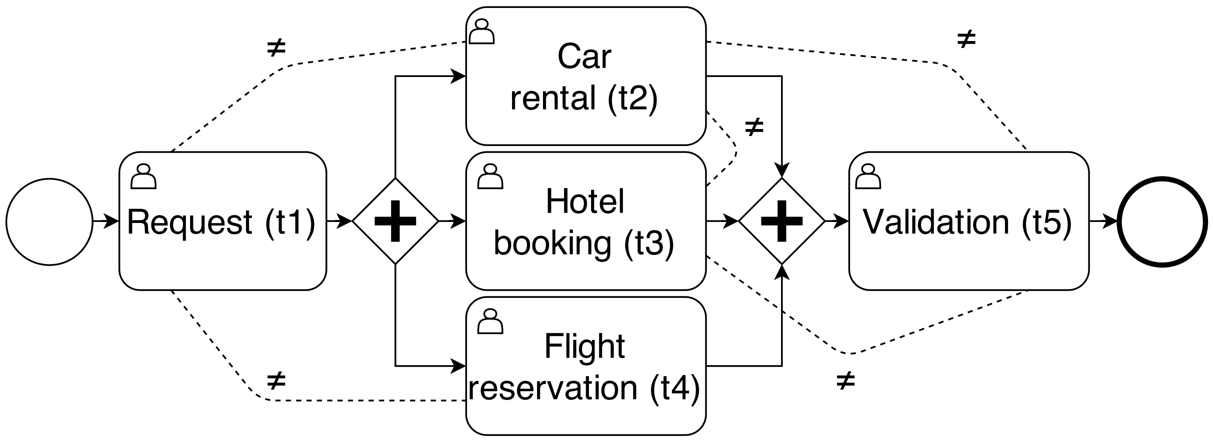


Figure 3.1: TRW in extended BPMN

to be distinct from the one that has executed $t1$, i.e. the user who requests the trip cannot also rent a car. The SoD constraints used in this example are not necessarily realistic, but are used for illustrative purposes. □

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

Off-line. We first construct a symbolic transition system S whose executions correspond to those of the security-sensitive workflow. Then, we use a symbolic model checker to explore all possible terminating executions of the workflow which satisfy both the control-flow and the authorization constraints. We assume the model checker to be able to return a symbolic representation R of the set of all states, called reachable, encountered during the exploration of the terminating executions of S. We use particular classes of formulae in first-order logic [58] to be the symbolic representations of S and R.

On-line. We derive a Datalog [26] program M from the formulae R, representing the set of states reachable in the terminating executions of S and the policy P specifying which users can perform which tasks.

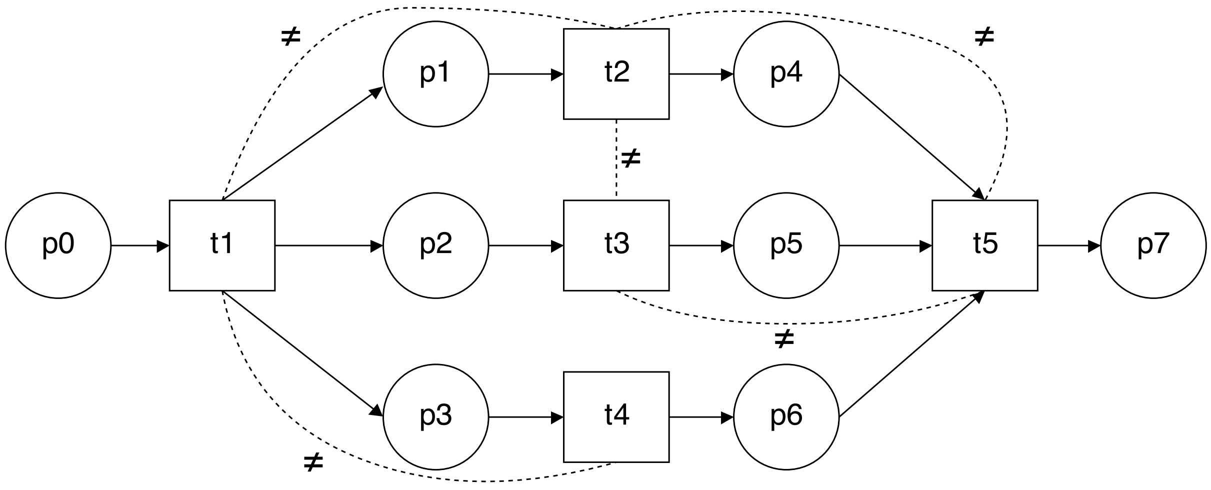


Figure 3.2: TRW as an extended Petri net

The Datalog program M derived in this way is the monitor capable of guaranteeing that any request of a user to execute a task is permitted by P, satisfies the authorization constraints (such as SoD), and the workflow can terminate its execution.

3.1 Overview

We illustrate the two phases of the monitor synthesis technique on the security-sensitive workflow in Figure 3.1.

3.1.1 Off-line phase

First, we build the symbolic transition system S in two steps: (i) we adopt the standard approach (see, e.g., [147]) of using (extensions of) Petri nets to formalize the semantics of workflows and (ii) we adapt the well-known translation of Petri nets to symbolic transition systems (see, e.g., [133]) to the class of extended Petri nets used in this thesis.

Figure 3.2 shows the extended Petri net that can be automatically derived from the BPM notation of Figure 3.1. Tasks (the boxes in the figure) are modeled as transitions or events, whereas places (the circles in the figure) encode their enabling conditions. At the beginning, there will be just one token in place $p0$, which enables the execution of transition $t1$. This corresponds to the execution constraint that task $t1$ must be performed before all the others. The execution of $t1$ removes the token in $p0$ and puts a token in $p1$, another in $p2$, and yet another in $p3$; this enables the execution of $t2$, $t3$, and $t4$. Indeed, this corresponds to the causality constraint that $t2$, $t3$, and $t4$ can be executed in any order after $t1$ and before $t5$. In fact, the executions of $t2$, $t3$, and $t4$ remove the tokens in $p1$, $p2$, and $p3$ and put a token in $p4$, $p5$, and $p6$ which, in turn, enables the execution of $t5$. This removes the token in $p4$, $p5$, and $p6$ and puts a token in $p7$, which enables no more transitions. This corresponds to the fact that $t5$ is the last task to be executed.

The fact that there is at most one token per place is an invariant of the Petri net. This allows us to symbolically represent the net as follows: we introduce a Boolean variable per place (named as the places in Figure 3.2) together with a Boolean variable representing the fact that a task has already been executed (denoted by d_t and if assigned to True implies that task t has been executed). So, for instance, the enabling condition for the execution constraint on task $t1$ can be expressed as $p0 \wedge \neg d_{t1}$, meaning that the token is in place $p0$ and transition $t1$ has not yet been executed. The effect of executing transition $t1$ is to assign F(alse) to $p0$ and T(rue) to $p1$, $p2$, $p3$, and d_{t1}; in symbols, we write $p0, p1, p2, p3, d_{t1} := F, T, T, T, T$. The other transitions are modeled similarly.

Besides the constraints on the execution of tasks, Figure 3.2 shows also the same authorization constraints of Figure 3.1. These are obtained by taking into consideration both the access control policy P granting or denying users the right to execute tasks and the SoD constraints between pairs of tasks. To formalize these, we introduce two functions a_t and h_t from users to Boolean, for each task t, which are such that $a_t(u)$ is true iff u has the right to execute t according to the policy P and $h_t(u)$ is true iff u has executed task t. Notice that a_t is a function that behaves as an abstract interface to the policy P, whereas h_t is a function that evolves over time and keeps track of which users have executed which tasks.

For instance, the enabling condition for the authorization constraint on task $t1$ is simply $a_{t1}(u)$, i.e. it is required that the user u has the right to execute $t1$, and the effect of its execution is to record that u has executed $t1$, i.e. $h_{t1}(u) := T$ (notice that this assignment leaves unchanged the value returned by h_{t1} for any user u' distinct from u). Notice that it is useless to take into account the SoD constraints between $t1$ and $t2$, $t4$ when executing $t1$ since $t2$ and $t4$ will always be executed afterwards. As another example, let us consider the enabling condition for the authorization constraint on $t2$: besides requiring that u has the right to execute $t2$ (i.e. $a_{t2}(u)$), we also need to require the SoD constraints with $t1$ and $t3$ (not that with $t5$, since this will be executed afterwards), i.e. that u has executed neither $t1$ (i.e. $\neg h_{t1}(u)$) nor $t3$ (i.e. $\neg h_{t3}(u)$). The authorization constraints on the other tasks are modeled in a similar way.

Table 3.1 shows the formalization of all transitions in the extended Petri net of Figure 3.2. The first column reports the name of the transition together with the fact that it is dependent on the user u taking the responsibility of its execution. The second column shows the enabling condition divided in two parts: CF, pertaining to the execution constraints, and Auth, to the authorization constraints. The third column lists the effects of the execution of the transition again divided in two parts: CF, for the control-flow, and Auth, for the authorization.

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

$$p0 \wedge \bigwedge_{i=1,\dots,7} \neg pi \wedge \bigwedge_{i=1,\dots,5} \neg d_{ti} \wedge \bigwedge_{i=1,\dots,5} \forall u. \neg h_{ti}(u) \tag{3.1}$$

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

$$p7 \wedge \bigwedge_{i=0,\dots,6} \neg pi \wedge \bigwedge_{i=1,\dots,5} d_{ti} \tag{3.2}$$

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

Formally, the way in which we specify the transition systems corresponding to security-sensitive workflows can be seen as an extended version of the assertional framework proposed in [138]. We emphasize that obtaining, from the extended BPM notation of Figure 3.1, the symbolic representation S of the initial and goal formulae with that of the transitions in Table 3.1 is a fully automated process.

Table 3.1: TRW as symbolic transition system

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

Exploring the search space

After obtaining the symbolic representation of the initial and goal states together with the transitions of the security-sensitive workflow, we invoke a symbolic model checker in order to compute the symbolic representation R of the set of (reachable) states visited while executing all possible sequences of transitions leading from an initial to a goal state. A crucial assumption of our approach is that the model checker is able to compute R for any finite number of users. By doing this, the interface functions a_t's can be instantiated with any policy P, i.e., containing any number of users. As a consequence, changes in the authorization policy do not imply to re-run the off-line phase. In summary, our goal is to compute a parametric—in the number n of users—representation of the set of states visited while executing all possible terminating sequences of transitions. From now on, we write R_n to emphasize this fact.

Although the computation of R_n seems to be a daunting task, there exist techniques available in the literature about parameterized model checking (see the seminal paper [1]) that allow us to do this. Among those available, we have chosen the Model Checking Modulo Theories approach proposed in [68] because it uses first-order formulae as the symbolic representation of transition systems and there are available tools, such as MCMT [69], capable of returning the set of reachable states as a first-order formula.

Figure 3.3 shows an excerpt (to keep it readable) of a graph-like representation of the formula R_n for the security-sensitive workflow described by the symbolic transition system derived from Figure 3.1. Each node is associated to a first-order formula: node 0 (bottom of the figure) is labeled by the goal formula (3.2), nodes 17–26 (top of the figure) are labeled by formulae describing sets of states that have non-empty intersection with the set of initial states characterized by the initial formula (3.1), all other

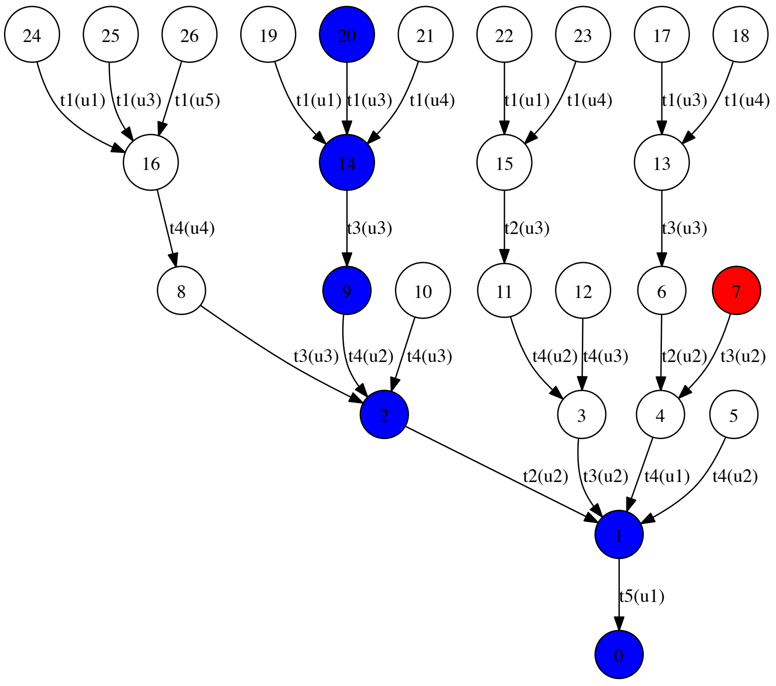


Figure 3.3: Graph-like representation of the set of reachable states for the TRW

nodes (namely, those from 1 to 16) are labeled with formulae describing sets of states that are visited by executing transitions (labeling the arcs of the graph) belonging to a terminating sequence of executions of the workflow.

For instance, node 1 is labeled by the formula

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

describing the set of states from which it is possible to reach a goal state when some user $u1$ takes the responsibility to execute task $t5$. The first two lines in the formula above require that there is a token in places $p4$, $p5$, and $p6$ (thereby enabling transition $t5$), tasks $t1$, $t2$, $t3$, and $t4$ have been executed, and $t5$ has not yet been performed. The last line requires that user $u1$ has the right to execute $t5$ and that he/she has performed neither $t2$ nor $t3$ (because of the SoD constraints between $t5$ and $t2$ or $t3$).

In general, let us consider an arc $\nu \xrightarrow{t(u)} \nu'$ in the graph of Figure 3.3: the formula labeling node ν describes the set of states from which it is possible to reach the set of states described by the formula labeling node ν' when user u executes task t. Thus, the paths starting from one of the nodes 17–26 (labeled by formulae representing states with non-empty intersection with the set of initial states) and ending in node 0 (labeled by the goal formula) describe all possible terminating executions of the workflow in Figure 3.1 (although nodes 5, 7, 10 and 12 seem to be exceptions, this is not the case: explaining their role requires a more precise description of how the graph is built and will be discussed in Section 3.2).

For instance, the sequence of blue nodes describes the terminating sequence $t1, t3, t4, t2, t5$ of task executions by the users $u3$, $u3$, $u2$, $u2$, and $u1$, respectively. It is easy to check that this sequence satisfies both the execution and the authorization constraints required by the workflow in Figure 3.1. In fact, $t1$ is executed first, $t5$ is executed last, and $t2, t3, t4$ are executed in between; there are three *distinct* users $u1, u2, u3$ that can execute the five tasks without violating any of the SoD constraints. By considering all possible paths in the graph of Figure 3.3, it is easy to see that there should be at least three distinct users to be able to terminate the security-sensitive workflow in Figure 3.1.

From what we said above, the formula R_n representing the set of states visited during terminating sequences of task executions of the security-sensitive workflow in Figure 3.1 can be obtained by taking the disjunction of the formulae labeling the nodes in the graph of Figure 3.3 except for the one labeling node 0 since, by construction, no task is enabled in the set of states represented by that formula. Let r_ν be the formula labeling node ν, then

$$R_n := \bigvee_{\nu \in N} r_\nu \tag{3.3}$$

where N is the set of nodes in the graph (in the case of Figure 3.3, we have $N = \{1, \ldots, 26\}$).

3.1.2 On-line phase

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

We have chosen Datalog as the programming paradigm in which to encode monitors for three main reasons. First, it is well-known [94] that a wide variety of access control policies can be easily expressed in Datalog. Second, Datalog permits efficient computations: the class of Datalog programs resulting from translating formulae R_n permits to answer queries in *LogSpace* (see below for more details). Third, it is possible to further translate the class of Datalog programs we produce to SQL statements so that run-time monitors can be easily implemented as database-backed applications [142].

In the rest of this section, we describe how it is possible to derive Datalog programs from formulae describing the set of reachable states computed by the model checker and then how to add the definitions of the interface functions a_{t1},. . . ,a_{t5}.

From R_n to Datalog

Recall the form (3.3) of R_n. It is not difficult to see that each r_ν can be seen as the conjunction of a formula r_ν^{CF} containing the Boolean functions $p0, \ldots, p7$ for places and $d_{t1}, \ldots, d_{t5}$ keeping track of task execution with a formula r_ν^{Auth} of the form

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

where $u0$ identifies the user taking the responsibility to execute task t, ρ_ν^{Auth} is a formula containing the variables $u0, u1, \ldots, uk$, the interface functions $a_{t1}, \ldots, a_{t5}$, the history functions $h_{t1}, \ldots, h_{t5}$, and all disequalities between pairwise distinct variables from $u0, u1, \ldots, uk$ (indeed, if there are no variables, there is no need to add such disequalities). For instance, formula r_1 labeling node 1 in Figure 3.3 is $r_1^{\text{CF}} \wedge r_1^{\text{Auth}}$ where

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

with $u0$ renamed to $u1$.

In general, each r_ν in the expression (3.3) for the formula R_n can be written as

$$r_\nu^{\text{CF}} \wedge a_t(u0) \wedge \rho_\nu^{\text{Auth}}(u0, u1, \ldots, uk) \tag{3.4}$$

and describes a set of states in which user $u0$ executes task t while guaranteeing that the workflow will terminate since ν is one of the nodes in the graph computed by the model checker while generating all terminating sequences of tasks. In other words, (3.4) implies that $u0$ can execute task t or, equivalently written as a Datalog clause: $can\_do(u0, t) \leftarrow$ (3.4), where *can_do* is a Boolean function returning true iff a user (first argument) is entitled to execute a task (second argument) while all execution and authorization constraints are satisfied and the workflow can terminate.

Notice that $can\_do(u0,t) \leftarrow$ (3.4) is a Datalog clause. So, we generate the following Datalog clauses

$$can\_do(u0,t) \leftarrow r_\nu^{\mathrm{CF}} \wedge a_t(u0) \wedge \rho_\nu^{\mathrm{Auth}}(u0,u1,\ldots,uk) \tag{3.5}$$

for each $\nu \in N$. In the following, let D_n be the Datalog program composed of all the clauses of the form (3.5). For instance, the Datalog clause corresponding to node 1 is

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

It is not difficult to show that $can\_do(u,t)$ iff there exists a disjunct of the form (3.4) in R_n for a given number n of users. Finally, observe that clauses of the form (3.5) contain negations but are non-recursive.

Specifying the policy P

We are left with the problem of specifying the access control policy P for a given number n of users. As already observed above, there should be at least three distinct users in the system to be able to terminate the execution of the workflow in Figure 3.1.

So, to illustrate, let $U = \{a,b,c\}$ be the set of users and use the RBAC model to express the policy. This means that we have a set $R = \{r_1, r_2, r_3\}$ of roles which are indirections between users and (permissions to execute) tasks. Let $UA = \{(a,r1),(a,r2),(a,r3),(b,r2),(b,r3),(c,r2)\}$ be the user-role assignments and $TA = \{(r_3,t1),(r_2,t2),(r_2,t3),(r_1,t4),(r_2,t5)\}$ be the role-task assignment. Then, a user u can execute task t iff there exists a role r such that $(u, r) \in UA$ and $(r, t) \in TA$. This can be formalized by the following Datalog clauses:

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

and denoted by D_P.

By taking the union of the clauses of D_n and D_P, we build a Datalog program $M_{n=3}$ allowing us to monitor the security-sensitive workflow of Figure 3.1. I.e. $M_{n=3}$ is capable of answering queries of the form $can\_do(u, t)$ in such a way that all execution and authorization constraints are satisfied and the workflow execution terminates.

An example of a run of the monitor is in Table 3.2, where each line represents a state of the system; columns 'CF' and 'Auth' describe the values of the variables in that state ('Token in' shows which places have a token and the various 'h_{ti}' hold the name of the user who executed task t_i); '$can\_do(u, t)$' represents user u requesting to execute task t and 'Response' is the corresponding response returned by the monitor (grant or deny the request).

Table 3.2: A run of the monitor program $M_{n=3}$ for the TRW

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

The execution in the table shows two denied requests, one in line 0 and one in line 2. In line 0, user a requests to execute task $t1$ but this is not possible since a is the only user authorized to execute $t4$, and if a executes $t1$, he/she will not be allowed to execute $t4$ because of the SoD constraint between $t1$ and $t4$ (see Figure 3.1). In line 2, user b requests to execute task $t2$ but again this is not possible since b has already executed task $t1$ and this would violate the SoD constraint between $t1$ and $t2$. All the other requests are granted, as they do not violate neither execution nor authorization constraints.

3.2 Formal description

Considering the specification of workflows as transition systems presented in Section 3.1, we now describe how a symbolic model checker can compute a reachability graph that represents all terminating executions of the workflow (off-line phase) and how this is then translated to a Datalog program that implements the run-time monitor for the WSP (on-line phase).

3.2.1 Off-line

As already observed in Section 3.1.1, it is standard to use (extensions of) Petri nets to give a formal semantics to workflows written in BPMN [147]. In turn, it is well-known how to represent (extension of) Petri nets as state transition systems (see, e.g., [133]), that are composed of a set of *state variables* and a set of *events*, as proposed in [138].

A *state* of the system is defined by the values of the variables. A *predicate* (Boolean function) over the state variables implicitly defines a set of states, i.e. the one containing the values of the variables for which the predicate evaluates to true. A state *satisfies* a predicate iff it belongs to the set of states implicitly defined by the predicate.

An event has an *enabling condition*, which is a predicate on the state variables, and an *action*, which updates the state variables. When the enabling condition of an event evaluates to true in a given state s, we say that the event is *enabled* at s. Executing an event enabled at state s results in a new state s' obtained by applying the update of the event to the values of the variables in s.

Definition 1. *A* behavior *is a sequence of the form* $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots$ *where* s_i *is a state,* e_i *is an event, and state* s_{i+1} *is obtained by executing event* e_i *in state* s_i*, for* $i = 0, 1, \ldots$. *We say that a state* s_n *is* reachable *from a state* s_0 *iff there exists a behavior* $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \cdots s_{n-1} \xrightarrow{e_{n-1}} s_n$.

For the class of security-sensitive workflows considered in this thesis, the set V of state variables is the union of a set V_{CF} and a set V_{Auth} where the former contains a Boolean variable p_i for each place in the Petri net (for $i = 0, 1, \ldots$) and a Boolean variable d_t for each transition t in the Petri net, whereas the latter contains two function variables a_t and h_t mapping the set U of users to Booleans for each transition t in the net.

Intuitively, p_i is true iff there is a token in the corresponding place, d_t is true iff task t has been executed, $a_t(u)$ is true iff user u has the right to execute task t, and $h_t(u)$ is true iff user u has executed task t.

The enabling condition and the action of an event t are of the following forms: $enabled_{\text{CF}} \wedge enabled_{\text{Auth}}$ and $act_{CF} || act_{Auth}$, respectively, where $enabled_{\text{CF}}$ is a predicate over V_{CF}, $enabled_{\text{Auth}}$ is a predicate over V_{Auth}, act_{CF} (act_{Auth}, resp.) is the parallel ($||$) updates of (some of) the variables in V_{CF} (V_{Auth}, resp.), which are written as $x_1, \ldots, x_k := v_1, \ldots, v_k$ for x_i a state variable and v_i is the value to which x_i should be updated to. An update of a function variable f from users to Booleans is written as $f(u) := b$ where u is a user, b is a Boolean value, and after the update the function is identical to the previous one except at u for which the value b is returned.

An event is a tuple $(t(u), \mathit{enabled}_{\mathrm{CF}} \wedge \mathit{enabled}_{\mathrm{Auth}}, \mathit{act}_{CF} || \mathit{act}_{Auth})$ written as

$$t(u) : \mathit{enabled}_{\mathrm{CF}} \wedge \mathit{enabled}_{\mathrm{Auth}} \rightarrow \mathit{act}_{CF} || \mathit{act}_{Auth} \tag{3.6}$$

where t is the name of the event (taken from a finite set) and u is a user. Notice that an event is parametric with respect to a user; thus, (3.6) specifies a collection of events, one for every u in the set U of users.

Definition 2. *A* security-sensitive (state) transition system over the finite set U of users *is a tuple* $(V_{CF} \cup V_{Auth}, Tr)$ *where* U *is a finite set of users,* $V_{CF} \cup V_{Auth}$ *is the set of state variables as described above, and* Tr *is the set of events obtained by considering all users in* U*.*

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

Symbolic behaviors

We solve the problem of enumerating all possible behaviors of a security-sensitive workflow $S = (V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Tr)$ for every subset U of users in $\mathcal{U}$ by using a symbolic representation for S and U. We use first-order logic formulae to represent sets of states.

A *state formula* is a first-order formula containing (at most) the state variables in $V_{\mathrm{CF}} \cup V_{\mathrm{Auth}} \cup V_{\mathrm{User}}$ as free variables where V_{User} is a set of variables taking values over the set U of users. A state formula P evaluates to true (in symbols, $s, v \models P$) or false (in symbols, $s, v \not\models P$) in a state s of the system and for an assignment v of the user variables (i.e., a mapping from V_{User} to U): for each variable x in $V_{\text{CF}} \cup V_{\text{Auth}} \cup V_{\text{User}}$ that appears free in P, replace x by its value in s or v and then evaluate the resulting formula. In other words, state formulae define predicates or, equivalently, sets of states. Examples of state formulae are (3.1) and (3.2) describing the sets of initial and final states, respectively, of the security-sensitive workflow in Figure 3.2.

A *symbolic event* is a tuple of the form (3.6) where u is a first-order variable in V_{User}, $enabled_{\text{CF}}$ is a state formula over V_{CF}, and $enabled_{\text{Auth}}$ is a state formula over $V_{\text{Auth}} \cup V_{\text{User}}$, act_{CF} is as before, and act_{Auth} is of the form $f(u) := b$ where b is a Boolean value and u is the same variable in the label $t(u)$.

Definition 3. *A* symbolic security-sensitive transition system *is a tuple* $(V_{CF} \cup V_{Auth} \cup V_{user}, Ev)$ *where* $V_{CF} \cup V_{Auth}$ *is the set of state variables,* V_{User} *is the set of user variables, and* Ev *is a finite set of symbolic events.*

The semantics of a symbolic security-sensitive transition system $(V_{\text{CF}} \cup V_{\text{Auth}} \cup V_{\text{User}}, Ev)$ is axiomatically defined by using the notion of weakest liberal precondition (wlp) [52]:

$$\mathsf{wlp}(Ev, P) \;:=\; \bigvee_{(t(u):en \to act) \in Ev} (en \wedge P[act]) \tag{3.7}$$

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

$\mathsf{wlp}(Ev, P)$ is equivalent to $\bigvee_{ev \in Ev} \mathsf{wlp}(ev, P)$. To make expressions more compact, we also write $\mathsf{wlp}(t(u), P)$ instead of $\mathsf{wlp}(t(u) : en \to act, P)$.

Example 2. To illustrate, we compute $\mathsf{wlp}(t5(u), (3.2))$ where the symbolic event $t5(u)$ is defined in Table 3.1 by using (3.7):

$$\left(\begin{array}{l} p4 \wedge p5 \wedge p6 \wedge \neg d_{t5} \wedge \\ a_{t5}(u) \wedge \neg h_{t3}(u) \wedge \neg h_{t2}(u) \end{array}\right) \wedge \Big(\bigwedge_{i=0,\ldots,3} \neg pi \wedge \bigwedge_{i=1,\ldots,4} d_{ti} \Big)$$

which is equivalent to

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

and it identifies those states in which there is a token in places $p4$, $p5$, and $p6$, task $t5$ has not yet been executed whereas tasks $t1$, ..., $t4$ have been executed, user $u1$ has the right to execute $t5$ and has executed neither $t2$ nor $t3$. This is exactly the formula labeling node 1 in Figure 3.3. □

Definition 4. *A* symbolic behavior *is a sequence of the form* $P_0 \xrightarrow{e_0} P_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} P_n$ *where* P_i *is a state formula and* e_i *is a symbolic event such that (a)* $P_0 \wedge I$ *is satisfiable, (b)* P_i *is logically equivalent to* $\mathsf{wlp}(e_i, P_{i+1})$ *for* $i = 0, \ldots, n-1$*, and (c)* P_n *is* F *for* I *and* F *formulae characterizing the initial and final states, respectively.*

The crucial advantage of symbolic events is the use of variables to represent users instead of enumerating them. To illustrate, consider a simple security-sensitive workflow with just two tasks t_1, t_2 such that t_1 should be executed before t_2 and there is a SoD constraint between them. If the cardinality of the set U of users is n, then the cardinality of the set of all possible behaviors is $n^2 - n$. By using symbolic events, we can represent all such behaviors by a single symbolic behavior $P_0 \xrightarrow{t_1(u1)} P_1 \xrightarrow{t_2(u2)} P_2$ with the proviso that $u1 \neq u2$ where $u1, u2$ are variables.

Before stating formally this result, we need to introduce the notion of security-sensitive transition system $T = (V_{\mathrm{CF}} \cup V_{\mathrm{Auth}}, Ev_T)$ *associated to a symbolic security-sensitive transition system* $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ *and a finite set* $U \subseteq \mathcal{U}$ *of users*: if the symbolic event $t(ui) : en_S \to act_S$ is in Ev_S, then Ev_T contains an event $t(u_i) : en \to act$ where u is a user in U, en is the predicate interpreting the formula obtained from en_S by substituting the variable ui with u_i and all other user variables with users in U (in all possible ways), and act is obtained from act_S by substituting ui with u_i.

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

Proof. (Sketch) The proof is by a standard induction on the length of the behaviors. It exploits the fact that the enforcement of authorization constraints depends only on two aspects. First, the identity of users (via the state variables a_t's modeling the interface to the concrete authorization policy establishing if a user has the right to execute a task). Second, the history of the computation (via the state variables h_t's keeping track of who has executed which tasks so that SoD and BoD constraints can be guaranteed to hold). □

This result tells us that a symbolic behavior is an adequate (and hopefully compact) representation of a set of behaviors.

Computation of symbolic behaviors

Algorithm 1 computes the set of all possible symbolic behaviors of a symbolic security-sensitive workflow. It takes as input the symbolic security-sensitive workflow S together with the state formula F defining the set of final states and returns a labeled graph RG, called *reachability graph*, whose set of labeled paths is the set of all symbolic behaviors of S ending with F. The procedure incrementally builds the reachability graph RG by updating the set N of nodes, the set E of edges, and the labeling function λ from N to state formulae.

Initially (line 1), a new node i is created (by invoking the auxiliary function `new`, which returns a "fresh" node—i.e. distinct from any other node already in N—at each invocation), N is assigned to the singleton containing node i, which is also labeled (via λ) by the final formula F.

Algorithm 1 Building a symbolic reachability graph

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

The algorithm also maintains the set TBV of nodes to be visited, which is made equal to N.

Then, the main loop (lines 2–14) is entered by checking if there are some nodes to be visited (line 2). At each iteration, it is first (line 3) checked whether the set of states identified by the wlp of the formula $\lambda[i]$ with respect to the set Ev_S of symbolic events is included in the union of the sets of states that have been already generated. This is done by invoking `subsumed`(i,N,N') which returns true iff, for each symbolic event $ev \in Ev_S$, there exists a subset N' of $N - \{i\}$ and $wlp(ev, \lambda[i])$ implies the formula $\bigvee_{j \in N'} \lambda[j]$ (notice that the third argument N' is passed by reference). If this is the case, we can avoid to add a new node ν to N labeled by $wlp(ev, \lambda[i])$ as the symbolic behaviors arriving in ν have already been generated when visiting the nodes in N'. Thus, we can delete node i from TBV, add a new node j labeled by $wlp(ev, \lambda[i])$ together with an edge from j to i labeled by $\overline{ev}$ and—by invoking the auxiliary function `connect`—duplicate the initial part of each path passing through a node n' in N' by replacing n' with j provided that the newly created path is a symbolic behavior of the symbolic transition system.

Example 3. To illustrate, consider node 7 in Figure 3.3 (colored in red): $wlp(ti(u), \lambda[7])$ is unsatisfiable for $i = 1, 3, 4, 5$ (and can thus be ignored) whereas $wlp(t2(u3), \lambda[7])$ is satisfiable and implies $\lambda[13]$; this is checked by invoking `subsumed` with $N' = \{13\}$. Thus, we create a new node (say) 29, with $\lambda[29]$ equal to $wlp(t2(u3), \lambda[7])$, draw an edge from 29 to 7 with label $t2(u3)$, duplicate the initial parts of the paths passing through node 13 (namely $\lambda[17] \xrightarrow{t1(u3)} \lambda[13]$ and $\lambda[18] \xrightarrow{t1(u4)} \lambda[13]$) while replacing 13 with 29 (thus obtaining $\lambda[17] \xrightarrow{t1(u3)} \lambda[29]$ and $\lambda[18] \xrightarrow{t1(u4)} \lambda[29]$), and then check that the newly created paths, namely

$$\lambda[17] \xrightarrow{t1(u3)} \lambda[29] \xrightarrow{t2(u3)} \lambda[7] \xrightarrow{t3(u2)} \lambda[4] \xrightarrow{t4(u1)} \lambda[1] \xrightarrow{t5(u1)} \lambda[0] \text{ and}$$
$$\lambda[18] \xrightarrow{t1(u4)} \lambda[29] \xrightarrow{t2(u3)} \lambda[7] \xrightarrow{t3(u2)} \lambda[4] \xrightarrow{t4(u1)} \lambda[1] \xrightarrow{t5(u1)} \lambda[0]\,,$$

are symbolic behaviors. It turns out that only the latter is so, since the former violates the SoD constraint between $t1$ and $t2$. We thus add only the path $\lambda[18] \xrightarrow{t1(u4)} \lambda[29] \xrightarrow{t2(u3)} \lambda[7]$ to the graph in Figure 3.3. Nodes 5, 10, and 12 are handled similarly. These extensions to the graph in Figure 3.3 are omitted to keep it readable. □

If node i is not subsumed by those in N (i.e. `subsumed`(i,N) returns false), we compute the wlp with respect to all symbolic events (inner loop 6–11). I.e., for each ev in Ev_S, we compute $\mathsf{wlp}(ev, \lambda[i])$ labeling the node i being visited (line 6) and verify if it defines a set of states which is non-empty, by checking the satisfiability of the resulting formula (line 7). If this is the case, we add a fresh node j, labeled by the wlp just computed, to N, an edge from i to j labeled by the name $\overline{ev}$ of the symbolic event ev, and add the newly created node j to the set TBV (lines 8 and 9). For instance, when computing the wlp of the formula labeling node 0 in Figure 3.3, we found out that only the symbolic event named $t5(u1)$ generates a formula denoting a non-empty set of states and thus we added node 1 labeled by such a formula and an edge from 1 to 0 labeled by $t5(u1)$.

After exiting the inner loop, if the set TBV of nodes to be visited is non-empty, we consider another node to be visited by invoking the auxiliary function `pickOne`(TBV) which non-deterministically selects an element from TBV (when this is empty, `pickOne` returns a distinguished element), which is then deleted, and we start the main loop again.

Theorem 2. *Let I be the initial state formula. If Algorithm 1 returns the reachability graph RG when taking as input the symbolic security-sensitive transition system $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ and the final state formula*

F, then the set of all symbolic behaviors of S is the set of labeled paths in RG starting with a node labeled by a formula whose conjunction with I is satisfiable and ending with a node labeled by F.

Proof. (Sketch) The proof of this theorem uses two previous concepts. First, the definition of wlp in formula 3.7. Second, the properties discussed above about the auxiliary functions `subsumed` and `connect`, i.e., the fact that Algorithm 1 returns a complete RG. It is possible to show that Algorithm 1 always terminates by adapting the results in [16]. $\square$

3.2.2 On-line

Theorem 2 implies that starting from an initial state (i.e., one satisfying the initial formula I) in the reachability graph computed by Algorithm 1, it is always possible to reach a final state (i.e., one satisfying the final formula F). If no event can be enabled infinitely often without being executed—the *strong fairness* assumption—then a final state is eventually reached. (As observed in [150], the assumption of strong fairness is reasonable in the context of workflow management since decisions to execute tasks are under the responsibility of applications or humans.) This is the key to prove the following result, underlying the correctness of the automated technique—to be described below—for extracting (part of) the monitor from the reachability graph computed by Algorithm 1.

Theorem 3. *Let $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ be a symbolic security-sensitive transition system and $T = (V_{CF} \cup V_{Auth}, Ev_T)$ be the associated security-sensitive transition system for the finite set $U \subseteq \mathcal{U}$ of users. Furthermore, let $RG = (N, \lambda, E)$ be the symbolic reachability graph computed by Algorithm 1 when taking as input S and a final state formula F. If the state s satisfies a formula $\lambda[i]$ for some $i \in N$, then there exists a behavior $s_0 \overset{t_0(u_0)}{\to} s_1 \overset{t_1(u_1)}{\to} \cdots s_{n-1} \overset{t_{n-1}(u_{n-1})}{\longrightarrow} s_n$ of T such that (i) $s_0 = s$, (ii) s_n satisfies*

F, and (iii) $(i, t(x), j) \in E$ with $t_0 = t$ and $s_0(x) = u_0$.

Proof. (Sketch) This theorem is a direct consequence of three assumptions. First, strong fairness (described above) states that a final state is eventually reached. Second, Theorem 1 connects the symbolic behavior in S to the behavior in T. Third, Theorem 2 shows that the set of all symbolic behaviors of S is the set of labeled paths in RG from an initial to a final state. □

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

To make this operational, we observe that we can associate the Datalog clause

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

for each node $i \in N$ and edge $(i, t(u), j) \in E$, where Γ is the conjunction of atoms of the form $is\_user(x)$ for each variable x in $C_k[i]$ with $RG = (N, \lambda, E)$ and $\bigvee_{k=1}^{n_i} C_k[i]$ is the disjunctive normal form of $\lambda[i]$. Let $D(RG)$ be the set of Datalog clauses built in this way from the reachability graph RG. (It is straightforward to check that $D(RG)$ is non-recursive; see [26] for a precise definition). Formally, the addition of Γ is needed to make $D(RG)$ a safe Datalog program (see again [26] for a precise definition) so that answering queries always terminates.

After building the Datalog program $D(RG)$, it is straightforward to build a run-time monitor. Let U be a finite set of users, $A \subseteq V_{\text{Auth}}$ be the subset of state variables a_t's modeling the interface to the concrete authorization policy establishing if a user has the right to execute a task, and P

be a Datalog program formalizing an authorization policy (i.e., P contains a clause of the form $is\_user(u)$ for each $u \in U$ and clauses whose heads contain only the predicates in A). We call P a *Datalog authorization policy program over the interface variables in V_{Auth}*. (How to write authorization policies in Datalog is outside the scope of this thesis, the interested reader is pointed to [94].)

Any assignment over the state variables in $V_{\text{CF}} \cup (V_{\text{Auth}} - A)$ can be represented by a set Σ of Datalog facts of the forms p, $\neg p$, d_t,$\neg d_t$, $h_t(u)$, or $\neg h_t(u)$ for $p, d_t \in V_{\text{CF}}$ and $h_t \in (V_{\text{Auth}} - A)$. We call Σ a *partial Datalog state over the state variables in $V_{CF} \cup (V_{Auth} - A)$*.

Theorem 4. *Let $S = (V_{CF} \cup V_{Auth} \cup V_{User}, Ev_S)$ be a symbolic security-sensitive transition system, $T = (V_{CF} \cup V_{Auth}, Ev_T)$ be the associated security-sensitive transition system for the finite set $U \subseteq \mathcal{U}$ of users, and $RG = (N, \lambda, E)$ be the symbolic reachability graph computed by Algorithm 1 when taking as input S and a final state formula F. Additionally, let P be a Datalog authorization policy over the interface variables in V_{Auth} and Σ be a partial Datalog state. A user $u \in U$ can execute task t guaranteeing the satisfaction of all authorization constraints and the termination of the workflow iff the query $can\_do(u, t)$ is answered positively by the Datalog program $D(RG) \cup P \cup \Sigma$.*

Proof. (Sketch) This theorem is a consequence of the definitions of Datalog authorization policy program and partial Datalog state (defined just before this theorem), and Theorem 3.

$D(RG)$ can be seen as an abstract version of the monitor where the policy P is left unspecified or, equivalently, $D(RG)$ works for any possible policy. The addition of the clauses in P that define the policy has the effect of instantiating $D(RG)$ to the particular policy P. Finally, the addition of Σ instantiates $D(RG) \cup P$ to a particular state in the execution of the security-sensitive workflow. □

This is the main result of this Chapter and guarantees the correctness of our procedure to synthesize run-time monitors. Notice that when both $D(RG)$ and P are non-recursive (stratified) Datalog programs, queries can be answered efficiently in *LogSpace* and can be translated to SQL without aggregate operators (such as AVG and COUNT).

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

Our idea is to split a complex workflow into components that can be again decomposed into smaller components up to a desired level of detail. Several workflow management systems support this style of specification following an established line of works in both academia [143] and industry [134]. It is also possible to decompose large workflows into smaller components by using pre-existing techniques [125].

Besides fostering reuse and simplifying maintenance of complex workflows, modular specifications allow for the development of a divide-and-conquer strategy when applying Algorithm 1. I.e. given a modular workflow, it is possible to compute its monitor for the WSP by first computing the monitors for each of the components separately and then "gluing" them together. This modular strategy is detailed in Chapter 4.

Chapter 4

Modularity for security-sensitive workflows[1]

Business process designers constantly strive to adapt to rapidly evolving markets under continuous pressure of regulatory and technological changes. In this respect, a frequent problem faced by companies is the lack of automation when trying to incorporate new requirements into existing processes. A traditional approach to business process modeling frequently results in large models that are difficult to change and maintain. This makes it critical that business process models be modular and flexible, not only for increased modeling agility at design-time but also for greater robustness and flexibility of enacting at run-time (see, e.g., [106] for a discussion about this and related problems).

The modular design of business processes has been advocated for a long time in academia because of its support to reuse at design-time and scalability at run-time [123, 124]. In industry, it is more and more common to find solutions allowing the reuse of (parts of) workflows to realize complex business processes. For instance, SAP Operational Process Intelligence[2] supports the creation of end-to-end business processes spanning multiple

[1]Parts of this chapter were previously published in [54]

[2]`https://help.sap.com/hana-opint`

workflows. Such (template) workflows can be created once and stored in a repository to be then operated in different contexts. As an example, a *Purchase Order* workflow with tasks *Create Purchase Order* and *Create Invoice* would be part of any end-to-end business process selling goods, whereas a *Warehouse Management* workflow composed of tasks *Locate Product* and *Send Product* would be included only in cases where physical goods are involved.

Although techniques for modular specification and enactment of workflows and their impact have been extensively studied in the literature (see, e.g., [123, 124, 93]), the same is not true for security-sensitive workflows. In this special class of workflows, not only the control-flow spans several modules, but even authorization constraints may be defined across different components. Given the difficulties in specifying and enforcing execution and authorization constraints in this context, it is not surprising that vulnerabilities can be exploited by malicious users. For example, recently, the incorrect handling of authorization constraints between a *Purchase Order* and a *Warehouse Management* workflow allowed an Amazon employee to pay for cheap products and deliver expensive electronics to himself[3]. This kind of fraud could be avoided by specifying at design-time and enforcing at run-time a SoD constraint between tasks *Create Purchase Order* and *Send Product*.

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

(i) how to specify authorization constraints that span multiple modules (inter-module constraints)?

(ii) how to enforce such constraints?

(iii) how to scale the enforcement mechanism and handle large workflows?

[3]`https://goo.gl/1bySZH`

Indeed, the last question is not unique to modular workflows, but is important to the whole class of security-sensitive workflows. In this Chapter, we introduce an approach capable of answering the questions above by making the following contributions:

- the definition of security-sensitive workflow components equipped with interfaces (Section 4.2) that allow to glue components together and define constraints between them (Section 4.3), to answer question (i); and

- an automated technique, extending the work in Chapter 3, to synthesize run-time monitors from workflow components ensuring that all tasks can be executed without violating the policy or the constraints (Section 4.4), to answer questions (ii) and (iii).

4.1 Modular design and enactment

We introduce our approach for the modular design and enactment of security-sensitive workflows by combining the TRW introduced in Example 1 with another example workflow.

Example 4. Figure 4.1 shows the Moderate Discussion Workflow (MDW) whose goal is to organize a discussion and voting process in an organization. It is composed of four tasks: Request ($t1$), Moderate Conference Call ($t6$), Moderate e-mail Discussion ($t7$), and Validation ($t5$). Four SoD constraints must be enforced: $(t1, t6)$, $(t6, t5)$, $(t6, t7)$, and $(t7, t5)$. Each task is executed under the responsibility of a user who is entitled to do so according to some authorization policy, which we leave unspecified for the sake of brevity and because the synthesis technique that we use generates a monitor that can accommodate any authorization policy. □

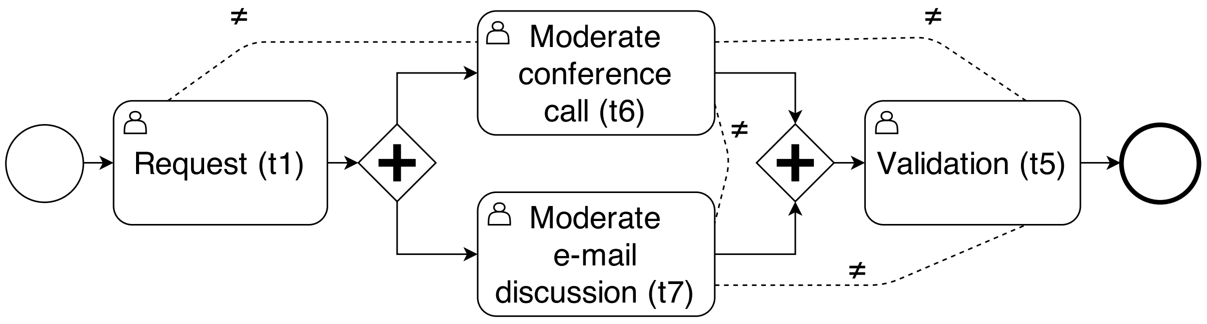

Request (t1)
Moderate conference call (t6)
Moderate e-mail discussion (t7)
Validation (t5)
≠
≠
≠
≠

Figure 4.1: MDW in extended BPMN

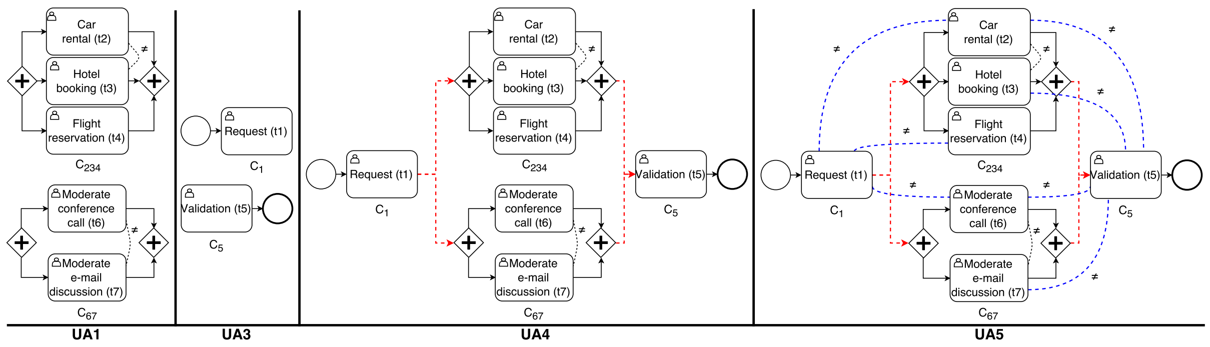


Figure 4.2: User actions necessary to specify and compose modules representing the TRW and MDW

Notice that tasks $t1$ and $t5$ in Figures 3.1 and 4.1 are the same in both the TRW and the MDW. The notion of security-sensitive component introduced in this Chapter allows to reuse the specification of tasks $t1$ and $t5$ in different workflows so that only the specification of the parallel execution of tasks $t2$, $t3$, and $t4$ for the TRW and $t6$ and $t7$ for the MDW must be developed from scratch.

By using the approach in this Chapter, a process designer can model both the TRW and the MDW by executing the following user actions, that are also depicted in Figure 4.2 (where the elements in black represent the internal specification of components, the red dashed arrows represent inter-component execution (control-flow) constraints and the blue dashed lines represent inter-component authorization constraints):

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

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

UA3 import, from the available workflow repository, the security-sensitive components containing tasks $t1$ and $t5$ in Figures 3.1 and 4.1, called $\mathcal{C}_1$ and $\mathcal{C}_5$, respectively;

UA4 define the control-flow among components; and

UA5 define inter-component authorization constraints.

To enact the modularly designed business processes TRW and MDW, the designer can simply add an authorization policy and deploy the process to the run-time environment. Behind the scenes, the monitors of the various components are automatically combined to build one for the composed processes, namely TRW and MDW. This combination is done by using a set G of "gluing assertions," which are logical assertions connecting the components, i.e. transferring control-flow and constraining the execution of tasks in the next components.

The main result of this Chapter (Theorem 6) shows that the combination of monitors M_1, M_{234}, and M_5 synthesized for components $\mathcal{C}_1$, $\mathcal{C}_{234}$ and $\mathcal{C}_5$, respectively, with their Datalog authorization policies P_1, P_{234}, P_5 and their execution histories H_1, H_{234}, H_5, and using the assertions in G, answers to user requests in the same way as a monitor M computed for the TRW as a single component. Formally, $M_1, M_{234}, M_5, G, P_1, P_{234}, P_5, H_1, H_{234}, H_5 \vdash \mathit{can\_do}(u,t)$ iff $M, P, H \vdash \mathit{can\_do}(u,t)$. Therefore, a similar run as the one shown in Table 3.2 for M can be obtained with M_1, M_{234}, M_5.

Indeed, the simplicity of the TRW and the MDW spoils the advantages of a modular approach. However, for large workflows the advantages are substantial. To give an intuition of this, imagine replacing the tasks reused in both workflows, i.e. $t1$ and $t5$, with complex workflows: reusing their specifications and synthesized run-time monitors in larger workflows in which they are plugged, becomes much more interesting.

4.2 Security-sensitive workflow components

The goal of this Section is to identify a refinement of the notion of security-sensitive workflow (introduced in Chapter 3) that can be modularly composed with others through an appropriate interface.

Technically, this is done by extending and partitioning the state variables of the transition system representing a security-sensitive workflow and then adding an appropriate notion of interface to support composition. The resulting notion is called a security-sensitive component. A *(symbolic) security-sensitive component* is a pair (S, Int) where S is a (symbolic) security-sensitive transition system and Int is its interface.

Security-sensitive transition system

Recall the description of the transition system $S = (V, Tr)$, with $V = V_{CF} \cup V_{Auth}$, associated to a security-sensitive workflow. We redefine a *(symbolic) security-sensitive transition system* as follows.

Definition 5. *A (symbolic) security-sensitive transition system S is a tuple of the form $((P, D, A, H, C), Tr, B)$ where $(P \cup D \cup A \cup H \cup C) = V$ are the state variables, Tr is a set of transitions, and B is a set of constraints on the state variables in C.*

The finite set P contains Boolean variables representing the places of the Petri net associated to a BPMN specification of the security-sensitive workflow and D is a finite set of Boolean variables representing the fact that a task has been executed or not; $P \cup D$ are called *execution constraint variables*. The finite set A contains interface predicates to the authorization policy, H is a set of predicates recording which users have executed which tasks, and C is a set of interface predicates to the authorization constraints; $A \cup H \cup C$ are called *authorization constraint variables*. The set Tr contains the *transitions* (or *events*) of the form

$$t(u) : en_{\mathrm{EC}}(P, D) \wedge en_{\mathrm{Auth}}(A, C) \rightarrow act_{\mathrm{EC}}(P, D) || act_{\mathrm{Auth}}(H) \tag{4.1}$$

where t is the name of a task taken from a finite set, u is a variable ranging over a set U of users, $en_{\mathrm{EC}}(P, D)$ is a predicate on $P \cup D$ (called the *enabling condition for the execution constraint*), $en_{\mathrm{Auth}}(A, C)$ is a predicate on $\{v(u) | v \in A \cup C\}$ (called the *enabling condition for the authorization constraint*), $act_{\mathrm{EC}}(P, D)$ contains parallel assignments of the form $v := b$ where $v \in P \cup D$ and b is a Boolean value (called the *update of the execution constraint* of the security sensitive workflow), and $act_{\mathrm{Auth}}(H)$ contains parallel assignments of the form $v(u) := b$ where $v \in H$ and b is a Boolean value (called the *update of the authorization history* of the security sensitive workflow).[4] Finally, the finite set B contains *always constraints* of the form

$$\forall u . v(u) \Leftrightarrow hst, \tag{4.2}$$

where u is a variable ranging over users, v is a variable in C, and hst is a Boolean combination of atoms of the form $w(u)$ with $w \in H$.

Interface of a security-sensitive component

Definition 6. *The interface Int of a symbolic security-sensitive component (S, Int) is a tuple of the form (A, P^i, P^o, H^o, C^i) where*

[4]The assignment $v(u) := b$ leaves unchanged the value returned by v for any u' distinct from u. In other words, after the assignment, the value of v can be expressed as follows: $\lambda x.\mathit{if}\ x = u\ \mathit{then}\ b\ \mathit{else}\ v(x)$.

- *$P^i \subseteq P$ and each $p^i \in P^i$ is such that $p^i := T$ does not occur in the parallel assignments of an event of the form (4.1) in Tr,*
- *$P^o \subseteq P$ and each $p^o \in P^o$ is such that $p^o := T$ occurs in the parallel assignments of an event of the form (4.1) in Tr whereas $p^o := F$ does not,*
- *$H^o \subseteq H$, $C^i \subseteq C$, and*
- *only the variables in $(C \setminus C^i) \cup H^o$ can occur in a symbolic always constraint of B.*

When P^i, P^o, H^o, and C^i are all empty, the component (S, Int) can only be interfaced with an authorization policy via the interface variables in A. The state variables in D are only used internally, to indicate that a task has been or has not been executed; thus, none of them is exposed in the interface Int. The variables in P, H, and C are local to S but some of them can be exposed in the interface in order to enable the combination of S with other components in a way which will be described below (Section 4.3). The super-scripts i and o stand for input and output, respectively. The requirement that variables in P^i are not assigned the value T(rue) by any transition of the component allows their values to be determined by those in another component. Dually, the requirement that variables in P^o can only be assigned the value T(rue) by any transition of the component allows them to determine the values of variables in another component. Similarly to the values of the variables in P^i, those of the variables in C^i are fixed when combining the module with another; this is the reason for which only the variables in $C \setminus C^i$ can occur in the always constraints of the component.

Example 5. We now illustrate the notion of security-sensitive component by considering the TRW and the MDW. As said in Section 4.1, we want to reuse tasks $t1$ and $t5$ in both TRW and MDW. To do so, we split the specification of each workflow in four components C_1, C_{234}, C_{67}, and C_5 as

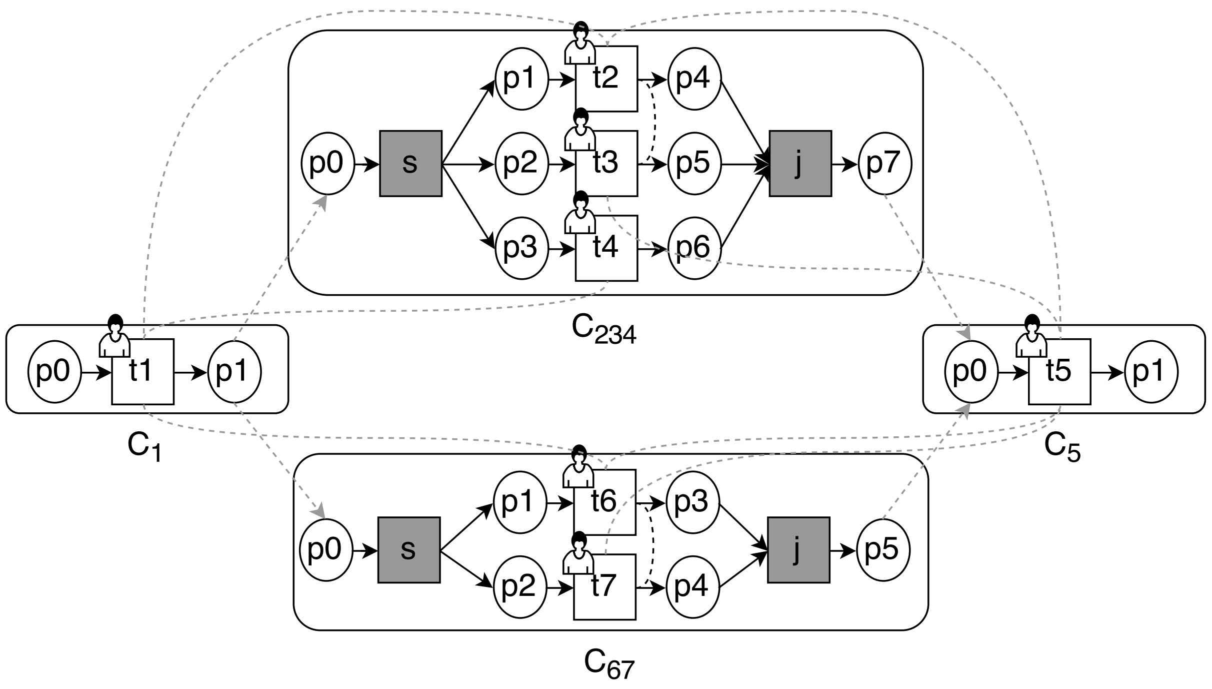


Figure 4.3: TRW and MDW as combinations of security-sensitive components

shown in Figure 4.3, where the sequential composition of C_1, C_{234}, and C_5 yields the TRW and that of C_1, C_{67}, and C_5 gives the MDW.

The figure shows the extended Petri nets representing the four components and how they are connected: circles represent places, rectangles with a man icon transitions to be executed under the responsibility of users, rectangles without the icon transitions not needing human intervention, (black) dashed lines represent SoD constraints between tasks belonging to the same component, (gray) dashed lines SoD constraints between tasks belonging to distinct components, (black) solid arrows the control flow in the same component, and (gray) dashed arrows the control flow between two components.

Notice that the control flow between two components is outside of the semantics of extended Petri nets. For example, a token in place $p0$ of C_1 goes to $p1$ of C_1 after the execution of $t1$ and, at the same time a token is put in place $p0$ of C_{234} because of the (gray) dashed arrow from $p1$ in C_1

to $p0$ in C_{234} representing an inter execution constraint. When the token is in $p0$, the system executes the split transition s in C_{234} that removes the token from $p0$ and puts one in $p1$, $p2$, and $p3$ so that $t2$, $t3$, and $t4$ in C_{234} become enabled. Notice that the execution of $t2$ is constrained by a SoD constraint from task $t1$ in component C_1 (dashed arrow between $t1$ in C_1 and $t2$ in C_{234}): this means that the user who has executed $t1$ in C_1 cannot execute also $t2$ in C_{234}.

We now show how to formalize the components depicted in Figure 4.3 by defining $\mathcal{C}_1 = (S_1, Int_1)$, $\mathcal{C}_5 = (S_5, Int_5)$, $\mathcal{C}_{234} = (S_{234}, Int_{234})$, and $\mathcal{C}_{67} = (S_{67}, Int_{67})$ where $S_y = ((P_y, D_y, A_y, H_y, C_y), Tr_y, B_y)$, and $Int_y = (A_y, P^i_y, P^o_y, H^o_y, C^i_y)$ for $y = 1, 5, 234, 67$. For components $\mathcal{C}_1$ and $\mathcal{C}_5$, we set

$$\begin{array}{l} P_y := \{p0_y, p1_y\},\ D_y := \{d_{ty}\},\ A_y := \{a_{ty}\},\ H_y := \{h_{ty}\}, C_y := \{c^i_{ty}\}, \\ B_y := \emptyset,\ P^i_y := \{p0_y\},\ P^o_y := \{p1_y\},\ H^o_y := \{h_{ty}\}. \end{array}$$

for $y = 1, 5$, and take

$$\begin{array}{l} Tr_1 := \{t1(u) : p0_1 \wedge \neg d_{t1} \wedge a_{t1}(u) \rightarrow p0_1, p1_1, d_{t1}, h_{t1}(u) := F, T, T, T\}, \\ Tr_5 := \{t5(u) : p0_5 \wedge \neg d_{t5} \wedge a_{t5}(u) \wedge c^i_{t5}(u) \rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad p0_5, p1_5, d_{t5}, h_{t5}(u) := F, T, T, T\}, \\ C^i_1 := \emptyset,\ C^i_5 := \{c^i_{t5}\}. \end{array}$$

According to the transition in Tr_1, task $t1$ is enabled when there is a token in place $p0_1$ (place $p0$ of component $\mathcal{C}_1$ in Figure 4.3), $t1$ has not been already executed ($\neg d_{t1}$) and there exists a user u capable of executing $t1$ ($a_{t1}(u)$). The effect of executing such a transition is to move the token from $p0_1$ to $p1_1$ (places $p0$ and $p1$ of component $\mathcal{C}_1$ in Figure 4.3, respectively), set d_{t1} to true meaning that $t1$ has been executed, and recording that $t1$ has been executed by u.

The interface of each component is the following: $p0_y$ is the input place, $p1_y$ is the output place, and the history variable h_{ty} can be used to constrain the execution of tasks in other components (for instance of $t2$ in the TRW as $t1$ and $t2$ are involved in a SoD, shown by the gray dashed line between the two tasks in Figure 4.3). Notice that the execution of task $t1$ cannot be constrained by the execution of tasks in other components (thus $C_1^i := \emptyset$) since $t1$ is always executed before all other tasks and cannot possibly be influenced by their execution.

The definition of the transition in Tr_5 is similar to that in Tr_1 except for the fact that the execution of task $t5$ can be constrained by the execution of tasks in other components (thus $C_5^i := \{c_{t5}^i\}$) since $t5$ is always executed after all other tasks and can be influenced by their execution. In particular, c_{t5}^i will be defined so as to satisfy the SoD constraints between $t5$ and $t2$ or $t3$ for the TRW and $t6$ or $t7$ for MDW.

For component $\mathcal{C}_{234}$, we set

$$P_{234} := \{py_{234} | y = 0, \ldots, 7\},\ D_{234} := \{s_{234}, j_{234}, d_{ty} | y = 2, 3, 4\},$$
$$A_{234} := \{a_{ty} | y = 2, 3, 4\},\ H_{234} := \{h_{ty} | y = 2, 3, 4\},$$
$$C_{234} := \{c_{t2}, c_{t3}, c_{ty}^i | y = 2, 3, 4\},$$
$$B_{234} := \{\forall u.c_{t2}(u) \Leftrightarrow \neg h_{t3}(u), \forall u.c_{t3}(u) \Leftrightarrow \neg h_{t2}(u)\},$$
$$Tr_{234} := \left\{ \begin{array}{lll} s_{234} & : & p0_{234} \wedge \neg d_s \rightarrow \\ & & p0_{234}, p1_{234}, p2_{234}, p3_{234}, d_{s_{234}} := F, T, T, T, T \\ t2(u) & : & p1_{234} \wedge \neg d_{t2} \wedge a_{t2}(u) \wedge c_{t2}(u) \wedge c_{t2}^i(u) \rightarrow \\ & & p1_{234}, p4_{234}, d_{t2}, h_{t2}(u) := F, T, T, T \\ t3(u) & : & p2_{234} \wedge \neg d_{t3} \wedge a_{t3}(u) \wedge c_{t3}(u) \wedge c_{t3}^i(u) \rightarrow \\ & & p2_{234}, p5_{234}, d_{t3}, h_{t3}(u) := F, T, T, T \\ t4(u) & : & p3_{234} \wedge \neg d_{t4} \wedge a_{t4}(u) \wedge c_{t4}^i(u) \rightarrow \\ & & p3_{234}, p6_{234}, d_{t4}, h_{t4}(u) := F, T, T, T \\ j_{234} & : & p4_{234} \wedge p5_{234} \wedge p6_{234} \wedge \neg d_j \rightarrow \\ & & p4_{234}, p5_{234}, p6_{234}, p7_{234}, d_{j_{234}} := F, F, F, T, T \end{array} \right\}$$
$$P_{234}^i := \{p0_{234}\},\ P_{234}^o := \{p7_{234}\},\ H_{234}^o := \{h_{t2}, h_{t3}\},\ C_{234}^i := \{c_{t2}^i, c_{t4}^i\}.$$

Transitions s_{234} and j_{234} (corresponding to the rectangles labeled s and j of component $\mathcal{C}_{234}$ in Figure 4.3) model the parallel composition of tasks

$t2$, $t3$, and $t4$ in the TRW and the MDW. Since no human intervention is needed, the enabling conditions for the authorization constraint of both transitions are omitted. Tasks $t2$ and $t3$ are involved in a SoD constraint (cf. the dashed lines labeled by $\neq$ between $t2$ and $t3$ in Figure 4.3). For this reason, their enabling conditions contain $c_{t2}(u)$ and $c_{t3}(u)$ which are defined in B_{234} so as to prevent the execution of $t2$ and $t3$ by the same users: to execute $t3$ ($t2$, resp.), user u must be such that $\neg h_{t2}(u)$ ($\neg h_{t3}(u)$, resp.), i.e. u should have not executed $t2$ ($t3$, resp.). Transitions $t2$, $t3$, and $t4$ in Tr_{234} have enabling conditions that contain $c^i_{t2}(u)$, $c^i_{t3}(u)$, and $c^i_{t4}(u)$ which will be defined so as to satisfy the SoD constraints in which the tasks are involved (cf. the gray dashed lines across the rectangles in Figure 4.3).

The definition of component $\mathcal{C}_{67}$ is quite similar (albeit simpler) to that of $\mathcal{C}_{234}$:

$$
\begin{array}{l}
P_{67} := \{py_{67} | y = 0, \ldots, 5\},\ D_{67} := \{s_{67}, j_{67}, d_{ty} | y = 6, 7\},\\
A_{67} := \{a_{ty} | y = 6, 7\},\ 5H_{67} := \{h_{ty} | y = 6, 7\},\ C_{67} := \{c_{ty}, c^i_{ty} | y = 6, 7\},\\
B_{67} := \{\forall u. c_{t6}(u) \Leftrightarrow \neg h_{t7}(u), \forall u. c_{t7}(u) \Leftrightarrow \neg h_{t6}(u)\}\\
Tr_{67} := \left\{ \begin{array}{lcl}
s_{67} & : & p0_{67} \wedge \neg d_s \rightarrow p0_{67}, p1_{67}, p2_{67}, d_{s_{67}} := F, T, T, T\\
t6(u) & : & p1_{67} \wedge \neg d_{t6} \wedge a_{t6}(u) \wedge c_{t6}(u) \wedge c^i_{t6}(u) \rightarrow\\
 & & p1_{67}, p3_{67}, d_{t6}, h_{t6}(u) := F, T, T, T\\
t7(u) & : & p2_{67} \wedge \neg d_{t7} \wedge a_{t7}(u) \wedge c_{t7}(u) \wedge c^i_{t7}(u) \rightarrow\\
 & & p2_{67}, p4_{67}, d_{t7}, h_{t7}(u) := F, T, T, T\\
j_{67} & : & p3_{67} \wedge p4_{67} \wedge \neg d_j \rightarrow\\
 & & p3_{67}, p4_{67}, p5_{67}, d_{j_{67}} := F, F, T, T
\end{array} \right\}\\
P^i_{67} := \{p0_{67}\},\ P^o_{67} := \{p5_{67}\},\ H^o_{67} := \{h_{t6}, h_{t7}\},\ C^i_{67} := \{c^i_{t6}\}.
\end{array}
$$

Section 4.3 below explains how components $\mathcal{C}_1$, $\mathcal{C}_{234}$, $\mathcal{C}_{67}$, and $\mathcal{C}_5$ can be "glued together" to build the TRW and the MDW. □

Semantics of a security-sensitive component

The notion of symbolic security-sensitive transition system introduced here is equivalent to that in Chapter 3; the only difference being the presence of the authorization constraint variables in C together with the always constraints in B.

It is easy to see that, given a transition system $((P, D, A, H, C), \mathit{Tr}, B)$, it is always possible to eliminate the variables in C occurring in B from the conditions of transitions in Tr by using (4.2): it is sufficient to replace each occurrence of $v(u)$ with hst. Let $[[\mathit{tr}]]_B$ denote the transition obtained from tr by exhaustively replacing the variables in C that also occur in B as explained above. Since no variable in C may occur in the update of a transition and in the enabling condition for the execution constraint of a transition, by abuse of notation, we apply the operator $[[\cdot]]_B$ to the enabling condition for the authorization constraint of tr. The substitution process eventually terminates since in hst there is no occurrence of variables in the finite set C, only the variables in H may occur.

The possibility of eliminating the variables in C allows us to give the semantics of the class of (symbolic) security-sensitive transition systems considered here by using the notion of weakest liberal precondition (wlp) as done in Chapter 3. The intuition is that computing a wlp with respect to the transitions in Tr and the always constraints in B is equivalent to computing that with respect to $[[\mathit{Tr}]]_B$. Formally, we define

$$\mathsf{wlp}(\mathit{Tr}, B, K) \;:=\; \bigvee_{\mathit{tr}\in \mathit{Tr}} (\mathit{en}_{\mathrm{EC}} \wedge [[\mathit{en}_{\mathrm{Auth}}]]_B \wedge K[\mathit{act}_{\mathrm{EC}} || \mathit{act}_{\mathrm{Auth}}]) \quad (4.3)$$

where B is a set of always constraints, tr is of the form (4.1), K is a predicate over $P \cup D \cup A \cup C$, and $K[\mathit{act}_{\mathrm{EC}} || \mathit{act}_{\mathrm{Auth}}]$ denotes the predicate obtained from K by substituting

- each variable $v \in P \cup D$ with the value b when the assignment $v := b$ is in $\mathit{act}_{\mathrm{EC}}$ and

- each variable $v \in H$ with $\lambda x.\mathit{if}\ x = u\ \mathit{then}\ b\ \mathit{else}\ v(x)$ when $v(x) := b$ is in act_{Auth} for b a Boolean value.

It is easy to show that $\mathsf{wlp}(Tr, B, K)$ is $\bigvee_{tr \in Tr} \mathsf{wlp}(tr, B, K)$. When Tr is a singleton containing one symbolic transition tr, we write $\mathsf{wlp}(tr, B, K)$ instead of $\mathsf{wlp}(\{tr\}, B, K)$.

Definition 7. *A* symbolic behavior *of a security-sensitive transition system $S = ((P, D, A, H, C), Tr, B)$ is a sequence of the form $K_0 \xrightarrow{tr_0} K_1 \xrightarrow{tr_1} \cdots \xrightarrow{tr_{n-1}} K_n$ where K_i is a predicate over $P \cup D \cup A \cup C$ and tr_i is a symbolic transition such that K_i is logically equivalent to $\mathsf{wlp}(tr_i, B, K_{i+1})$ for $i = 0, \ldots, n-1$.*

The semantics of the security-sensitive transition system S is the set of all possible symbolic behaviors. The semantics of a security-sensitive component (S, Int) is the set of all possible symbolic behaviors of the security-sensitive transition system S.

Example 6. We consider component $\mathcal{C}_5$ (cf. Example 5) and compute the wlp with respect to $t5(u)$ (in the set Tr_5 of transitions) for the following predicate $\neg p0_5 \wedge p1_5 \wedge d_{t5}$ characterizing the set of final states of $\mathcal{C}_5$, i.e., those states in which task $t5$ has been executed and there is just one token in place $p1_5$.

By using definition (4.3) above, we obtain $p0_5 \wedge \neg d_{t5} \wedge a_{t5}(u) \wedge c^i_{t5}(u)$, which identifies those states in which there is a token in place $p0_1$, task $t1$ has not yet been executed, and user u has the right to execute $t1$ and authorization constraints imposed by other components are satisfied (e.g., the SoD constraint between $t5$ and $t2$ in $\mathcal{C}_{234}$ for the TRW). □

4.3 Gluing together security-sensitive components

We now show how components can be combined together in order to build other, more complex, components. For $l = 1, 2$, let (S_l, Int_l) be a symbolic security-sensitive component where $Int_l = (A, P^i_l, P^o_l, H^o_l, C^i_l)$ and $S_l = ((P_l, D_l, A_l, H_l, C_l), Tr_l, B_l)$ is such that P_1 and P_2, D_1 and D_2, A_1 and A_2, H_1 and H_2, C_1 and C_2 are pairwise disjoint sets. Furthermore, let $G = G_{\text{EC}} \cup G_{\text{Auth}}$ be a set of *gluing assertions over* Int_1 *and* Int_2, where

- G_{EC} is a set of formulae of the form $p^i \Leftrightarrow p^o$ for $p^i \in P^i_k$ and $p^o \in P^o_j$, called *inter execution constraints*, and
- G_{Auth} is a set of always constraints in which only the variables in $C^i_k \cup H^o_j$ may occur,

for $k, j = 1, 2$ and $k \neq j$.

Intuitively, the gluing assertions in G specify inter component constraints; those in G_{EC} define how the control flow is passed from one component to another, whereas those in G_{Auth} specify authorization constraints across components, i.e., how the fact that a task in a component is executed by a certain user constrains the execution of a task in another component by a subset of the users entitled to do so.

Definition 8. *The symbolic security-sensitive component (S, Int) obtained by gluing (S_1, Int_1) and (S_2, Int_2) together with G, in symbols $(S, Int) = (S_1, Int_1) \oplus_G (S_2, Int_2)$, is defined as $S = ((P, D, A, H, C), Tr, B)$ and $Int = (A, P^i, P^o, H^o, C^i)$, where*

- *$P = P_1 \cup P_2$, $D = D_1 \cup D_2$, $A = A_1 \cup A_2$, $H = H_1 \cup H_2$, $C = C_1 \cup C_2$,*
- *$Tr := [Tr_1]_{G_{EC}} \cup [Tr_2]_{G_{EC}}$ where $[Tr_j]_{G_{EC}} := \{[tr_j]_{G_{EC}} | tr_j \in Tr_j\}$ and $[tr_j]_{G_{EC}}$ is obtained from tr_j by adding the assignment $p^i := b$ if p^i is in P^i_j, there exists an inter execution constraint of the form $p^i \Leftrightarrow p^o$ in G_{EC}, p^o is in P^o_k, and $p^o := b$ is among the parallel assignments of tr_j; otherwise, tr_j is returned unchanged, for $j, k = 1, 2$ and $j \neq k$,*

- $B = B_1 \cup B_2 \cup G_{Auth}$,
- $P^i = \{p \in (P^i_1 \cup P^i_2) | p \textit{ does not occur in } G_{EC}\}$,
- $P^o = \{p \in (P^o_1 \cup P^o_2) | p \textit{ does not occur in } G_{EC}\}$,
- $H^o = H^o_1 \cup H^o_2$, *and*
- $C^i = \{c \in (C^i_1 \cup C^i_2) | c \textit{ does not occur in } G_{Auth}\}$.

The definition is well formed since S is obviously a security-sensitive transition system and *Int* satisfies all the structural constraints at page 73.

Example 7. Let us consider components $\mathcal{C}_1$ and $\mathcal{C}_{234}$ of Example 5. We glue them together by using the following set $G = G_{\mathrm{EC}} \cup G_{\mathrm{Auth}}$ of gluing assertions where $G_{\mathrm{EC}} := \{p1_1 \Leftrightarrow p0_{234}\}$ and $G_{\mathrm{Auth}} := \{\forall u.c^i_{t2}(u) \Leftrightarrow \neg h_{t1}(u), \forall u.c^i_{t4}(u) \Leftrightarrow \neg h_{t1}(u)\}$. The inter execution constraint in G_{EC} corresponds to the dashed arrow connecting $p1$ in component $\mathcal{C}_1$ ($p1_1$) to $p0$ in component $\mathcal{C}_{234}$ ($p0_{234}$) in Figure 4.3. The always constraints in G_{Auth} formalize the dashed lines linking task $t1$ of component $\mathcal{C}_1$ to tasks $t2$ and $t4$ of component $\mathcal{C}_{234}$. The component obtained by gluing $\mathcal{C}_1$ and $\mathcal{C}_{234}$ together with G (in symbols, $\mathcal{C}_1 \oplus_G \mathcal{C}_{234}$) is such that

- its set of transitions contains all transitions in Tr_{234} plus the transition in Tr_1 modified to take into account the inter execution constraint in G_{EC}, i.e.

$$t1(u) : p0_1 \wedge \neg d_{t1} \wedge a_{t1}(u) \rightarrow$$
$$p0_1, p1_1, d_{t1}, h_{t1}(u) := F, T, T, T || p0_{234} := T$$

  ensuring that when the token is put in $p1_1$ it is also put in $p0_{234}$ (in this way, we can specify how the control flow is transferred from $\mathcal{C}_1$ to $\mathcal{C}_{234}$);

- its set of always constraints contains all the constraints in B_1 and B_{234} plus those in G_{Auth} so that the SoD constraints between task $t1$ in $\mathcal{C}_1$ and tasks $t2$ and $t4$ in $\mathcal{C}_{234}$ are added;

- if its interface is (A, P^i, P^o, H^o, C^i), then $P^i := \{p1_1\}$ since $p0_{234}$ occurs in G_{EC}, $P^0 := \{p7_{234}\}$ since $p1_1$ occurs in G_{EC}, and $C^i := \emptyset$ since both c^i_{t2} and c^i_{t4} occur in G_{Auth}.

Notice that $\mathcal{C}_1 \oplus_G \mathcal{C}_{234}$ can be combined with $\mathcal{C}_5$ to form a component corresponding to the TRW in Figure 3.1. This is possible by considering the following set $G' = G'_{\text{EC}} \cup G'_{\text{Auth}}$ of gluing assertions where $G'_{\text{EC}} := \{p7_{234} \Leftrightarrow p0_5\}$ and $G'_{\text{Auth}} := \{\forall u. c^i_{t5}(u) \Leftrightarrow \neg h_{t2}(u) \wedge \neg h_{t3}(u)\}$. The inter execution constraint in G'_{EC} corresponds to the dashed arrow connecting $p7$ in component $\mathcal{C}_{234}$ ($p7_{234}$) to $p0$ in component $\mathcal{C}_5$ ($p0_5$) in Figure 4.3. The always constraint in G_{Auth} formalizes the dashed lines linking task $t5$ of $\mathcal{C}_5$ with tasks $t2$ and $t3$ of $\mathcal{C}_{234}$. □

We now illustrate the computation of the wlp with respect to the transitions of a composed component by means of an example.

Example 8. Let us consider $(\mathcal{C}_1 \oplus_G \mathcal{C}_{234}) \oplus_{G'} \mathcal{C}_5$ of Example 7 and the predicates

$$\begin{array}{rcl} K_1 & := & \neg p0_1 \wedge d_{t1} \\ K_{234} & := & \bigwedge_{i=0,\ldots,6} \neg pi_{234} \wedge d_{t2} \wedge d_{t3} \wedge d_{t4} \wedge d_{s_{234}} \wedge d_{j_{234}} \\ K_5 & := & \neg p0_5 \wedge p1_5 \wedge d_{t5} \end{array}$$

whose conjunction K characterizes the final states of the TRW, i.e. those situations in which all tasks have been executed and there is just one token in place $p1_5$ (notice that K_{234} does not mention $p7_{234}$ whose value is implied by K_5 and the inter execution constraints $p7_{234} \Leftrightarrow p0_5$ and similarly K_1 does not mention $p1_1$ whose value is implied by K_{234} and the inter execution constraint in $p1_1 \Leftrightarrow p0_{234}$).

Now we compute $\mathsf{wlp}(t5, B, K)$ where B is the union of B_1, B_{234}, B_5, G_{Auth}, and G'_{Auth} given in Example 7 by using (4.3):

$$K_1 \wedge K_{234} \wedge (p0_5 \wedge \neg d_{t5} \wedge a_{t5}(u) \wedge \neg h_{t2}(u) \wedge \neg h_{t3}(u)) . \tag{4.4}$$

Notice how K_1 and K_{234} have not been modified since the parallel updates of $t5$ do not mention any of the state variables in $\mathcal{C}_1$ and $\mathcal{C}_{234}$ but only those of $\mathcal{C}_5$, namely $p0_5$ and d_{t5}.

To illustrate how the computation of wlp takes into account the transfer of the control flow from one component to another, let us compute the wlp of (4.4) with respect to transition j in component $\mathcal{C}_{234}$. According to the definition of composition of components, transition j_{234} becomes

$$\begin{aligned} j^*_{234} \ : \ & p4_{234} \wedge p5_{234} \wedge p6_{234} \wedge \neg d_{j_{234}} \rightarrow \\ & p4_{234}, p5_{234}, p6_{234}, p7_{234}, d_{j_{234}} := F, F, F, T, T || p0_5 := T\,. \end{aligned}$$

Notice the added assignment $p0_5 := T$ to take into account the inter execution constraint in G'_{EC} (see Example 7) ensuring that when the token is put in $p7_{234}$, it is also put in $p0_5$. By using (4.3), we have that $\mathsf{wlp}(j^*_{234}, B, (4.4))$ is

$$K_1 \wedge \left[\begin{array}{l} \neg p0_{234} \wedge \neg p1_{234} \wedge \neg p2_{234} \wedge \\ \neg p3_{234} \wedge p4_{234} \wedge p5_{234} \wedge p6_{234} \wedge \\ \neg d_{t2} \wedge \neg d_{t3} \wedge \neg d_{t4} \wedge d_{s_{234}} \wedge \neg d_{j_{234}} \end{array}\right] \wedge \left(\begin{array}{l} \neg d_{t5} \wedge a_{t5}(u) \wedge \\ \neg h_{t2}(u) \wedge \neg h_{t3}(u) \end{array}\right) \quad (4.5)$$

Notice how K_1 is left unmodified since it describes the state of component $\mathcal{C}_1$ and no gluing assertions involve state variables of $\mathcal{C}_1$ and those in the update of j^*_{234}. K_{234} instead is modified substantially (see the predicate in square brackets) since j_{234} is a transition of component $\mathcal{C}_{234}$, while the remaining part of (4.5) is almost identical to the formula between parentheses in (4.4) except for the deletion of $p0_5$ because of the additional assignment $p0_5 := T$ in j^*_{234}, introduced to take into account the inter execution constraint in G'_{Auth}.

An alternative way of computing $\mathsf{wlp}(j^*_{234}, B, (4.4))$ is the following. Observe that the value of $p7_{234}$ is fixed to T because of the inter execution constraint $p0_5 \Leftrightarrow p7_{234}$ in G_{Auth} and the fact that (4.4) implies that $p0_5$ is T. Thus, we can consider the predicate $K_{234} \wedge p7_{234}$ and then compute

$\mathsf{wlp}(j_{234}, B_{234}, K_{234} \wedge p7_{234})$ which is the predicate in square brackets of (4.5). By taking the conjunction of this formula with K_1 and the predicate obtained by deleting $p0_5$ from $\mathsf{wlp}(t5, B_5 \cup G_{\text{Auth}}, K_5)$ in which we delete $p0_5$ (because (4.4) implies that $p0_5$ is T) thereby obtaining the predicate between parentheses in (4.4), we derive (4.5) as before. □

The last paragraph of the example suggests a modular approach to computing wlp's. It is indeed possible to generalize the process described above and derive a modularity result for computing the wlp of a complex component by using the wlp's of its components by taking into account the gluing assertions. We do not do this here because it is not central to the applications of the notion of component discussed in Section 4.4 below.

Theorem 5. *Let (S_k, Int_k) be a symbolic security-sensitive component for $k = 1, 2, 3$, $G_{1,2}$ be a set of gluing assertions over Int_1 and Int_2, and $G_{2,3}$ be a set of gluing assertions over Int_2 and Int_3. Then,*

Commutativity: $(S_1, Int_1) \oplus_{G_{1,2}} (S_2, Int_2) = (S_2, Int_2) \oplus_{G_{1,2}} (S_1, Int_1)$ *and*

Associativity: $((S_1, Int_1) \oplus_G (S_2, Int_2)) \oplus_G (S_3, Int_3) = (S_1, Int_1) \oplus_G ((S_2, Int_2) \oplus_G (S_3, Int_3))$ *for $G = G_{1,2} \cup G_{2,3}$.*

Proof. (Sketch) The proof is straightforward and based on Definition 8 and the commutativity and associativity of set union, since the definition of $\oplus_G$ uses the union of the sets of variables in each of the security-sensitive components and in G. Notice that the associativity property above is expressed by taking into account the union of the gluing assertions over the interfaces of the reusable systems being combined. □

Example 9. Recall the components of Example 7. Because of Theorem 5, we have that the TRW can be expressed as $\mathcal{C}_1 \oplus_{G''} \mathcal{C}_{234} \oplus_{G''} \mathcal{C}_5$ for $G'' = G \cup G'$ where G, G' have been defined in Example 7.

Notice that, despite the commutativity of the operator $\oplus$, the task in $\mathcal{C}_1$ will always be executed before all tasks in components $\mathcal{C}_{234}$ and $\mathcal{C}_5$ because of the gluing assertions in G''. Thus, the component $\mathcal{C}_{234} \oplus_{G''} \mathcal{C}_1 \oplus_{G''} \mathcal{C}_5$ obtained by considering the components in a different order is equivalent to TRW. $\square$

4.4 Modular synthesis of run-time monitors

In Chapter 3, we have shown how to automatically derive a monitor capable of solving the run-time version of the WSP for a security-sensitive transition system.

As already discussed in Section 4.2 ("**Semantics of a security-sensitive component**"), the notion of security-sensitive transition system introduced here and that in Chapter 3 are equivalent. In particular, given a security-sensitive transition system $((P, D, A, H, C), Tr, B)$ we can derive an equivalent security-sensitive transition system of the form $((P, D, A', H, \emptyset), \{[[tr]]_B | tr \in B\}, \emptyset)$, which is precisely a security-sensitive transition system of Chapter 3, where A' contains the variables in A and those in C which are not mentioned in B.

Let $\mathcal{RM}$ be the procedure which takes as input a security-sensitive transition system $S = ((P, D, A, H, C), Tr, B)$, applies the transformation above, and then the procedure for the synthesis of run-time monitors described in Chapter 3, which returns a Datalog program $\mathcal{RM}(S)$ defining a predicate $can\_do(u, t)$ such that user u can execute task t and the workflow can successfully terminate iff $can\_do(u, t)$ is a logical consequence (in the sense of Datalog) of $\mathcal{RM}(S) \cup \mathcal{P} \cup \mathcal{H}$ (in symbols $\mathcal{RM}(S), \mathcal{P}, \mathcal{H} \models can\_do(u, t)$), where $\mathcal{P}$ is a Datalog program defining the meaning of the predicates in A (i.e. the authorization policy) and $\mathcal{H}$ is a set of *history facts* of the form $h_t(u)$, recording the fact that user u has executed task t.

We now show how to reuse $\mathcal{RM}$ for the modular construction of run-time monitors for the WSP, i.e., we build a monitor for a composite component by combining those for their constituent components. Let $G = G_{\text{EC}} \cup G_{\text{Auth}}$ be a set of gluing assertions where G_{EC} is a set of inter execution constraints and G_{Auth} a set of always constraints over an interface (A, P^i, P^o, H^o, C^i), then $\langle G \rangle := \langle G_{\text{EC}} \rangle \cup \langle G_{\text{Auth}} \rangle$, where $\langle G_{\text{EC}} \rangle := \{p^i \leftarrow p^o | p^i \Leftrightarrow p^o \in G_{\text{EC}}\}$ and $\langle G_{\text{Auth}} \rangle := \{c^i(u) \leftarrow \mathit{hst}^i(u) | \forall u. c^i(u) \Leftrightarrow \mathit{hst}^i(u) \in G_{\text{Auth}}\}$. Intuitively, the shape of the Datalog clauses in $\langle G_{\text{EC}} \rangle$ models how the execution flow is transferred from a component (that with an output place) to the other (that with an input place).

Theorem 6. *Let (S_k, Int_k) be a symbolic security-sensitive component, $S_k = ((P_k, D_k, A_k, H_k, C_k), Tr_k, B_k)$, $\mathcal{H}_k$ is a set of (history) facts over H_k, and $\mathcal{P}_k$ a Datalog program (for the authorization policy) over A_k, for $k = 1, 2$. If G is a set of gluing assertions over Int_1 and Int_2, then $\mathcal{RM}(S), \mathcal{H}_1, \mathcal{H}_2, \mathcal{P}_1, \mathcal{P}_2 \models can\_do(u, t)$ iff $\mathcal{RM}(S_1), \mathcal{H}_1, \mathcal{P}_1, \langle G \rangle, \mathcal{RM}(S_2), \mathcal{H}_2, \mathcal{P}_2 \models can\_do(u, t)$, where $(S, Int) = (S_1, Int_1) \oplus_G (S_2, Int_2)$.*

Proof. (Sketch) The idea underlying the proof of this theorem is that the monitors for the components are computed by considering any possible values for the variables in their interfaces. The additional constraints in the gluing assertions simply consider a subset of all these values by specifying how the execution flow goes from one component to the other and how the authorization constraints across components further constrain the possible executions of a component depending on which users have executed certain tasks in the other. $\square$

Chapter 5

Assisting the deployment of security-sensitive workflows[1]

As discussed in Chapter 4, there is a trend in BPM of collecting and reusing business process models [145, 50, 71]. At deployment time, it is crucial for customers reusing a template from the library to understand whether it can be successfully instantiated with the authorization policy adopted by their organization.

This means that customers want to scrutinize concrete *execution scenarios* showing the termination of the instantiated business process model by giving evidence that some of the employees can successfully execute the various tasks in the workflow. Variants of this basic problem may be of interest to assist customers in deploying security-sensitive workflows under specific operational conditions.

A first refinement is to focus on those scenarios that can be executed by a given "small" set of users, called *minimal user base* in the literature [42]. This would enable organizations to assess the likelihood of emergencies or extraordinary situations due to, e.g., employee absences.

A second variant of the problem of finding execution scenarios is to discover those that are *resilient* to the absence of users, i.e., whether the

[1]Parts of this chapter were previously published in [53]

termination of the workflow is guaranteed even if a certain number of users (regardless of which concrete users) is unavailable to execute tasks for a given execution [154].

A refinement of all previous problems consists of considering only those scenarios satisfying some additional conditions on the execution of a workflow, such as only a given user can execute a task, the Boolean value of a conditional is true (or false), the order of execution of a set of parallel tasks is fixed, or arbitrary combinations of such constraints.

In the following, we call these types of problems *Scenario Finding Problems (SFPs)*. In this Chapter, we give the following contributions:

- we give precise statements of four SFPs together with a discussion of their relationships with the WSP (Section 5.2); and
- we describe methods to automatically solve the four SFPs by adapting the technique for the synthesis of run-time monitors for the WSP (Section 5.3).

5.1 Preliminaries

Let T be a finite set of tasks and U a finite set of users. An *execution scenario* (or, simply, a *scenario*) is a finite sequence of pairs of the form (t, u), written as $t(u)$, where $t \in T$ and $u \in U$. The intuitive meaning of a scenario $\eta = t_1(u_1), \ldots, t_n(u_n)$ is that task t_i is executed before task t_j for $1 \leq i < j \leq n$ and that task t_k is executed by user u_k for $k = 1, \ldots, n$.

A *workflow* $W(T, U)$ is a set of scenarios. Among the scenarios in a workflow, we are interested in those that describe successfully terminating executions in which users execute tasks satisfying the authorization constraints and the authorization policy. Since the notion of successful termination depends on the definition of the workflow (e.g., in case of a conditional choice, we will have two acceptable execution sequences according to the Boolean value of the condition), in the following we focus only on the authorization policy and the authorization constraints while assuming that all the scenarios in the workflow characterize successfully terminating behaviors.

Given a workflow $W(T,U)$, an *authorization relation* TA is a subset of $U \times T$. Intuitively, $(u,t) \in TA$ means that u is authorized to execute task t. We say that a scenario η of a workflow $W(T,U)$ is *authorized* according to TA iff (u,t) is in TA for each $t(u)$ in η.

An *authorization constraint* over a workflow $W(T,U)$ is a tuple (t_1,t_2,ρ) where $t_1,t_2 \in T$ and ρ is a subset of $U \times U$. (It is possible to generalize authorization constraints to the form (T_1,T_2,ρ) where T_1,T_2 are sets of tasks as done in, e.g., [40]. We do not do this here for the sake of simplicity.) For instance, a SoD constraint between tasks t and t' can be formalized as $(t,t',\neq)$ with $\neq$ being the relation $\{(u,u')|u,u' \in U \text{ and } u \neq u'\}$. A scenario η of $W(T,U)$ *satisfies* the authorization constraint (t_1,t_2,ρ) over $W(T,U)$ iff there exist $t_1(u_1)$ and $t_2(u_2)$ in η such that $(u_1,u_2) \in \rho$. Let C be a (finite) set of authorization constraints, a scenario η satisfies C iff η satisfies c, for each c in C. A scenario η of a workflow $W(T,U)$ is *eligible according to a set C of authorization constraints* iff η satisfies C.

We use the TRW introduced in Chapter 3 to illustrate the main notions in this Chapter.

Example 10. A simple situation in which the TRW in Example 1 can be deployed is a tiny organization with a set $U = \{a,b,c\}$ of three users and the following authorization policy $TA = \{(a,t1),(b,t1),(a,t2),(b,t2),(c,t2),(a,t3),(b,t3),(c,t3),(a,t4),(a,t5),$ $(b,t5),(c,t5)\}$. The organization would then like to know if there is an execution scenario that allows the process to terminate according to TA. Indeed, this is the case as shown by the following sequence of task-user pairs: $\eta = t1(b),t3(c),t4(a),t2(a),t5(b)$. It is easy to check that the tasks in η are executed so that the ordering constraints on task execution are satisfied, each user u in each pair $t(u)$ of η is authorized to execute t since $(u,t) \in TA$, and each SoD constraint is satisfied (e.g., tasks $t1$ and $t2$ are executed by the distinct users b and a, respectively). $\square$

5.2 Scenario finding problems

As described in previous Chapters, it is possible to pre-compute—once and for all—the set E of eligible scenarios, i.e., those scenarios satisfying the authorization constraints, associated to a security-sensitive workflow in a library. We are then left with the problem of finding those scenarios in E that are still computable as soon as an authorization policy becomes available (and possibly satisfying some additional properties).

We begin by defining a basic version of the scenario finding problem, whose definition (and solution) will help us in understanding (and solving) more complex problems.

Definition 9 (Basic Scenario Finding Problem (B-SFP))**.** *Given the finite set E of eligible scenarios according to a set C of authorization constraints in a workflow $W(T,U)$, return (if possible) a scenario $\eta \in E$ which is authorized according to a given authorization relation TA.*

Example 11. Let us consider the TRW. If $U = \{Alice, Bob, Charlie, Dave, Erin, Frank\}$ is the set of users, then the set E of eligible scenarios contains, among many others, the following elements:

$$\begin{aligned}
\eta_1 &= t_1(Alice), t_2(Bob), t_3(Charlie), t_4(Dave), t_5(Erin)\\
\eta_2 &= t_1(Bob), t_2(Alice), t_3(Charlie), t_4(Alice), t_5(Bob)\\
\eta_3 &= t_1(Bob), t_4(Charlie), t_2(Alice), t_3(Dave), t_5(Bob)
\end{aligned}$$

Now, let $TA = \{(Alice, t_1), (Bob, t_1), (Alice, t_2), (Bob, t_2), (Charlie, t_3), (Alice, t_4), (Dave, t_4), (Bob, t_5), (Erin, t_5)\}$ be the authorization relation, then η_1 and η_2 are solutions to the B-SFP, while η_3 is not because $(Dave, t_3) \notin TA$. □

A scenario η solving the B-SFP is also a solution of the WSP and vice versa. So, in principle, to solve the B-SFP for a workflow $W(T, U)$, a set C of authorization constraints, an authorization policy TA, and $\eta_e = t(u), t'(u'), \ldots$ an eligible scenario in E, we can reuse an algorithm $\mathcal{A}$ returning answers to the WSP as follows. Initially, we consider the task-user pair $t(u)$ in η_e and create a new authorization relation $TA_1 = TA|_{(u,t)}$ derived from TA by deleting all pairs $(x, t) \in TA$ with $x \neq u$. We invoke $\mathcal{A}$ on the WSP for $W(T, U)$, C, and TA_1: if $\mathcal{A}$ returns a scenario, this must have the form $t(u), \eta$ where η is some sequence of task-user pairs (notice that $t(u), \eta$ and η_e are guaranteed to have only $t(u)$ as a common prefix). Afterwards, we move to the task-user pair $t'(u')$ in η_e and run $\mathcal{A}$ on the WSP for $W(T, U)$, C, and $TA_2 = TA_1|_{(u',t')}$. If $\mathcal{A}$ returns a scenario, this must have the form $t(u), t'(u'), \eta'$ where η' is some sequence of task-user pairs (notice that $t(u), t'(u'), \eta'$ and η_e are guaranteed to have only $t(u), t'(u')$ as a common prefix). By repeating this process for each η_e in E, until all tasks in η_e are executed, we can check if it is also authorized according to TA (besides being eligible as η_e is in E). Overall, there are at most $O(\ell_{max} \cdot |E|)$ invocations to $\mathcal{A}$, where ℓ_{max} is the longest (in terms of number of task-user pairs occurring in it) scenario of E. Indeed, this is computationally expensive because of the complexity of the WSP [154] and, most importantly, we do not exploit the fact that the scenarios in E are eligible.

A better approach to solve the B-SFP is to consider each eligible scenario η_e in E and check if all task-user pairs in η_e are authorized according to TA. This means that there are at most $O(\ell_{max} \cdot |E|)$ invocations to the algorithm for checking membership of a user-task pair to TA. The complexity of such an algorithm depends on how TA has been specified. Policy languages are designed to make such a check efficient (e.g., linear or polynomial); this is in sharp contrast to the heavy computational cost of running $\mathcal{A}$. Below, we assume authorization policies to be specified in Datalog so that checking membership to TA is equivalent to answering a Datalog query, which is well-known to have polynomial-time (data) complexity [26]. Even though checking for membership to TA is efficiently performed, the overall computational complexity may be problematic since such a check must be repeated $O(\ell_{max} \cdot |E|)$ and $|E|$ may be large.

For instance, as we will show below, already for the simple TRW with $|U| = 6$ (as in Example 11), the cardinality of E is $19,080$. Intuitively, the larger the set U of users, the higher the cardinality of E. It is thus important to design a suitable data structure to represent the available set E of eligible scenarios which permits to design an efficient strategy to search through all scenarios and identify one that is authorized. We will see this in Section 5.3.1.

5.2.1 Minimal user-base scenarios

A refinement of B-SFP is to search for (eligible and) authorized scenarios in which a "minimal" set of users occurs. Formally, let η be a scenario in a workflow $W(T,U)$, the set of users occurring in η is $usr(\eta) = \{u|t(u) \in \eta\}$.

Following [42], we define a *minimal user base* of a workflow $W(T,U)$ to be a subset U' of the set U of users such that there exists a scenario η in $W(T,U)$ in which $usr(\eta) = U'$ and there is no scenario η' in $W(T,U)$ in which $usr(\eta')$ is a strict subset of U'.

Definition 10 (Minimal user-base scenario finding problem (MUB-SFP))**.** *Given the set E of eligible scenarios according to a set C of authorization* *constraints in a workflow $W(T, U)$, return (if possible) a scenario $\eta \in E$ which is authorized according to a given relation TA and such that the set $usr(\eta)$ of users occurring in η is a minimal user base.*

Example 12. Let us consider again the TRW together with the set U of users, the set E of eligible scenarios, and the authorization relation *TA* of Example 11. A solution to the MUB-SFP is $\eta_M = t_1(Bob), t_2(Alice), t_3(Charlie), t_4(Alice), t_5(Bob)$ and a minimal user base is $usr(\eta_M) = \{Alice, Bob, Charlie\}$. □

An approach derived from that of solving the B-SFP can also solve the MUB-SFP. We consider each eligible scenario η_e in E and check if all task-user pairs in η_e are authorized according to *TA*. We also maintain a variable η_M storing an eligible scenario in E such that η_M is authorized (according to *TA*) and $usr(\eta_M)$ is a candidate minimal user base. Initially, η_M is set to the empty sequence ϵ. If the eligible scenario η_e under consideration is authorized and $\eta_M \neq \epsilon$, then we compare the cardinalities of $usr(\eta_e)$ and $usr(\eta_M)$: if $|usr(\eta_e)| < |usr(\eta_M)|$, then $\eta_M \leftarrow \eta_e$; otherwise η_M is left unchanged. When $\eta_M = \epsilon$, we do not perform the comparison between the cardinalities of $usr(\eta_e)$ and $usr(\eta_M)$ and simply set η_M to η_e. Indeed, when all eligible scenarios in E have been considered, $usr(\eta_M)$ stores a minimal user base.

This process requires that there are $O(\ell_{max} \cdot |E|)$ invocations to the algorithm for checking membership of a user-task pair to *TA*. Although the complexity bounds of solving the B-SFP and the MUB-SFP are identical, the bound for the latter is tighter than the former. This is so because we always need to consider all eligible scenarios in E for the MUB-SFP whereas we can stop as soon as we find an authorized scenario for the B-SFP.

5.2.2 Resilient scenarios

Resiliency in workflow systems concerns the question of whether a workflow is still satisfiable even if a number of users is absent for an instance execution. It is an important property of workflows and authorization relations, since it indicates the likelihood of the workflow terminating even in adverse conditions.

There are several possible definitions of resiliency. The most obvious one is to fix a set U_a of absent users and check if the workflow can terminate. This can be easily reduced to a B-SFP problem by eliminating all those execution scenarios in $W(T, U)$ in which at least one task is executed by an absent user and by removing all pairs containing an absent user in U_a from the authorization policy TA. If the resulting B-SFP has a solution, then it can be considered as a witness of the fact that the workflow $W(T, U)$ is resilient to the absence of the users in U_a. This definition could be useful, for instance, when it is known beforehand that a set of users will be absent for some time, e.g., to attend a conference.

Another, more interesting, notion of resiliency has been proposed in [154] and amounts to checking whether a workflow is resilient to the absence of *all* subsets of users of a fixed size k. This means that no matter which concrete users are absent, as long as no more than a given number is absent (at any given time), the workflow remains satisfiable. We adapt this idea of resiliency to our context by identifying resilient sets of execution scenarios as follows.

Definition 11 (Statically k-resilient set of scenarios)**.** *Given a workflow $W(T, U)$, an authorization relation TA, and an integer $k > 0$, a set of scenarios H is statically resilient up to k absent users if and only if for every subset U' of U of size $t = |U| - k$, there is (at least) one scenario $\eta \in H$ that satisfies $W(T, U)$ under $TA|_{U'}$.*

Notice that resiliency is defined *up to k* since a k-resilient set of scenarios is clearly also $(k-1)$-resilient. Notice that it is not possible to limit resiliency to a single scenario because the second notion of resiliency discussed above must hold for all subsets of users of size t and this implies that if any one of the users u in a scenario η is removed, then η can no more be executed until the end. Instead, by considering a set H of execution scenarios, when we consider a certain subset U' of absent users of size t, we can hope to find a scenario in H containing no user in U' that can still be executed until the end.

We are now ready to introduce the problem of finding sets of resilient scenarios.

Definition 12 (Statically k-Resilient Scenario Finding Problem (SkR-SFP))**.** *Given the set E of eligible scenarios according to a set C of authorization constraints in a workflow $W(T, U)$ and an integer k, return (if possible) a set H of scenarios η which are authorized according to a given relation TA and statically resilient up to k absent users.*

Example 13. Let us consider again the TRW together with the set U of users, the set E of eligible scenarios, and the authorization relation TA of Example 11. In that case, there is no solution to the SkR-SFP, even with $k = 1$, because if Charlie is removed from U, there is no user authorized to perform $t3$ and if Alice or Bob are removed the same user has to perform $t1$ and $t2$, which violates the SoD constraint.

Now consider the same workflow and set of users, but with $TA' = TA \cup \{(Charlie, t1), (Charlie, t2), (Alice, t3)\}$. In this case, a solution to the

SkR-SFP, with $k = 1$ is $H = \{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5\}$ where

$$\begin{aligned}
\eta_1 &= t1(\mathit{Charlie}), t2(\mathit{Bob}), t3(\mathit{Charlie}), t4(\mathit{Dave}), t5(\mathit{Erin}) \\
\eta_2 &= t1(\mathit{Charlie}), t2(\mathit{Alice}), t3(\mathit{Charlie}), t4(\mathit{Dave}), t5(\mathit{Erin}) \\
\eta_3 &= t1(\mathit{Alice}), t2(\mathit{Bob}), t3(\mathit{Alice}), t4(\mathit{Dave}), t5(\mathit{Erin}) \\
\eta_4 &= t1(\mathit{Bob}), t2(\mathit{Alice}), t3(\mathit{Charlie}), t4(\mathit{Alice}), t5(\mathit{Erin}) \\
\eta_5 &= t1(\mathit{Charlie}), t2(\mathit{Alice}), t3(\mathit{Charlie}), t4(\mathit{Dave}), t5(\mathit{Bob})\,.
\end{aligned}$$

Notice that η_1 is satisfiable when $U' = \{\mathit{Alice}\}$, η_2 is satisfiable when $U' = \{\mathit{Bob}\}$, η_3 is satisfiable when $U' = \{\mathit{Charlie}\}$, η_4 is satisfiable when $U' = \{\mathit{Dave}\}$, and η_5 is satisfiable when $U' = \{\mathit{Erin}\}$. All scenarios are also satisfiable when $U' = \{\mathit{Frank}\}$, because Frank is not used in any of them. Even with the new authorization relation TA' the workflow is not 2-resilient, because if, e.g., Alice and Charlie are removed from the set of users, again there is no user capable of executing $t3$. □

To solve the SkR-SFP, an obvious strategy is to initialize the set H of k-resilient execution scenarios to the empty set, generate all subsets of users with size $t = n - k$ (for n the total number of users), update the authorization policy TA by removing the k absent users, and then compute a solution of the resulting B-SFP. If such a scenario exists, we add it to the set H.

Indeed, enumerating all subsets of size t of a set of n elements is equal to the binomial coefficient $\binom{n}{t}$. Thus, we would need $O(n!)$ calls to a procedure solving the B-SFP in the worst case (or close to n^t when t is small compared with n [96]). It is possible to reduce the number of sets that must be considered by observing (as done in [96]) that some sets of users can be ignored as they are "dominated" by others with respect to the set of tasks that they can execute. We will make this precise in Section 5.3.2 below.

Another interesting problem is to find the maximal value k_{max} such that there is a set H of scenarios that are authorized according to a relation TA and resilient up to k_{max}. A possible way to attack this problem is to use a procedure solving SkR-SFP to find a set H of resilient scenarios for a value $k^* = 1$. If there is a solution, we increase the value of k^* by one and invoke again the procedure to solve the Sk^*R-SFP. We keep increasing the value of k until no more solutions are found and set the value of k_{max} to $k^* - 1$. Since there are at most $|U|$ users, the search for k_{max} eventually terminates. In case the set U of users is large, the search may go through several iterations, each requiring the invocation of the procedure solving the SkR-SFP. A better solution is to first find an upper bound to k_{max} by preliminarily finding a solution m to the MUB-SFP. Then, an upper bound on k_{max} is given by $k^*_{max} = |U| - m$ as it is not possible to complete the workflow with less than m users. Afterwards, we invoke the procedure to solve the SkR-SFP with $k = k^*_{max}$. If the problem is solvable, then we return such a value as k^*_{max}; otherwise, we decrease k by one and invoke again the procedure solving the SkR-SFP. In the worst case (i.e. when the workflow is 0-resilient), the procedure to solve the SkR-SFP will be invoked m times; this indeed implies termination.

5.2.3 Constrained scenarios

The three problems (B-SFP, MUB-SFP, and SkR-SFP) introduced above accept as solutions unconstrained scenarios, i.e., scenarios where any authorized user can execute any task and any allowed control path can be taken (e.g., any interleaving of parallel tasks is possible and any branch of a conditional is equally likely to be executed).

Sometimes, policy designers may be interested in finding scenarios satisfying certain properties, e.g., scenarios in which only certain authorized users can execute certain tasks or some control-flows can be executed. For instance, the designer may want to investigate scenarios in which the test of a conditional evaluates to true or in which only certain users perform some tasks. We use constraints to specify the properties that scenarios must additionally satisfy in the constrained versions of the three problems above. The constraints specified by the designer are represented as predicates and are collected in a set Γ.

We are now in the position to generalize the B-SFP problem by defining its constrained version as follows.

Definition 13 (Constrained Scenario Finding Problem (C-SFP))**.** *Given the set E of eligible scenarios according to a set C of authorization constraints in a workflow $W(T, U)$ and a set Γ of constraints, return (if possible) a scenario $\eta \in E$ which is authorized according to a given relation TA and such that all the predicates in Γ evaluate to true.*

Example 14. Let us consider again the TRW together with the set U of users, the set E of eligible scenarios, and the authorization relation TA of Example 11. A solution to the C-SFP with $\Gamma = \{t2(Bob)\}$ (requiring that Bob executes task $t2$ in any scenario) is the scenario $\eta = t_1(Alice), t_2(Bob), t_3(Charlie), t_4(Dave), t_5(Erin)$. □

Similar generalizations of the MUB-SFP and the SkR-SFP are possible, thereby obtaining their constrained versions that we call C-MUB-SFP and C-SkR-SFP, respectively. More precisely, a solution to the C-MUB-SFP is a scenario that uses a minimal number of users and makes all the predicates in the set of constraints evaluate to true. A solution to the C-SkR-SFP is a k-resilient set H of scenarios such that each scenario in H makes all the predicates in the set of constraints evaluate to true.

As we will see in Section 5.3.1, the procedures used to solve the unconstrained versions of the scenario finding problems can be lifted to solve also their constrained version.

5.3 From solving the WSP to solving SFPs

The reachability graph RG defined in Chapter 3 provides us with a compact data structure to represent the set of all eligible scenarios in a workflow, which is crucial for the design of an efficient solution to SFPs.

A sequence $\eta_s = t_1(\upsilon_{j_1}), \ldots, t_n(\upsilon_{j_n})$ of task-user pairs is a *symbolic execution scenario* where υ_{j_i} is a user variable with $1 \leq j_i \leq n$ and $i = 1, \ldots, n$. A *well-formed* path in RG is a path starting with a node without an incoming edge and ending with a node without an outgoing edge. The crucial property of RG is that the symbolic execution scenario $\eta_s = t_1(\upsilon_{j_1}), \ldots, t_n(\upsilon_{j_n})$ collected while traversing one of its well-formed paths corresponds to an eligible (according to C) execution scenario $\eta_c = \mu(\eta_s) = t_1(\mu(\upsilon_{j_1})), \ldots, t_n(\mu(\upsilon_{j_n}))$ for μ an injective function from the set $\Upsilon = \{\upsilon_{j_1}, \ldots, \upsilon_{j_n}\}$ of user variables (also called *symbolic users*) to the given set U of users of $W(T, U)$.

Three observations are in order. First, μ is extended to symbolic execution scenarios in the obvious way, i.e. by applying it to each user variable occurring in them. Second, since j_i can be equal to $j_{i'}$ for $1 \leq i \neq i' \leq n$, the cardinality of Υ is at most equal to the number n of tasks in the symbolic execution scenario. Third, since μ is injective, distinct user variables are never mapped to the same user.

Example 15. Recall the excerpt of the symbolic reachability graph for the TRW in Figure 3.3 where a task-user pair $t(\upsilon_k)$ labeling an edge is shown as $t(uk)$. For instance, the symbolic execution scenario $\eta_s = t1(\upsilon_3), t3(\upsilon_3), t4(\upsilon_2)$, $t2(\upsilon_2), t5(\upsilon_1)$ (cf. the well-formed path identified by the blue nodes in Figure 3.3) represents all those execution scenarios in which a symbolic user identified by υ_3 first performs task $t1$ followed by t_3, then a symbolic user identified by υ_2 performs $t4$ and $t2$ in this order, and finally a symbolic user identified by υ_1 executes $t5$. If we apply an injective function μ from the set $\Upsilon = \{\upsilon_1, \upsilon_2, \upsilon_3\}$ of user variables to any finite set U of users (of cardinality at least three), the corresponding execution scenario $\eta_c = \mu(\eta_s)$ is eligible according to the set C of SoD constraints shown in Figure 3.1. $\square$

5.3.1 Solving the B-SFP and the MUB-SFP

Preliminarily, we need to decide how the set E of eligible scenarios and the authorization policy *TA* are specified as input to the algorithm solving the scenario finding problems.

For *TA*, we assume the availability of a Datalog program P defining the binary predicates a_t's for each task t in the workflow. For E, we define the *set $E(RG, U)$ of eligible scenarios induced by a symbolic reachability graph RG and a set U of users* as the collection of all the scenarios of the form $\mu(t_1(\upsilon_{j_1}), \ldots, t_n(\upsilon_{j_n}))$ where $v_0 \xrightarrow{t_1(\upsilon_{j_1})} \cdots \xrightarrow{t_n(\upsilon_{j_n})} v_{n+1}$ is a well-formed path in RG and μ is an injective function from $\Upsilon = \{\upsilon_{j_1}, \ldots, \upsilon_{j_n}\}$ to U.

Two observations are important. First, there are several different sets $E(RG^*, U)$ induced by a fixed symbolic reachability graph RG^* and a varying set U of users. Second, a symbolic reachability graph—once a set of users is fixed—provides an implicit and compact representation of the set of eligible scenarios. This is due to two reasons: one is the sharing of common sub-sequences of task-user pairs in execution scenarios and the other is the symbolic representation of several execution scenarios by means of a single symbolic execution scenario. This is best illustrated by an example.

Example 16. Let us consider the TRW with a set U of 6 users. The graph in Figure 5.1 is, for the sake of readability, another excerpt of the full symbolic reachability graph showing only a small subset of all well-formed paths (similar to Figure 3.3 but showing a few more nodes and edges). The full graph has 46 nodes, 81 edges, and 61 well-formed paths

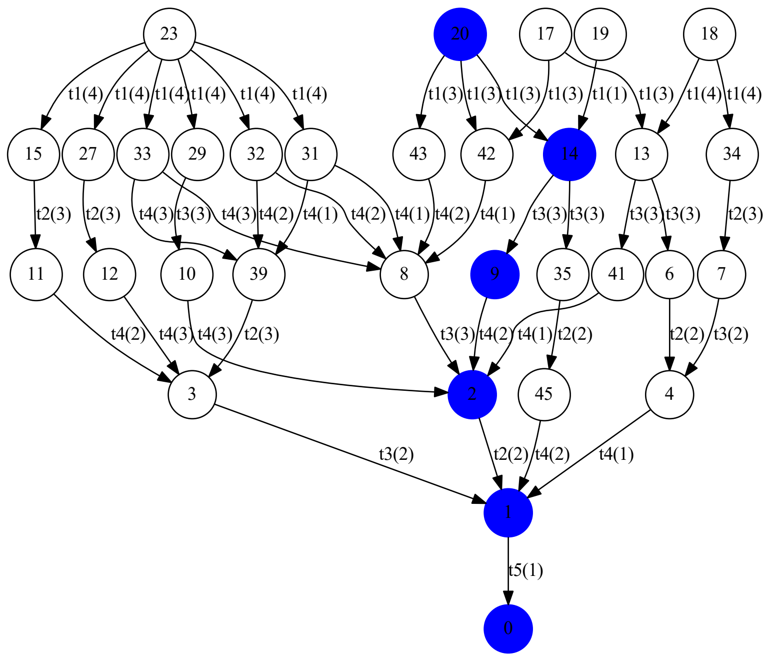


Figure 5.1: Another graph representation of the set of reachable states for the TRW

of which 21, 34, and 6 contain 3, 4, and 5, respectively, symbolic users.

For instance, notice how the sub-sequence $t3(\upsilon_2), t5(\upsilon_1)$ is shared by 6 distinct (symbolic) execution scenarios induced by the well-formed paths whose initial node is 23 (left of figure). Additionally, observe that, from the definition of $E(RG, U)$ above, in order to establish the number of all eligible paths when $|U| = n$, we just need to calculate how many injective functions there are from a set of cardinality k to a set of cardinality n—which is known to be $J(n, k) = n(n-1)(n-2)\cdots(n-k+1)$—for $n = 6$, $k = 3, 4, 5$, and take their sum. Thus, the number of all eligible paths in our case is $21 \cdot J(6, 3) + 34 \cdot J(6, 4) + 6 \cdot J(6, 5) = 19{,}080$. Compare this, with the number of well-formed paths in the symbolic reachability graph which is only 61: the blow-up factor is more than 300. Indeed, the increase is even more dramatic for larger sets of users. □

We are now ready to describe our technique, depicted in Algorithm 2, to solve the B-SFP. For the time being, let us ignore the additional input set Γ (by setting it to $\emptyset$); it will be explained in Section 5.3.3 below. The main idea underlying Algorithm 2 is to adapt a standard Depth-First Search (DFS) search to explore all well-formed paths in the reachability graph RG while checking that the scenario associated to the path is indeed authorized by using the run-time monitor, synthesized in the on-line phase of the technique in Chapter 3.

Lines 1–2 are the standard initialization phase of a DFS algorithm in

Algorithm 2 Solving the B-SFP

```
Require: $RG$ symbolic reachability graph, $U$ set of users, $P$ Datalog program defining
    $a_t$'s, $\Gamma$ set of facts
Ensure: $\eta$ authorized execution scenario
 1: for all $v \in \mathsf{Nodes}(RG)$ do $visited[v] \leftarrow \texttt{false}$;
 2: end for
 3: $\eta \leftarrow \epsilon$; $NI \leftarrow \mathsf{NoIncoming}(RG)$;
 4: while ($v \in NI$ and $\eta = \epsilon$) do
 5:     $\eta \leftarrow \mathrm{DFS}(v, \epsilon, \Gamma)$; $NI \leftarrow NI \setminus \{v\}$;
 6: end while
 7: return $\eta$
 8: function DFS($v, \eta, H$)
 9:     $visited[v] \leftarrow \texttt{true}$; $OE \leftarrow \mathsf{OutGoing}(v)$;
10:     if $OE = \emptyset$ then return $\eta$
11:     else
12:         for all $v \xrightarrow{t(v)} w \in OE$ do
13:             if (not $visited[w]$ and $M, P, H \vdash^{v \mapsto u} can\_do(t, v)$) then
14:                 return DFS($w, \mathsf{append}(\eta, t(u)), H \cup \{h(t, u)\}$)
15:             end if
16:         end for
17:     end if
18:     return $\epsilon$
19: end function
```

which all nodes in RG (returned by the function Nodes) are marked as not yet visited. Lines 3–6 invoke the (modified) DFS algorithm on each node without an incoming edge in RG (returned by the function NoIncoming) until either all such nodes have been considered (this allows us to consider all well-formed paths) or an authorized scenario (if any) has been found (line 7). Lines 8–19 show the (modified) DFS recursive function which takes as input a node v and extends a sequence η of task-user pairs to an authorized execution scenario (if possible). Line 9 marks as visited the node v under consideration and computes its set OE of outgoing edges (returned by the function OutGoing). Line 10 checks whether the set of outgoing edges of v is empty: if this is the case, then we have considered all task-user pairs in a well-formed path and the sequence η containing them is an authorized execution scenario. If this is not the case, we have not yet considered all task-user pairs in a well-formed path of RG and thus we need to consider the possible continuations in OE. This is done

Algorithm 3 Solving the MUB-SFP

```
Require: $RG$ symbolic reachability graph, $U$ set of users, $P$ Datalog program defining
    $a_t$'s, $\Gamma$ set of facts
Ensure: $\eta$ authorized execution scenario
 1: for all $v \in$ Nodes($RG$) do $visited[v] \leftarrow$ false;
 2: end for
 3: $\eta \leftarrow \epsilon$; $\eta_M \leftarrow \epsilon$; $NI \leftarrow$ NoIncoming($RG$);
 4: while $v \in NI$ do
 5:     $\eta \leftarrow \mathrm{DFS}(v, \epsilon, \Gamma)$; $NI \leftarrow NI \setminus \{v\}$;
 6:     if ($\eta \neq \epsilon$ and $\eta_M = \epsilon$) then
 7:         $\eta_M \leftarrow \eta$;
 8:     else if ($\eta \neq \epsilon$ and $|usr(\eta)| < |usr(\eta_M)|$) then
 9:         $\eta_M \leftarrow \eta$;
10:     end if
11: end while
12: return $\eta_M$
```

in the loop at lines 12-16: an edge $v \xrightarrow{t(v)} w$ in OE is selected (line 12), it is checked if the node w is not yet visited and if the run-time monitor combined with the authorization policy P can find a user u capable of executing the task t in label of the edge in OE under consideration (line 13). The second check (namely, $M, P, H \vdash^{v \mapsto u} can\_do(t, v)$) is done by asking a Datalog engine to find a user u in U to which the user variable v can be mapped (cf. superscript $v \mapsto u$ of $\vdash$) without violating the execution and the authorization constraints together with the authorization policy specified by P. If the test at line 13 is successful, line 14 is executed whereby a recursive call to the DFS function is performed in which the new node to consider is w, the sequence η of task-user pairs is extended with $t(u)$ (by invoking the function append), and the set H of facts keeping track of the tasks executed so far is also extended by $h(t, u)$. In case all edges in OE have been considered but none of them makes the check at line 13 successful, the empty sequence is returned (line 18). Notice that at line 13, instead of enumerating all suitable users in U to which v can be mapped, we exploit the capability of the Datalog engine to find the right user. This permits us to exploit well-engineered implementations of Datalog engines instead of designing and implementing new heuristics to reduce the time taken to enumerate the users in U. This concludes the description of the algorithm solving the B-SFP.

It is possible to modify Algorithm 2 following the idea discussed after Example 12 in order to *solve the MUB-SFP*. This requires to avoid returning the authorized scenario as soon as we find one (removing the condition $\eta = \epsilon$ in line 4) so that all well-formed paths in RG are considered. Moreover, a global variable η_M is maintained in which a candidate scenario with a minimal user base is stored and updated according to the strategy discussed above comparing the users occurring in η_M and those in the currently considered scenario. As a result of these modifications, we obtain

Algorithm 3 that is capable of solving the MUB-SFP. The DFS function used in this algorithm is the same as the DFS function in Algorithm 2).

The complexity of Algorithms 2 and 3 can be derived from that of the standard DFS algorithm, which is $O(n + m)$ for n the number of nodes and m the number of edges, when using an adjacency list to represent the graph. Notice that the most computationally intensive operation is the invocation of the Datalog engine at line 13, which takes polynomial time as the only part that changes over time is the set H of facts whereas the Datalog programs M and P are fixed; cf. the results on data complexity of Datalog programs in [26]. It is easy to see that we invoke (at most) $O(n+m)$ times the Datalog engine for both algorithms (line 13 in Algorithm 2 and hidden in the DFS function in Algorithm 3). This is much better than the upper bounds discussed in Section 5.2. To see this, consider the situation in Example 16: $\ell_{max} = 5$ and $|E| = 19,080$ so that the check for authorization (modulo constant factors) is invoked at most $95,400$ times whereas in Algorithm 2 (or its modified version for the MUB-SFP) the same check is invoked at most $n + m = 46 + 81 = 127$ times.

5.3.2 Solving the SkR-SFP

As discussed in Section 5.2.2, a naive solution to the SkR-SFP requires the enumeration of all subsets of the set of users of a given size. Indeed, this is expensive from a computational point of view. To alleviate this, it is possible to adapt an heuristic proposed in [96] and reduce the number of sets of a given size to be considered.

The idea is based on the notion of “dominance” that is defined in terms of which tasks a given set of users can execute. If a set A_1 of users can execute at least the same set of tasks that a second set A_2 of users can execute, then we say that A_1 dominates A_2 and the latter can be ignored without loss of generality. This implies that if an instance of the SkR-

SFP can tolerate the removal of users in a set of absent users A_1, then the problem instance can tolerate the removal of users in the second set A_2. This is why we can consider only the set A_1 and safely disregard A_2. Below, the set of users in the authorization policy TA is $\{u \mid (u,t) \in TA\}$ and the set $Perms(u)$ of permissions associated to a user u is $\{t \mid (u,t) \in TA\}$.

Definition 14 (Absent set domination)**.** *A user u_1 in TA dominates a user u_2 in TA iff $Perms(u_1) \supseteq Perms(u_2)$. A set of (absent) users A_1 dominates a set A_2 of (absent) users iff there exists a bijection f from A_1 to A_2 such that, for each user u in A_1, u dominates $f(u)$ in A_2.*

We are now in the position to adapt the result (Lemma 7) in [96] to avoid the enumeration of all subsets of a given size while solving the SkR-SFP.

Lemma 1. *Let E be a set of eligible scenarios according to a set C of authorization constraints containing only SoD (i.e. ρ is $\neq$ for each $(t,t',\rho) \in C$) in a workflow $W(T,U)$ and an integer k. If $A_1, A_2 \subseteq U$, A_1 dominates A_2, and the SkR-SFP for $W(T,U \setminus A_1)$ is solvable, then the SkR-SFP for $W(T,U \setminus A_2)$ is also solvable.*

Proof. (Sketch) Assume that A_1 dominates A_2 and the SkR-SFP for $W(T,U \setminus A_1)$ is solvable. The former is equivalent to

$$Perms(u) \supseteq Perms(f(u)) \quad \text{for each } u \text{ in } U \setminus A_1 \tag{5.1}$$

while the latter implies (by recalling the definition of solution to the WSP) that

$$\forall t \in T, \exists u \in U \setminus A_1.\ (u,t) \in TA \tag{5.2}$$

$$\forall (t_1,t_2,\rho) \in C, \exists u_1,u_2 \in U \setminus A_1.\ (u_1,u_2) \in \rho, \tag{5.3}$$

where (5.2) states that each task in a scenario solving the WSP is performed by an authorized user and (5.3) states that each authorization constraint is satisfied.

By (5.1), it is possible to show that if (5.2) holds (i.e. the problem is solvable after the removal of A_1), it also holds when replacing A_2 with A_1 (i.e. the problem is solvable after the removal of A_2), because A_2 contains the same number of users of A_1 (since f is a bijection) and each user in A_2 has the same permissions or less of the corresponding user (via the bijection f) in A_1 (a more detailed proof of this property proceeds along the same line of that in Lemma 7 in [96]).

Unfortunately, it is not possible to show for an arbitrary set C of constraints that (5.3) implies (5.3) with A_2 in place of A_1. This is so because we can imagine a scenario where the users removed in A_2, which are different from those in A_1, are necessary to fulfill some of the constraints in C. A somewhat contrived example is a constraint requiring a task $t2$ to be executed by a user more senior, given a seniority relation $<$ inside an organization [36], than that who executes $t1$, and a less senior user $u1$ with more permissions than a user $u2$, so that $u1 < u2$, but $Perms(u_1) \supseteq Perms(u_2)$. Although $u1$ dominates $u2$, there may be no user able to execute $t2$ due to the constraint.

Fortunately, when C contains only SoD constraints, one can observe that the bijection f from A_1 to A_2 maps distinct users into distinct users. As a consequence, from an execution scenario solving the SkR-SFP for $W(T, U \setminus A_1)$, it is always possible to find an execution scenario solving the SkR-SFP for $W(T, U \setminus A_2)$. □

Lemma 1 allows us to avoid enumerating all possible subsets of a given size when solving the SkR-SFP by exploiting the dominance relation among sets of absent users in presence of SoD constraints only. How this is done is shown in Algorithm 4.

In line 1, the variable H which stores the scenarios to be returned is initialized to the empty set. In line 2, the function NextSubset returns a new, i.e. not previously generated, subset of size $t = |U| - k$ of the set U

of users and s stores the returned subset. The NextSubset function is implemented as an iterator that returns only the next subset at each call, so that it is not necessary to generate all subsets up front, which is expensive from a computational point of view. Most importantly, NextSubset implements the smart enumeration of subsets that are not dominated by others according to Lemma 1 when the set C contains only SoD constraints. In line 3, the Datalog program defining the a_t's is updated by removing all assertions that include users not in the current subset S ($P|_S$), and in line 4 the SFP function (implemented as shown in Algorithm 2) is called to find an execution scenario. If no scenario is found (line 5), an empty set is returned (line 6) and execution halts. Notice that the execution must stop because the SkR-SFP is defined as finding a scenario for *every* subset of size t in U. If a scenario is found (line 7), it is added to the set H (line 8); after this is done for all subsets, NextSubset returns an empty set and the final H is returned (line 11).

Algorithm 4 Solving the SkR-SFP

```
Require: $RG$ symbolic reachability graph, $U$ set of users, $P$ Datalog program defining
    $a_t$'s,
        $\Gamma$ set of facts, $C$ set of authorization constraints, $k \in \mathbb{N}$ desired resiliency
Ensure: $H$ set of authorized execution scenarios
 1: $H \leftarrow \emptyset$;
 2: while ($S \leftarrow \mathsf{NextSubset(k, U)}$ and $S \neq \emptyset$) do
 3:     $P' \leftarrow P|_S$;
 4:     $\eta \leftarrow \mathrm{SFP}(RG, U, P', \Gamma)$;
 5:     if $\eta = \epsilon$ then
 6:         return $\emptyset$;
 7:     else
 8:         $H \leftarrow H \cup \{\eta\}$;
 9:     end if
10: end while
11: return $H$;
```

5.3.3 Solving the C-SFP

So far, we have assumed that the set Γ of constraints taken as input of Algorithm 2 is empty. Here, we consider the situation in which this is no more the case and show how Algorithms 2, 3, and 4 also solve the constrained versions of the B-SFP, MUB-SFP, and the SkR-SFP.

Let us consider Algorithm 2 and how it solves a C-SFP by taking as input a non-empty set Γ of facts, that can be used to drive the search for a scenario with particular characteristics. For instance, one can be interested in authorized scenarios in which a certain user only, say u^*, executes a given task, say t^*. It is possible to steer the search towards such scenarios by setting Γ to be the singleton containing the fact $h(t^*, u^*)$. Another use of Γ is guiding the search towards scenarios in which the tests of certain conditionals of the workflows are either true or false. Again, it is possible to add the facts encoding that particular control conditions are true or false to Γ in order to force Algorithm 2 to find scenarios taking only particular execution branches.

A more interesting approach is to develop an interactive algorithm where after each step in the search for a solution, the user of the algorithm is asked to enter his or her preferences to which user executes which task and which is the next task to be executed. If he or she does not specify anything, the algorithm picks an arbitrary user and task; otherwise the algorithm checks if the corresponding instance of the WSP is solvable and, if the case, would allow the user to proceed with the exploration.

Since Algorithms 3 and 4 (i.e. the procedures to find the solutions to the MUB-SFP and to the SkR-SFP) use Algorithm 2 as a sub-procedure, they straightforwardly inherit the capability to solve the constrained versions of B-SFP and MUB-SFP.

Part II

Applications

Chapter 6

Cerberus: integrating monitor synthesis in workflow management systems[1]

In this Chapter, we present CERBERUS[2], a tool that relies on the technique described in Chapters 3 and 4 to automatically synthesize, at design-time, enforcement monitors that can be used in workflow management systems to guarantee, at run-time, that workflow instances can terminate while satisfying the authorization policy and the authorization constraints. As described before, the synthesized monitors are parametric in the authorization policy so that they can be combined at run-time with authorization policies dedicated to different instances of a process. The tool also implements the solutions to the SFPs described in Chapter 5 to guide users during the deployment of workflows.

CERBERUS can be easily integrated in many workflow management systems, it is transparent to process designers, and does not require any knowledge beyond usual BP modeling. To demonstrate the tool, we integrated

[1]Parts of this chapter were previously published in [31]

[2]Cerberus is a three-headed watchdog in Greek mythology, with the first head associated to the past, the second to the present and the third to the future. CERBERUS acts as a monitor that takes into account the history of executions, the current authorization relation and future executions to grant or deny requests.

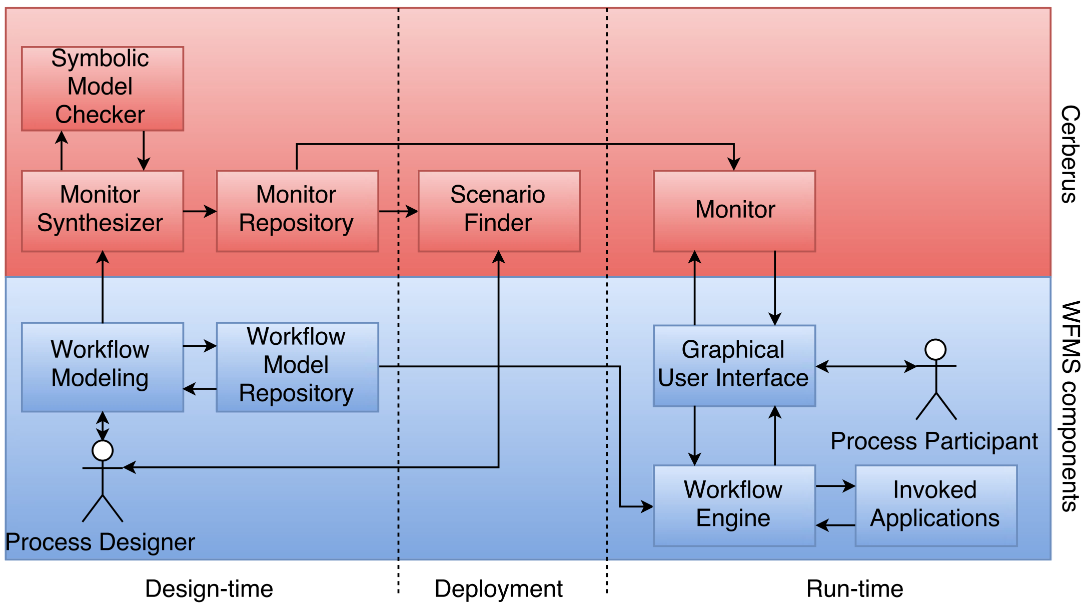


Figure 6.1: Tool Architecture

it into the SAP HANA Operational Intelligence (OpInt) platform, which offers a BPMN modeling and enactment environment integrated with a repository capable of storing workflow models.

6.1 Tool architecture and implementation

A reference architecture for WFMS [158] is composed of the five blue elements shown in Figure 6.1. *Workflow Modeling* is a graphical user interface for a *Process Designer* to create workflow models using a modeling language such as BPMN or YAWL. The models are stored in a *Workflow Model Repository*, while the *Workflow Engine* interprets the models and directs the execution to the *Invoked Applications*, in the case of system and script tasks, or to a *Graphical User Interface (GUI)*, in the case of user tasks, which are performed by *Process Participants*.

On top of the WFMS components, we add the CERBERUS components shown in red in Figure 6.1. The *Monitor Synthesizer* is responsible for interpreting the workflow model and automatically translating it into a transition system format, with initial and final formulae, accepted by a *Symbolic Model Checker* capable of computing a reachability graph whose paths are all possible executions of the workflow (cf. Algorithm 1). The reachability graph is translated into a language such as Datalog or SQL and stored in the *Monitor Repository.* The *Monitor* itself is invoked by the GUI and grants or denies user requests to execute tasks. This is because users only access tasks through the GUI and automatic tasks are not part of the authorization policy or constraints.[3]

The *Scenario Finder* can be used by a process designer (or a policy designer) during deployment, when an authorization policy is specified, to make sure that the desired execution scenarios are possible (cf. Algorithm 2, 3, and 4). After a *Monitor* is synthesized, the designer invokes the *Scenario Finder*, inputs an authorization policy, chooses which kind of scenarios to find (e.g., basic or minimal) and optionally gives a set of constraints (as described in Section 5.2.3). The *Scenario Finder* outputs found scenarios and the designer can repeat the process with different parameters.

The modular approach described in Chapter 4 has been implemented in the tool and whenever a process designer uses the modeling component, he/she can import models with their associated monitors from the repositories, combine the models with new or pre-existing models and export the resulting complex component back to the workflow repository, alongside its monitor stored in the monitor repository. The modular approach and decomposition of models is also key to scalability for monitor synthesis. With the use of the modular approach, scalability is shown in Section 6.3.

The main goals in the design of CERBERUS are usability, scalability and

[3]This is a limitation of the current implementation. Nonetheless the approach is able to monitor any task subject to an authorization policy.

minimal interference with pre-existing functionalities. Usability is achieved because the tool is fully automated and all the formal details are hidden from the process designer, who only has to input the workflow model with a set of authorization constraints that he/she wishes to be enforced (which can be done graphically). Scalability is ensured by the use of modular monitor synthesis (decomposing workflows into components, synthesizing monitors for them and combining the results) and minimal interference is guaranteed by using the tool as a plug-in, so that both monitor synthesis and enforcement can be easily activated or deactivated.

The Cerberus implementation is built on top of OpInt to synthesize, store, combine and retrieve run-time monitors for security-sensitive workflows therein modeled and enacted. HANA Studio is the IDE that acts as the *Workflow Modeling* component, while the HANA Repository implements both the *Workflow Model Repository* and the *Monitor Repository*. We added the authorization constraint specification and monitor synthesis capabilities in the IDE and used MCMT as the *Symbolic Model Checker*. The *Monitor Synthesizer* is written in Python (core algorithms) and JavaScript (IDE and repository integration). The monitors are output in SQL as a view that is queried by the execution engine. The *Workflow Engine* differs from traditional WFM systems because OpInt does not directly execute the BPMN models, but instead translates them to executable artifacts (JavaScript and SQL code) that manage and perform the tasks in the workflows. The invoked applications are handled by SQL procedure calls and the GUI for user tasks is integrated in a web task management dashboard.

Since we build on top of a reference architecture, other possible implementations of Cerberus could use different versions of the WFMS components, e.g., the IBM Business Process Manager platform[4] or open-

[4]`https://www-03.ibm.com/software/products/en/business-process-manager-family`

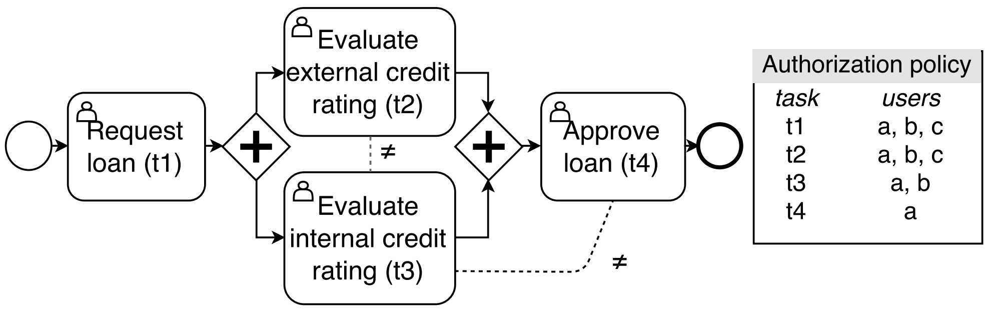


Figure 6.2: Loan Origination Process

source components, e.g., Activiti[5]. The *Monitor Synthesizer* is capable of generating monitors for SWI Prolog[6], pyDatalog[7], MySQL[8], and SAP HANA SQLScript[9], but there is no integration with other WFMS systems.

6.2 Using Cerberus

To demonstrate how to use Cerberus, we introduce another workflow example.

Example 17. The simplified Loan Origination Process shown in Figure 6.2 is composed of four tasks, *Request loan* ($t1$), *Evaluate external credit rating* ($t2$), *Evaluate internal credit rating* ($t3$) and *Approve loan* ($t4$), and two SoD constraints, which impose that the user who executes $t2$ ($t3$, resp.) cannot also execute $t3$ ($t4$, resp.). The authorization policy, associating users to tasks, is described in the table shown in the Figure. □

The usage of the tool involves four steps: design-time specification, monitor synthesis, deployment, and run-time enforcement. SAP HANA is an in-memory relational database, so the BPMN artifacts and the monitors

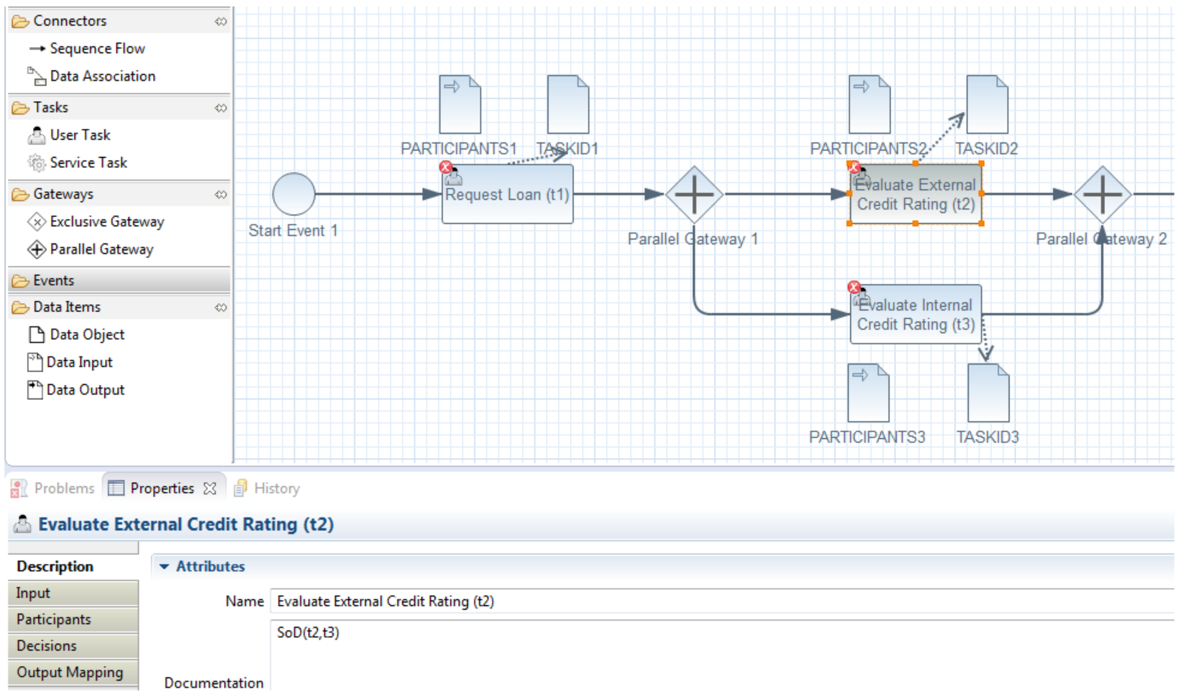
[5]http://activiti.org/
[6]http://www.swi-prolog.org/
[7]https://sites.google.com/site/pydatalog/home
[8]https://www.mysql.com/
[9]http://help.sap.com/hana/SAP_HANA_SQL_Script_Reference_en.pdf

are translated to SQL. There is a long tradition of works using relational languages, such as Datalog and SQL, to express role-based access control and other authorization policies [130]. Moreover, we use database tables to store the users (`USERS`), authorization policy (one `PARTICIPANTS` table for each task) and execution history (`HST`).

6.2.1 Design-time

At design-time, a process designer uses the HANA Studio IDE to model the control-flow and authorization constraints of a workflow. Authorization constraints are not part of standard BPMN and we simply use task documentation to input the constraints in textual form. This can be changed in the future so that constraints are specified as graphical elements. Authorization policies are specified by linking each task to an assignment table in the database. The tables will be populated only at deployment-time. When design is complete, the model is translated to SQL by pressing a button in the IDE.

To model the example of Figure 6.2, a process designer uses the IDE to create a new BPMN file and graphically drags, drops and connects the required elements: start and end events (the circles in the figure), user tasks (rounded rectangles), sequence flows (solid arrows) and parallel gateways (diamonds labeled by “+”). The authorization constraints are input in the documentation of the second and third tasks, the `PARTICIPANTS` data objects are linked to database tables with the same names (which are empty at the moment) and the task UIs are linked to web pages (see Figure 6.3).



Evaluate External Credit Rating (t2)

Name: Evaluate External Credit Rating (t2)

Documentation: SoD(t2,t3)

Figure 6.3: Workflow constraints

6.2.2 Monitor synthesis

The monitor synthesis runs in parallel with the BPMN-to-SQL compiler and is completely transparent to end users. When the monitor synthesizer receives a request to generate a monitor, the BPMN model file (in XML) is read from the repository and translated to a symbolic transition system that is fed to the monitor synthesis tool. The resulting SQL view, using the database tables representing users, authorization policy and history of execution, is stored in the repository and queried at run-time by the execution engine.

In the example of Figure 6.2, the monitor consists of an SQL view defined by a procedure containing, among others, the following query for $t2$ (simplified for the sake of clarity):

```
SELECT U2.ID FROM USERS AS U1, USERS AS U2 WHERE HST.dt1 = 1 AND HST.dt2 = 0
AND HST.dt3 = 1 AND HST.dt4 = 0 AND (U1.ID <> U2.ID) AND NOT HST.t3by = U1.ID
AND NOT HST.t3by = U2.ID AND U2.ID IN (SELECT ID FROM PARTICIPANTS2) AND U1.ID
IN (SELECT ID FROM PARTICIPANTS4)
```

which encodes the fact that, to execute $t2$, the system must be in a state where $t1$ and $t3$ have been executed, but neither $t2$ nor $t4$ (`dt1 = 1 AND dt2 = 0 AND dt3 = 1 AND dt4 = 0`), there must be a user $u1$ who can execute $t2$ (`SELECT ID FROM PARTICIPANTS2`), and a different user $u2$ (`U1.ID <> U2.ID`) who can execute $t4$ (`SELECT ID FROM PARTICIPANTS4`) and neither user should have executed $t3$ because of the SoDs between $t2$ and $t3$ and between $t3$ and $t4$ (`NOT t3by = U1.ID AND NOT t3by = U2.ID`). Other queries for $t2$ and all queries for other tasks are similar.

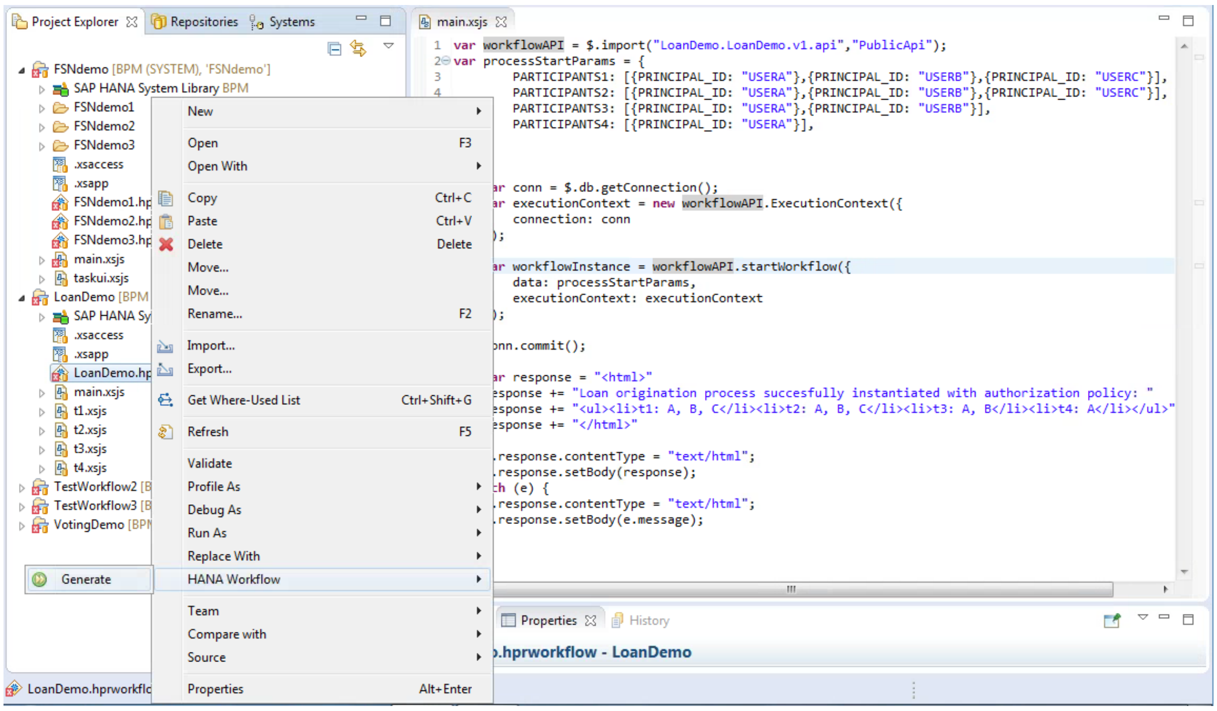
6.2.3 Deployment

For the deployment of a workflow it is necessary to specify the concrete authorization policy by populating the linked database tables `PARTICIPANTS1` to `PARTICIPANTS4`. End users manage workflows using a generated API, as shown in Figure 6.4. The snippet of code in Figure 6.4 shows the programmatic instantiation of the authorization policy given in the table shown in Figure 6.2.

As discussed above, at this moment the process designer can also invoke the scenario finder tool to find concrete execution scenarios with a given authorization policy. Some valid execution scenarios for Example 17 are $t1(C), t2(A), t3(B), t4(A)$ and $t1(A), t3(B), t2(C), t4(A)$.

6.2.4 Run-time

At run-time, there is a running job responsible for calling the next tasks based on tokens stored in the database, whose flow is specified by the control-flow in the BPMN model. When a human task is executed, the monitor associated to the workflow is called into action by the automatic invoking of a procedure from the task UI. To grant or deny a request, the monitor queries the `USERS`, `PARTICIPANTS1` to `PARTICIPANTS4` and `HST`



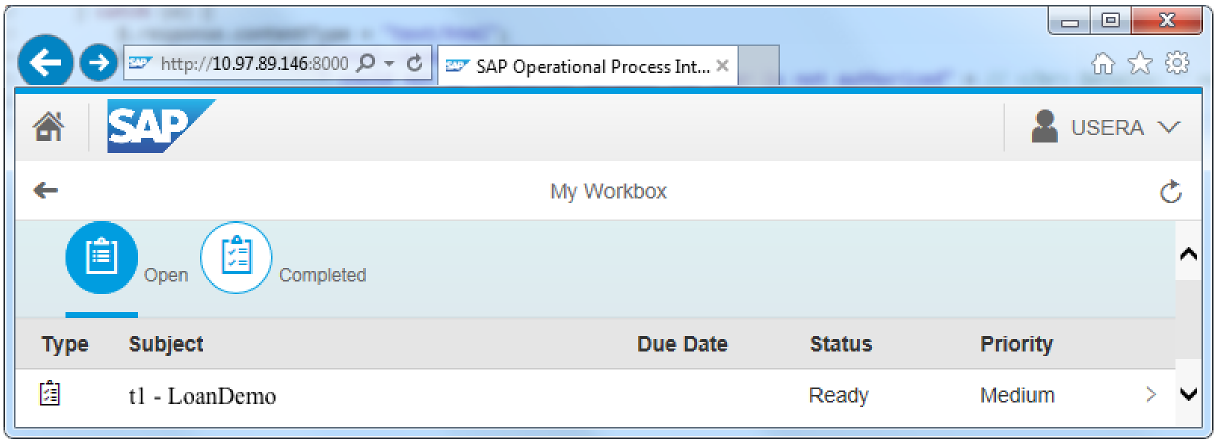
```
var workflowAPI = $.import("LoanDemo.LoanDemo.v1.api","PublicApi");
var processStartParams = {
        PARTICIPANTS1: [{PRINCIPAL_ID: "USERA"},{PRINCIPAL_ID: "USERB"},{PRINCIPAL_ID: "USERC"}],
        PARTICIPANTS2: [{PRINCIPAL_ID: "USERA"},{PRINCIPAL_ID: "USERB"},{PRINCIPAL_ID: "USERC"}],
        PARTICIPANTS3: [{PRINCIPAL_ID: "USERA"},{PRINCIPAL_ID: "USERB"}],
        PARTICIPANTS4: [{PRINCIPAL_ID: "USERA"}],


ar conn = $.db.getConnection();
ar executionContext = new workflowAPI.ExecutionContext({
    connection: conn
);

ar workflowInstance = workflowAPI.startWorkflow({
    data: processStartParams,
    executionContext: executionContext
);

onn.commit();

ar response = "<html>"
esponse += "Loan origination process succesfully instantiated with authorization policy: "
esponse += "<ul><li>t1: A, B, C</li><li>t2: A, B, C</li><li>t3: A, B</li><li>t4: A</li></ul>"
esponse += "</html>"

.response.contentType = "text/html";
.response.setBody(response);
ch (e) {
.response.contentType = "text/html";
.response.setBody(e.message);
```

Figure 6.4: Generate the monitor and instantiate the workflow

tables described above to ensure that the requesting user is entitled to perform the task, the user has or has not performed another conflicting task, and the execution of this task will not prevent the satisfaction of the workflow (as shown in the example query above). An example of the UI with the list of tasks that a user is entitled to perform is shown in Figure 6.5. When a user selects a task to perform he/she is taken to a task UI displaying the task actions and when the user selects to complete the task, the monitor is called and the request is either granted (Figure 6.6) or denied (Figure 6.7).

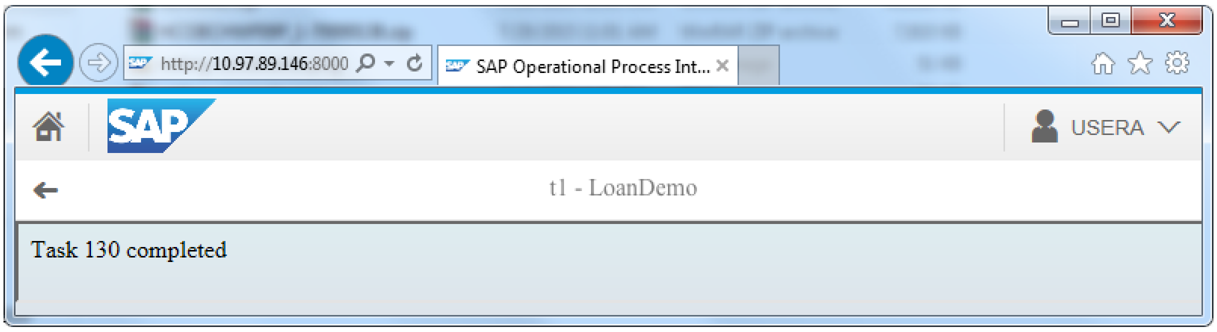
6.3 Experiments

The previous Section described the use of CERBERUS on a simple example, but to further validate the techniques in this thesis, we performed a series of experiments, focusing on monitor synthesis for real-world work-



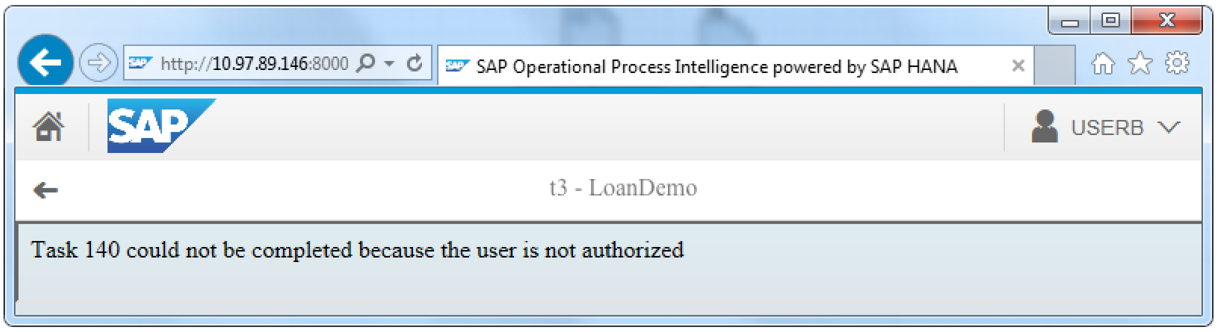
Type	Subject	Due Date	Status	Priority
	t1 - LoanDemo		Ready	Medium

Figure 6.5: Task list



Task 130 completed

Figure 6.6: Request granted



Task 140 could not be completed because the user is not authorized

Figure 6.7: Request denied

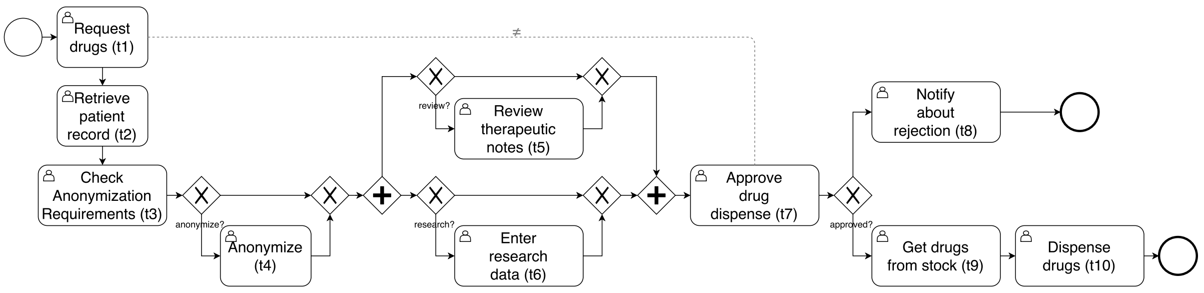

Request drugs (t1)
Retrieve patient record (t2)
Check Anonymization Requirements (t3)
anonymize?
Anonymize (t4)
review?
Review therapeutic notes (t5)
research?
Enter research data (t6)
≠
Approve drug dispense (t7)
approved?
Notify about rejection (t8)
Get drugs from stock (t9)
Dispense drugs (t10)

Figure 6.8: Drug dispensation process from [10].

flows (Section 6.3.1), scalability of monitor synthesis (Section 6.3.2) and scenario finding (Section 6.3.3).

All experiments were run on a MacBook Air 2014 laptop with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM. All monitor synthesis and query answering times in the experiments are related to monitors for the pyDatalog engine.

6.3.1 Monitor synthesis - real-world workflows

We have experimented with real-world workflows taken from related works (e.g., [12, 10]) and available repositories (e.g., Signavio reference models[10], SAP Business Process Repository[11], and the BPM Academic Initiative[12]). Below, we discuss one example using the basic control-flow patterns (sequential, parallel, and alternative execution [148]) to demonstrate the expressiveness of our approach.

Example 18. Figure 6.8 shows the Drug dispensation process from [10]. The execution of an instance of this workflow starts with a Patient requesting drugs to a Nurse ($t1$). The Nurse consults the Patient's record and sends it to a PrivacyAdvocate ($t2$), who decides if this data should be anonymized ($t3$ and $t4$). If the drug prescription has therapeutic notes,

[10]Available at `http://www.signavio.com/reference-models/`
[11]Available at `https://implementationcontent.sap.com/bpr`
[12]Available at `http://bpmai.org/`

they must be reviewed by a Therapist ($t5$) and in parallel, a Researcher can add data related to experimental drugs ($t6$). In the end a Pharmacist either approves or denies the process ($t7$) and a Nurse carries out the related tasks: collect and dispense the drugs ($t9$ and $t10$) or notify the Patient ($t8$). A SoD constraint for this workflow is $(t1, t7, \neq)$: the same user cannot act as Patient and Pharmacist, so that a Pharmacist cannot dispense drugs to himself. $\square$

A workflow of this size (10 tasks) would be intractable for our tool without modular decomposition. In fact, without using modular specifications, for a workflow with up to 5 tasks, running Algorithm 1 (the most expensive step of our technique) takes few seconds on a standard laptop; for 6 tasks, around a minute; and for 7 tasks, already two hours and a half.

Thus, we come up with a specification consisting of two components to be executed one after the other; the former contains tasks $t_1, \ldots, t_4$ and the latter, tasks $t_5, \ldots, t_{10}$. This particular decomposition is easy because it splits the workflow roughly in half and the first component ends before the parallel gateway where the second component starts. According to some control flow operators, not all tasks must be executed in the workflow for its successful termination. In fact, tasks $t4$, $t5$ and $t6$ may or may not be executed depending on certain conditions (e.g., "*anonymize?*"), while tasks $t8$ and $t9$ are mutually exclusive.

To represent the decisions that have to be taken to complete the workflow, we create transitions for the various branches whose enabling conditions depend on additional variables—called *environment variables*—modeling non-deterministic choices of the environment. For instance, the fact that task $t7$ is followed by the decision point *approved?* can be represented by the following two transitions:

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

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

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

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

The time spent to compute the reachability graph and translate it to a Datalog program is around 15s (roughly, 3s for the first and 12s for the second). The time taken by the synthesized monitor to answer access requests is almost negligible.

6.3.2 Monitor synthesis - scalability

To test the scalability of our approach, we have extended the generator of random workflows used in [42] to produce modular workflows. Our generator takes the following parameters: n_w, the number of components and n_{tw}, the number of tasks in each component ($n_t = n_w \cdot n_{tw}$ is the total number of tasks; n_u, the number of users; p_a, called authorization density, the ratio between the cardinality of $\bigcup_t \{a_t(u) = true | u \text{ is a user}\}$ and $n_t \cdot n_u$ (where t ranges over the set of tasks); and p_c, called constraint density, the ratio between the number of SoD constraints in C and n_t.

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

According to our experience with real-world workflows, we set n_{tw} to 5 and increase the number n_w of components so that the total number n_t of tasks in the generated workflows range from 10 to 500 (although workflows of this size are rarely seen in practice, we want to emphasize the scalability of the technique. Notice that [42] considers workflows with at most 150 tasks). More precisely, we let $n_t = 10, 20, \ldots, 150, 200, 250, \ldots, 500$ and, following [42], $n_u = n_t$, $p_a = 100\%, 50\%, 10\%$, $p_c = 5\%, 10\%, 20\%$.

Figure 6.9 shows the behavior of our prototype tool for the off-line phase and Figure 6.10 shows the performance for the on-line phase. Both refer to the modular workflows produced by the random generator with the parameters described above. The x-axis shows the number n_t of tasks in the workflow and the y-axis, the timings in seconds. Each line corresponds to different values for the pair (p_a, p_c) of parameters (recall that $n_u = n_t$).

Discussion

It is clear that the computation time of our tool in both the on-line and off-line phases is linear in the number of tasks in the workflows for any value of the pair (p_a, p_c) of parameters. For the off-line phase, this is so because of the divide-and-conquer strategy supported by modular workflows. For the on-line phase, the linear growth is because the synthesized Datalog programs belong to a class whose requests can be answered in linear-time.

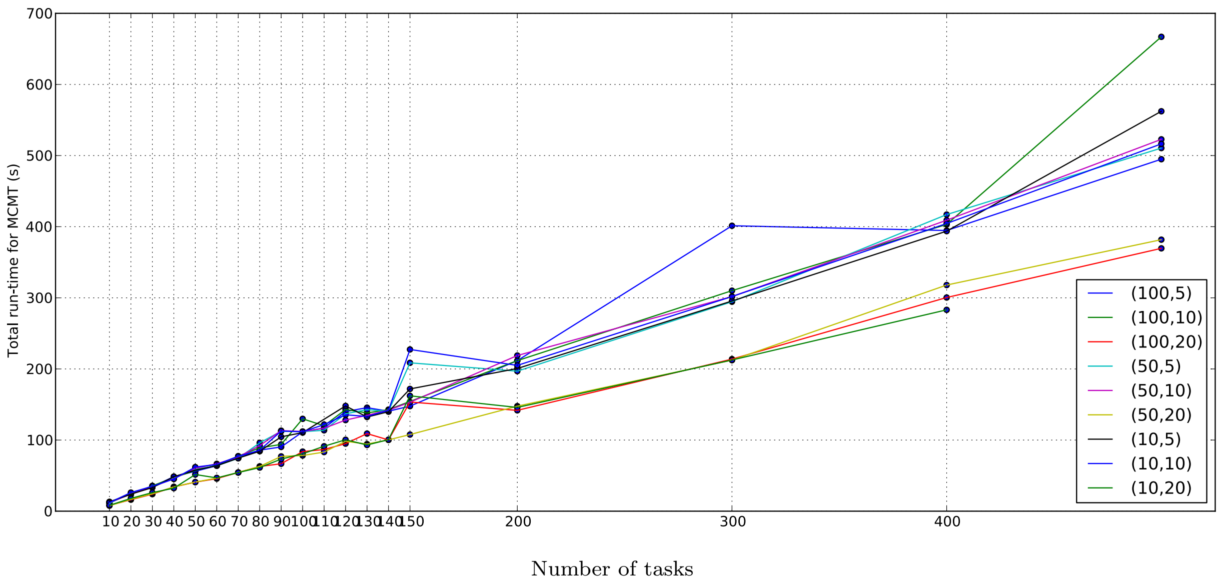


Figure 6.9: Total run-time of off-line phase by the number of tasks in all configurations

Notice also that for workflows with $n_t \leq 200$, the (median) time to answer a request is under 1 second while for workflows with $200 < n_t \leq 500$, it is around 1.6 seconds. This clearly demonstrates that the monitors synthesized by our tool are suitable to be used at run-time.

6.3.3 Scenario finding

In the scenario finding experiments, we use the TRW described in Chapter 3 as well as two real-world examples, shown in Figures 6.11 and 6.12, derived from business processes available in an on-line repository provided by Signavio,[13] which contains models inspired by the ISO9000 standard for quality management and the ITIL 2011 set of best practices for IT service management.

Example 19. The goal of the ITIL workflow is to report costs and revenues of an IT Service. It is composed of 7 tasks and 2 SoD constraints. Tasks $t1$, $t2$, $t3$, $t6$ and $t7$ are for the checking and correction of bookings,

[13]Available at `http://www.signavio.com/reference-models/`

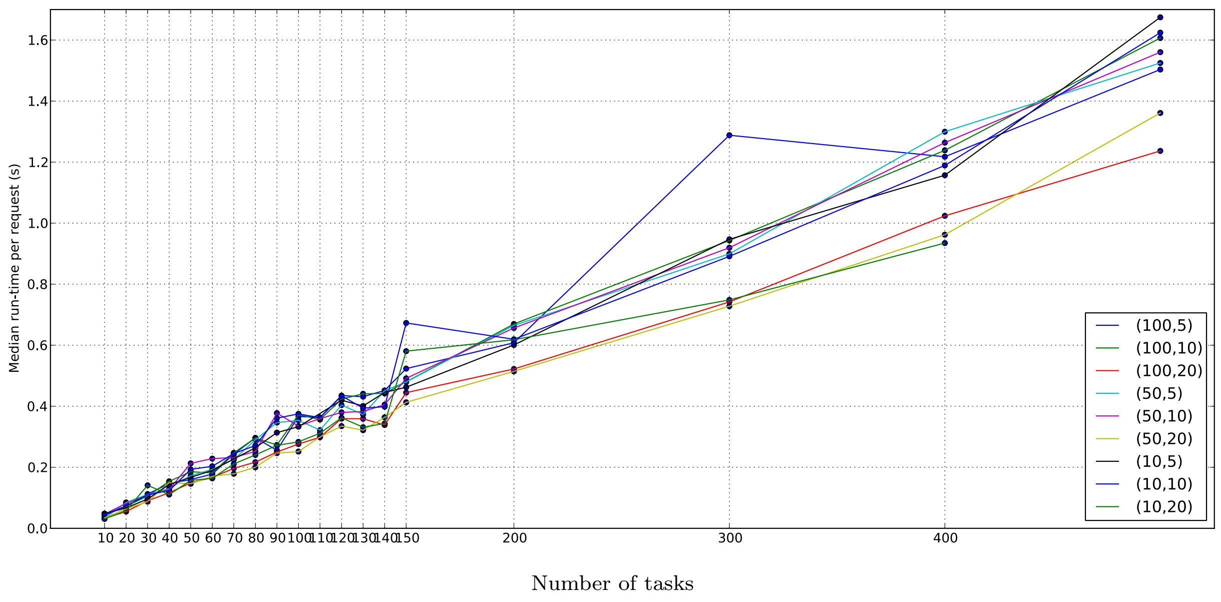


Figure 6.10: Total run-time of the on-line phase by the number of tasks in all configurations

compilation of the financial report, and its communication; tasks $t4$ and $t5$ are for checking and defining corrections. The execution of tasks $t2$ and $t5$ depends on the conditions associated to two exclusive gateways: *correct1?* (abbreviated with $c1$) and *correct2?* (abbreviated with $c2$), respectively. The SoD constraints forbid that the same user compiles a draft report and checks for errors $(t3, t4, \neq)$ or compiles the draft and defines the corrections $(t3, t5, \neq)$. □

Example 20. The goal of the ISO workflow is to plan for enough financial resources to fulfill quality requirements of an organization. It is composed of 9 tasks and 3 SoD constraints. Tasks $t1$, $t2$, ..., $t6$ involve the detailed preparation and consolidation of a draft budget, whereas tasks $t7$, $t8$ and $t9$ are for the approval of the previous activities, the integration into the total budget, and the communication of the results. The execution of tasks $t8$ or $t9$ depends on the exclusive gateway *approved?* (abbreviated with *appr*). The SoD constraints forbid that the same user prepare and consolidate a budget $(t1, t6, \neq)$, prepare and approve a budget $(t1, t7, \neq)$, or consolidate

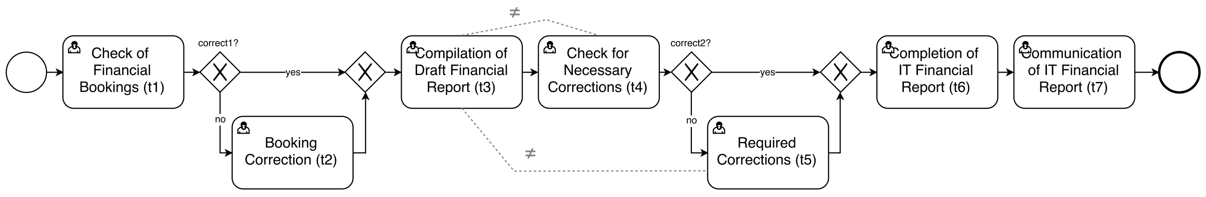

Check of Financial Bookings (t1)
correct1?
yes
no
Booking Correction (t2)
Compilation of Draft Financial Report (t3)
Check for Necessary Corrections (t4)
≠
correct2?
yes
no
Required Corrections (t5)
≠
Completion of IT Financial Report (t6)
Communication of IT Financial Report (t7)

Figure 6.11: ITIL 2011—IT Financial Reporting (abbreviated ITIL)

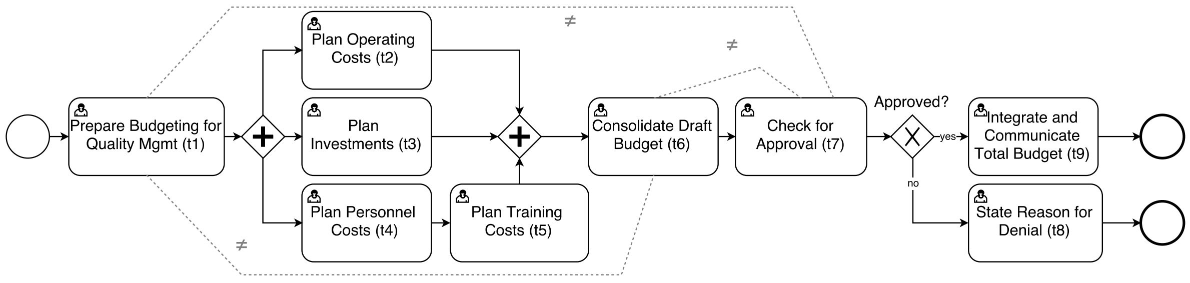

Prepare Budgeting for Quality Mgmt (t1)
Plan Operating Costs (t2)
Plan Investments (t3)
Plan Personnel Costs (t4)
Plan Training Costs (t5)
Consolidate Draft Budget (t6)
Check for Approval (t7)
Approved?
yes
no
Integrate and Communicate Total Budget (t9)
State Reason for Denial (t8)
≠
≠
≠

Figure 6.12: ISO9000—Budgeting for Quality Management (abbreviated ISO)

and approve a budget $(t6, t7, \neq)$. □

Although none of the workflows comes with an authorization policy, swimlanes (not shown in Figures 6.11 and 6.12) suggest that a controlling manager executes tasks $t1$, $t2$, $t3$, $t6$ and $t7$ while a financial manager executes tasks $t4$ and $t5$ for ITIL and that a quality manager executes tasks $t1$, ..., $t6$ and a controlling manager executes tasks $t7$, $t8$ and $t9$ for ISO. These indications are taken into consideration for designing the authorization policies (based on the RBAC model and encoded in Datalog) in various scenarios with a fixed set $U = \{u1, \ldots, u9\}$ of users. For the TRW, we consider two policies P_0 and P_1: the former is that in the Example of Chapter 3 and the latter is derived from the former in such a way that no user is authorized to execute $t1$ (thus no authorized scenario should be found). For ITIL, we have policies P_2 and P_3, each one with 3 users as financial managers, 3 users as controlling managers, and 3 with both roles; P_3 is derived from P_2 by preventing users to be able to execute task $t6$. For ISO, we consider policies P_4 and P_5, each one with 3 users assigned to the role of quality manager, 3 users as controlling managers, and 3 users assigned to both roles; P_5 is derived from P_4 by preventing users to be able to execute task $t3$.

Before invoking our algorithms for solving the SFPs, we need to build the symbolic reachability graph (and the run-time monitor) for each example. We did this by running the implementation of the off-line step from Chapter 3. For TRW, the symbolic reachability graph is computed in around a second. For ITIL, the graph is computed in around 3.5 seconds. For ISO, graph building takes around 10.5 seconds. The time for deriving the monitor M from the symbolic reachability graph of each example is negligible and thus omitted.

Table 6.1 shows the findings of our experiments for the B-SFP, MUB-SFP, and C-SFP. Each entry in column 'Instance,' describing the input to Algorithm 2 (or its modification to solve the MUB-SFP), is of the form $W + P_i$ where W is the identifier of one of the three security-sensitive workflows and P_i is one of the authorization policies described above. Column 'Γ' shows the facts in the set Γ that can be used to drive the search of execution scenarios with particular properties. For instance, ITIL contains two exclusive gateways labeled with conditions $c1$ and $c2$: we may be interested in those scenarios in which $c1$ and $c2$ take some particular truth values (see lines 1–4 and 12–15 of the table). Another use of the set Γ is shown at line 8: we are interested in finding authorized scenarios of TRW under the authorization policy P_0 in which task $t2$ is always executed by user b. There is no such scenario (the 'Solution Scenario' column reports the empty sequence ϵ) since when b performs $t2$, a must perform $t1$—because of the SoD constraint $(t1, t2, \neq)$—but if a performs $t1$, no user can perform $t4$—because of the other SoD constraint $(t1, t4, \neq)$. Column 'Time' reports the running time (in seconds) taken to find a scenario (if any).

We report the performance of the implementation of Algorithm 4, to solve instances of the SkR-SFP on the TRW, ITIL, and ISO workflows in Table 6.2. Column 'Instance' shows the workflows used in the experiments. Each one of the columns 'Time' reports the time taken (in seconds) to find

Table 6.1: Experiments for the B-SFP, MUB-SFP, and C-SFP

#	Instance	Γ	Solution Scenario	Time
			B-SFP	
0	TRW+P_0	$\emptyset$	$t1(b), t2(a), t4(a), t3(c), t5(b)$	0.288
1	ITIL+P_2	$\{c1, c2\}$	$t1(u3), t3(u9), t4(u8), t6(u9), t7(u9)$	4.267
2	ITIL+P_2	$\{c1, \textbf{not } c2\}$	$t1(u3), t3(u3), t4(u7), t5(u8), t6(u3), t7(u7)$	4.454
3	ITIL+P_2	$\{\textbf{not } c1, c2\}$	$t1(u3), t2(u1), t3(u9), t4(u8), t6(u9), t7(u9)$	4.374
4	ITIL+P_2	$\{\textbf{not } c1, \textbf{not } c2\}$	$t1(u3), t2(u1), t3(u3), t4(u7), t5(u8), t6(u3), t7(u7)$	4.561
5	ISO+P_4	$\{appr\}$	$t1(u3), t4(u7), t5(u8), t2(u3), t3(u7), t6(u9), t7(u7), t9(u8)$	6.581
6	ISO+P_4	$\{\textbf{not } appr\}$	$t1(u3), t4(u7), t5(u8), t2(u3), t3(u7), t6(u7), t7(u8), t8(u6)$	6.637
7	TRW+P_1	$\emptyset$	ϵ	0.407
8	TRW+P_0	$\{t2(b)\}$	ϵ	1.554
9	ITIL+P_3	$\emptyset$	ϵ	9.562
10	ISO+P_5	$\emptyset$	ϵ	44.076
			MUB-SFP	
11	TRW+P_0	$\emptyset$	$t1(b), t2(c), t3(b), t4(a), t5(a)$	2.385
12	ITIL+P_2	$\{c1, c2\}$	$t1(u1), t3(u1), t4(u7), t6(u1), t7(u1)$	108.819
13	ITIL+P_2	$\{c1, \textbf{not } c2\}$	$t1(u3), t3(u3), t4(u7), t5(u7), t6(u3), t7(u3)$	116.525
14	ITIL+P_2	$\{\textbf{not } c1, c2\}$	$t1(u1), t2(u1), t3(u1), t4(u7), t6(u1), t7(u1)$	108.827
15	ITIL+P_2	$\{\textbf{not } c1, \textbf{not } c2\}$	$t1(u3), t2(u3), t3(u3), t4(u7), t5(u7), t6(u3), t7(u3)$	116.533
16	ISO+P_4	$\{appr\}$	$t1(u5), t3(u5), t2(u5), t4(u5), t5(u5), t6(u3), t7(u7), t9(u7)$	166.632
17	ISO+P_4	$\{\textbf{not } appr\}$	$t1(u5), t3(u5), t2(u5), t4(u5), t5(u5), t6(u9), t7(u6), t8(u9)$	166.644

a solution for $k = 2$, $k = 4$, $k = 6$, and $k = 8$. The experiments have been run with three policy configurations, containing $n = 9$, $n = 18$, and $n = 27$ users in U (indicated with $|U| = n$ in the table). The performance results are for the version of the algorithm using the subset enumeration optimization (discussed in Section 5.3.2), since all the workflows used in the experiments contain only SoD constraints. We do not report the performance without the optimization because it is unsuitable.

Discussion

Our experiments indicate that the SFPs, their constrained versions, and the algorithms for solving them introduced in this thesis fit well with emerging practices for BPM reuse. Whenever a customer wants to deploy a business process by reusing a workflow template, some SFP is solved (if possible) to provide him or her with an authorized scenario showing that a template business process can be successfully instantiated by his or her authorization policy.

Table 6.2: Experiments for the SkR-SFP

#	Instance	Time ($k = 2$)	Time ($k = 4$)	Time ($k = 6$)	Time ($k = 8$)
$\|U\| = 9$					
0	TRW	2.612	3.209	3.990	0.504
1	ITIL	3.606	6.012	7.416	1.308
2	ISO	10.815	12.034	13.820	26.254
$\|U\| = 18$					
3	TRW	91.606	95.892	109.895	226.074
4	ITIL	114.128	159.105	161.935	165.059
5	ISO	107.864	135.725	197.564	207.941
$\|U\| = 27$					
6	TRW	680.767	712.977	815.842	841.445
7	ITIL	1252.796	1370.184	1410.400	1637.548
8	ISO	702.154	1179.483	1557.616	2185.207

The efficiency of the proposed approach exploits the fact that the eligible scenarios (resulting from execution and authorization constraints) can be computed once and reused with every authorization policy. In this way, multiple changes to a policy, which are well-known to be costly [100], become much less problematic to handle and customers can even explore and evaluate the suitability of variants of a policy. This is in sharp contrast to the approach discussed in Section 5.2 (after Example 11) that consists of re-invoking an available algorithm for solving the WSP on every task-user pair in a scenario.

To illustrate, consider the instance at line 4 of Table 6.1. Recall that the off-line step for ITIL takes around 3.5 seconds and observe that this is computed once and for all. If, instead, we use the technique to solve the WSP from Chapter 3 as a black-box (i.e., without being able to retrieve the symbolic reachability graph computed during the off-line phase), which is common to (almost) all techniques available in the literature, solving the same B-SFP would require almost 30 seconds resulting from recomputing 7 times (corresponding to the 7 task-user pairs in the returned scenario) the same symbolic reachability graph (compare this with the timing of 4.561 seconds reported in the table). This is a significant performance gain despite the small size of the example.

For each workflow, the times shown in Table 6.2 grow with the number of users ($|U|$) and desired resiliency (k) for two reasons. First, there are more absent subsets to be considered. Second, finding a scenario for each subset takes longer, since the time to answer Datalog queries depends on the number of users. For a combination of many users and a large resiliency, the time taken may be unacceptable, which is inherent to the complexity of the problem. In any case, the observation above about reusing the pre-computed symbolic reachability graph is even more important for the SkR-SFP. If the graph was not reused, it would have to be computed from scratch for each user-task pair of each absent subset. For the SkR-SFP, the optimization based on absent set domination is crucial for acceptable performance.

As an example, in line 2, column '$k = 2$', the reported time (10.815) is obtained considering only 3 dominant subsets of users, out of a total of $\binom{9}{2} = 36$. If the 36 subsets are considered, the total time is 107.724. This difference is even larger for instances with more users or a larger k.

Another advantage of Algorithm 4 is that it stops as soon as a concrete scenario is not found for a subset (line 6 in the algorithm). Notice that the times in column '$k = 8$' of lines 0 and 1 are less than the time in columns '$k = 2$', '$k = 4$', and '$k = 6$' of the same lines, since those instances are not 8-resilient. The instance in column '$k = 8$' of line 3 is also not resilient, but the algorithm took much longer to encounter a subset with no concrete scenario.

Chapter 7

Aegis: automatic enforcement of security policies in workflow-driven web applications[1]

Web applications are nowadays one of the preferred ways of exposing business processes and services to users. Many web applications implement workflows, i.e. there is a pre-defined sequence of tasks that must be performed by users to reach a goal [9].

If an application does not correctly enforce its workflows, attackers can exploit this vulnerability to subvert it. In an e-commerce application, for instance, users must *Select products*, *Checkout*, *Enter shipping information*, *Enter payment information* and *Confirm*. If the application does not verify that user actions follow this sequence, a user can, e.g., skip the payment step and receive products without paying. Even simple navigation errors, when a user accesses pages in an unexpected order, if handled improperly, can be exploited [74].

Workflow and business logic vulnerabilities are listed in the Common Weakness Enumeration (CWE)[2], in the OWASP Testing Guide [108] and (tangentially) in the OWASP Top 10 [117]. Enforcing the correct workflow

[1]Parts of this chapter were previously published in [32]

[2]`https://goo.gl/yH4xrP` and `https://goo.gl/bFvzjK`

of an application is known as control-flow integrity and it has been used in web applications to prevent workflow attacks and others, such as forceful browsing and race conditions [19, 20].

Data-flow integrity is also crucial and incorrect enforcement can lead to vulnerabilities where, e.g., a user can change the price of a product being purchased to pay less for it [161]. This kind of vulnerability is even more prominent in multi-party scenarios, where a user receives data objects, such as tokens from one party (e.g., an identity provider) and must relay them to another party (e.g., a service provider). Several vulnerabilities have been discovered in recent years due to improper enforcement of data-flow integrity [161, 119, 140].

Besides control- and data-flow integrity, access control is fundamental for web application security whenever users must access only data and functionalities that they are authorized to by a given policy. Access control vulnerabilities are common and hard to find [141]. Moreover, some web applications implement collaborative work, in which many users work together to achieve the goal of a workflow. Examples are Enterprise Resource Planning (ERP) software, used by employees in an organization to manage, e.g., purchases, sales and finance; and e-health applications, used by doctors and technical staff to manage patient records. In these applications, not only it is important to enforce authorization policies, but it may also be necessary to support authorization constraints, such as BoD or SoD. A valuable use of these constraints is to avoid errors and frauds in security-critical applications that must follow regulatory compliance rules.

Nonetheless, none of the applications we experimented with provided support for an easy to use, declarative specification of constraints. Including Odoo[3], an open-source ERP platform with more than 5000 developers and 2 million users, among them big companies such as Toyota and Danone.

[3]`https://www.odoo.com/`

Without declarative specifications and proper enforcement, authorization constraints have to be implemented as application code and embedded into each task [14] or translated to static assignments in the authorization policy. Both solutions are error-prone and can hardly scale.

Even with suitable specification and enforcement mechanisms, support for authorization policies and constraints may lead to situations where an application workflow cannot be completed because no user can execute an action without violating them, emphasizing the need for solutions to the WSP.

The WSP has received much attention in the workflow security community [78], but, to the best of our knowledge, has never been considered in web applications. In fact, transferring WSP solutions to the web domain is not trivial. These solutions often rely on a workflow model specification and a workflow management system to handle the control-flow of tasks and to provide an interface for users to request task executions, elements which are frequently not available for web applications.

In this Chapter, we present Aegis[4], a technique to synthesize run-time monitors for web applications that are capable of automatically (i) enforcing security policies composed of combinations of control- and data-flow integrity, authorization policies, and authorization constraints; and (ii) solving the run-time version of the WSP by granting or denying, at run-time, requests of users to perform tasks based on the satisfaction of the policy and constraints and the possibility to terminate the current workflow instance.

To synthesize a monitor, Aegis first infers, using process mining [144], workflow models of the target application from a set of HTTP traces representing user actions. Traces must be manually edited to contain only

[4]Aegis was the mythological shield carried by Athena, and “under the aegis of” means “under the protection of.”

actions that should be controlled by the monitor. Inferred models are Petri nets [111] labeled with HTTP requests representing tasks and annotated with data-flow properties obtained by using a set of heuristics based on differential analysis (as e.g., [161, 140]). These Petri nets (or a more user-friendly representation, such as BPMN) can be refined by a human user who, optionally, specifies authorization constraints and an authorization policy. A monitor is then generated from a model by invoking the synthesis technique described in Chapter 3.

At run-time, a reverse proxy is used to (i) capture login actions to later establish the acting users, and (ii) capture incoming requests and query the monitor to determine whether to allow or deny the request. Aegis is completely black-box and can be used with new or legacy applications to add support for the enforcement of security-related properties or to mitigate logic vulnerabilities.

7.1 Overview

Aegis synthesizes run-time monitors for workflow-driven web applications, i.e. applications implementing business processes and customer services as workflows. Hereafter, *web application* is used as an abbreviation for *workflow-driven web application*, unless stated otherwise.

A monitor synthesized by Aegis can enforce three security-related properties in web applications: authorization policies ($\mathcal{P}$), defining which users are entitled to perform which tasks; authorization constraints ($\mathcal{C}$), defining run-time restrictions on the execution of tasks, e.g., a SoD requiring two different users to perform a pair of tasks; and control- and data-flow integrity ($\mathcal{I}$), specifying the authorized control-flow paths that the application must follow, as well as data invariants.

Different web applications have different enforcement needs, which allows for the synthesis of different configurations of monitors, depending on which properties are switched on or off. We identify each configuration as a tuple containing the active properties, e.g., $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$, $\langle \mathcal{P}, \mathcal{I} \rangle$, $\langle \mathcal{C}, \mathcal{I} \rangle$, $\langle \mathcal{I} \rangle$. Control- and data-flow integrity are in the same category because it is not realistic that an application needs to enforce one and not the other.

AEGIS takes as input sets of HTTP traces representing user actions executed while interacting with a target web application. It synthesizes and outputs an external monitor composed of a set of queries to be used by a proxy sitting between users and the application. Each set of input traces is produced by a user simulating real clients completing a workflow as foreseen by the application ("good traces"). Traces can be collected using test automation tools such as Selenium[5] or OWASP ZAP[6] and must be manually edited to contain only critical tasks. After trace collection, the whole technique is automated.

The monitor only enforces those workflows given as input by the user, having no impact on the rest of the application besides the overhead of a reverse proxy (which is frequently used in any case to implement, e.g., load balancing).

Figure 7.1 shows an overview of AEGIS. The top of the Figure shows the entire approach, where rectangles represent the three main steps (with sub-steps), yellow notes represent inputs, and ovals represent generated artifacts. The bottom of the Figure details the internals of the *Run-time Monitoring* component. The three main steps are the following.

1. Model Inference. The set of *HTTP traces* is automatically *stripped* of all information except request and response URLs, headers, and bodies; each request and response is *annotated* with *data-flow properties* inferred by a set of heuristics; traces are *aggregated* into a file called event log; and

[5]`http://www.seleniumhq.org/`
[6]`https://goo.gl/XvxKd1`

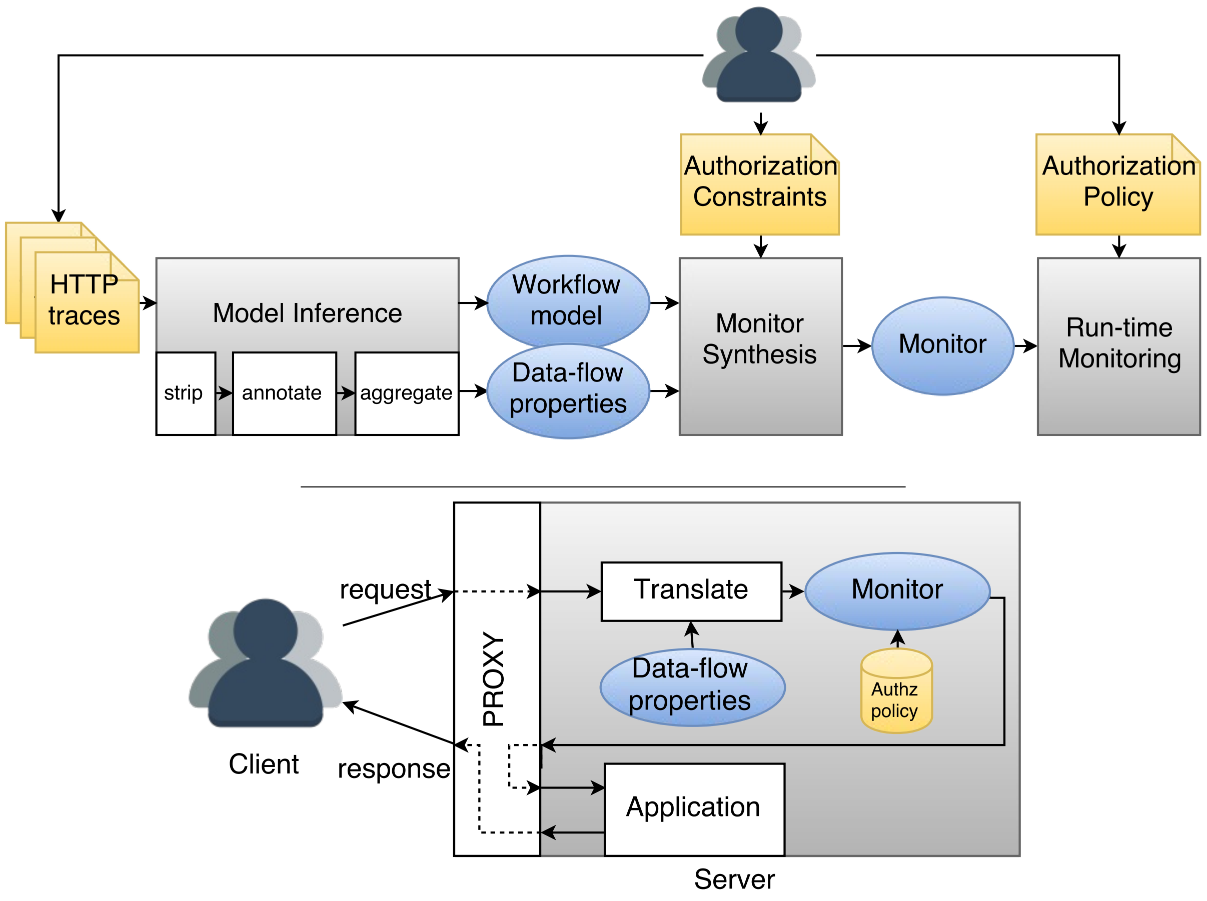


Figure 7.1: Overview of the technique

a process mining [144] tool takes the log as input to generate a Petri net *workflow model* whose transitions are labeled by the annotated requests. The inferred model can be refined according to the user's understanding of the application.

2. Monitor Synthesis. Given a workflow model, the user specifies the *Authorization Constraints* to be enforced (if any) and whether an *Authorization Policy* will be provided at run-time. Control- and data-flow integrity are obtained automatically from the model, can be optionally modified by the user, and are always enforced. The workflow model can be presented to the user in a convenient notation such as BPMN, and the specification of constraints can be done graphically. A run-time *Monitor* capable of enforcing the chosen properties is synthesized by invoking the tool described in Chapter 3.

3. Run-time Monitoring. A reverse *proxy* is instantiated with the synthesized monitor and a concrete authorization policy, if required by the application. As shown at the bottom of Figure 1, it sits between users and the application, filters *requests* and *translates* them to the monitor. The monitor enforces the properties defined in step 2, granting a request if the control-flow is respected, the data-flow invariants hold, the user issuing the request is authorized by the policy, the authorization constraints are not violated and the current instance execution can still terminate. The proxy, based on the response from the monitor, may forward requests to the application or drop them to prevent the violation of some property.

A single application may implement several workflows. Steps 1 and 2 are performed for each workflow to be monitored, generating one monitor for each workflow. Step 3 uses all the synthesized monitors and queries the correct one depending on the incoming request. Requests not related to any monitored workflow go direct to the application, without triggering a monitor query.

Below, we present two motivating examples that illustrate the configurations $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$, $\langle \mathcal{C}, \mathcal{I} \rangle$ (first example), and $\langle \mathcal{I} \rangle$ (second example). The first example motivates the distinctive contributions of our technique: support for authorization policies, constraints, and workflow satisfiability. The second example shows that Aegis is capable of mitigating logic vulnerabilities related to control- and data-flow integrity.

7.1.1 Example 1 - Enforcing constraints

Dolibarr[7] is an open-source ERP web application for small and medium enterprises. It implements, among others, a business process similar to

[7] `http://www.dolibarr.org/`

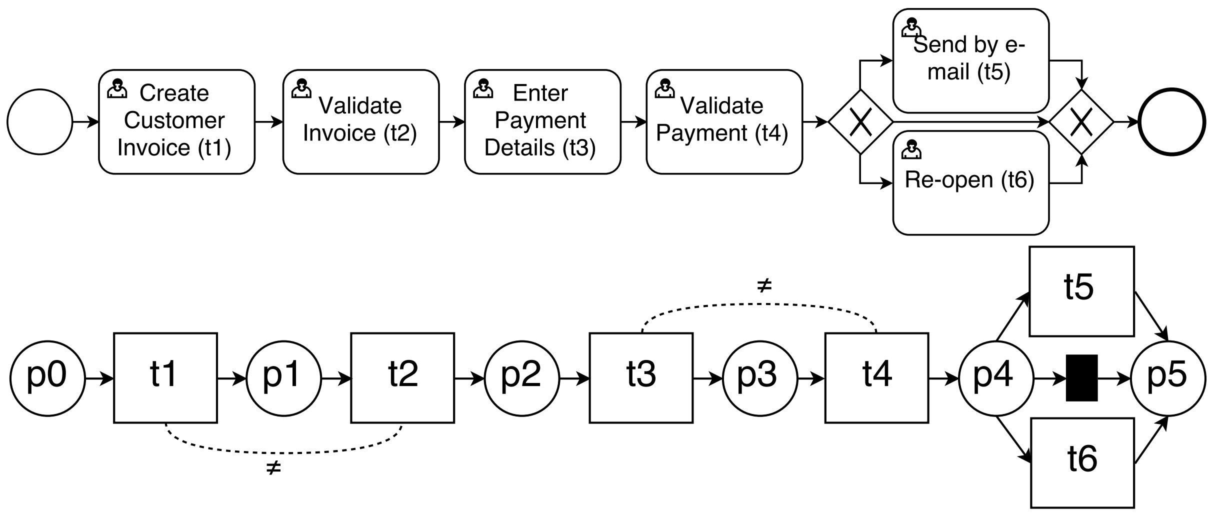


Figure 7.2: Customer invoice process in BPMN (top) and as a Petri net (bottom)

the one shown at the top of Figure 7.2 (in BPMN) to manage customer invoices.

The process contains 6 tasks. Tasks $t1$ to $t4$ must be performed in sequence, while either $t5$, $t6$ or neither are performed last. The original application implements each of the tasks shown in Figure 7.2. An authorization policy, control-flow, and possible data-flow invariants are implemented in an ad-hoc way, whose correctness is hard to verify, which may lead to vulnerabilities. The authorization policy originally supported by the application has a granularity of permissions that does not match the user-task assignment we support (there is no specific permission to, e.g., re-open an invoice or send it by e-mail). Authorization constraints are not supported. As a result, it is not trivial to prevent a malicious user from creating and validating a customer invoice (SoD between $t1$ and $t2$) or inserting and validating a payment (SoD between $t3$ and $t4$), which would allow him to, e.g., close invoices with an incorrect payment.

A user who wants to securely deploy this application can use Aegis to generate a $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ monitor to enforce control-flow integrity, ensuring that all the steps in the customer invoice process are performed in the correct order; an authorization policy, ensuring that only authorized users can execute each task; and the SoD constraints described above, to avoid frauds. If the user prefers to leave authorization enforcement to the application, a $\langle \mathcal{C}, \mathcal{I} \rangle$ monitor could be generated to only add support for constraints and integrity.

To generate a monitor for the invoicing process, without impacting other parts of the application, the user starts by collecting traces simulating users performing the process. Some HTTP traces representing these executions are[8]:

$$\begin{aligned}
\tau_1 = \ & \{\texttt{/invoice?action=create\&value=10\&prod=abc},\\
& \texttt{/invoice/validate?id=1}, \texttt{/invoice/pay/create?id=1\&value=10},\\
& \texttt{/invoice/pay/validate?id=1}\},\\
\tau_2 = \ & \{\texttt{/invoice?action=create\&value=20\&prod=def},\\
& \texttt{/invoice/validate?id=2}, \texttt{/invoice/pay/create?id=2\&value=20},\\
& \texttt{/invoice/pay/validate?id=2}, \texttt{POST /invoice/send BODY id=2}\},\\
\tau_3 = \ & \{\texttt{/invoice?action=create\&value=30\&prod=ghi\&prod2=jkl},\\
& \texttt{/invoice/validate?id=3}, \texttt{/invoice/pay/create?id=3\&value=30},\\
& \texttt{/invoice/pay/validate?id=3}, \texttt{/invoice/reopen?id=3}\}.
\end{aligned}$$

Each trace τ_i represents one possible execution of the invoicing process and each request represents one task. The first four requests in each trace are essentially the same, but with different parameter values (e.g., `id` is 1 in τ_1, 2 in τ_2, and 3 in τ_3). They represent tasks $t1$, $t2$, $t3$, and $t4$. τ_1 is an example of the branch where only the first four tasks are executed, while $t5$ is executed after $t4$ in τ_2, and $t6$ is executed after $t4$ in τ_3.

The traces are automatically analyzed to extract data-flow properties, annotated and aggregated into an event log, sent to a process mining tool

[8]for the sake of readability, we show only simplified URLs, but headers and body are also part of the traces.

and the resulting Petri net labeled with a task-to-URL map (**Step 1**). Figure 7.2 shows, at the bottom, the Petri net obtained from the process mining tool (ignore for a moment the dashed lines). The tasks in the net are labeled as t_i, with the following task-to-URL map:

t_1 : `/invoice?action=create&value=<<I>>&prod=<<DC>>`,

t_2 : `/invoice/validate?id=<<IID>>`,

t_3 : `/invoice/pay/create?id=<<IID>>&value=<<I>>`,

t_4 : `/invoice/pay/validate?id=<<IID>>`,

t_5 : `POST /invoice/send BODY id=<<IID>>`,

t_6 : `/invoice/reopen?id=<<IID>>`

Data-flow properties are represented by annotations on the URLs. The `<<IID>>` (instance identifier) annotation is applied to the elements used to bind all the requests to the same instance of a workflow, in this case the `id` parameter. The `<<I>>` (invariant) annotation is applied to values that should not change during the whole workflow, in this example the `value` of the invoice in t_1 should be the same as the `value` of the payment in t_2. The `<<DC>>` ("don't care") annotation is applied to parameters that should be present in the request to help identify it as a unique action, but whose values are irrelevant. The parameter `prod2`, which is present in the request of $t1$ only in τ_3, is dropped in the task-to-URL map because it is considered optional, i.e., a trace may represent an invoice with one or more products, so only the first `prod` parameter needs to be present. These data-flow properties, as well as others not used in this example, are obtained by using heuristics detailed in Section 7.2.

The user then specifies the constraints that must be enforced, shown as dashed lines labeled by $\neq$ in Figure 7.2. The model is used to synthesize a monitor (**Step 2**), which is composed of a set of SQL queries like

```
1 SELECT U2.ID FROM USERS AS U1, USERS AS U2, HST WHERE
2 HST.dt1 AND NOT HST.dt2 AND NOT HST.dt3 AND
3 NOT HST.dt4 AND NOT HST.dt5 AND NOT HST.dt6 AND
4 NOT HST.t1by = U1.ID AND U2.ID IN (SELECT * FROM
5 TA2) AND U1.ID IN (SELECT * FROM TA3) AND U2.ID IN
6 (SELECT * FROM TA4) AND U1.ID IN (SELECT * FROM TA5)
```

encoding the fact that, to perform task $t2$, only task $t1$ must have been executed (lines 2 and 3), there must be an authorized user $u2$ who has not performed $t1$ (line 4), and there must be other users capable of executing the remaining tasks (lines 5 and 6). This query is for the $t5$ branch, there are similar queries for the $t6$ branch and the branch where neither is executed, as well as other queries using a different number of users.

At run-time (**Step 3**), in the $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ configuration, a policy is specified as a task-user assignment, e.g., $TA = \{(u1, t1), (u1, t2), (u2, t2), (u3, t3), (u4, t3), (u4, t4), (u5, t5), (u6, t6)\}$, where $(u, t) \in TA$ means that user u is authorized to execute task t. The assignment is stored in the appropriate tables in the database, and a reverse proxy is instantiated with the synthesized monitor. The proxy is capable of receiving a request such as

```
GET /invoice/validate?id=5  Cookie:  sid=abcd1234
```

and identifying that it refers to task $t2$ of instance 5 of the invoicing process being performed by user $u2$ (whose cookie `sid`, sent in the header, has been stored during login). It then queries the monitor and, assuming that $u1$ has previously executed $t1$ and $t2$ has not yet been executed, the query presented above is satisfied and the request is granted. On the other hand, a request can be blocked in several cases, such as if $u3$ tries to execute $t2$ ($(u3, t2) \notin TA$, i.e., he is not authorized according to the authorization policy), if $u1$ tries to execute $t2$ (there is a SoD between $t1$ and $t2$), if any user tries to execute $t3$ before $t2$ (because of the control-flow), or if any user issues a request for $t3$ with a `value` different from the one sent for $t1$ (because of the invariant).

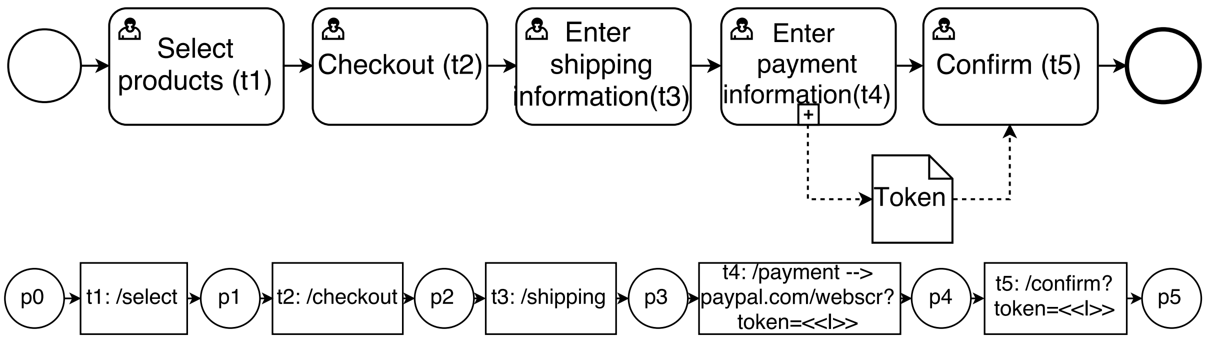

Select products (t1)
Checkout (t2)
Enter shipping information(t3)
Enter payment information(t4)
Confirm (t5)
Token
p0
t1: /select
p1
t2: /checkout
p2
t3: /shipping
p3
t4: /payment --> paypal.com/webscr? token=<<l>>
p4
t5: /confirm? token=<<l>>
p5

Figure 7.3: Checkout process in BPMN (top) and as a Petri net (bottom)

To solve the WSP, regardless of the execution history, any request of $u4$ to execute $t3$ should also be blocked. Granting that request would mean that the only user authorized to execute $t4$ has already executed $t3$, while both tasks are in SoD. Therefore, any execution where $u4$ performs $t3$ would either not terminate or terminate with the violation of some constraint or policy.

The synthesized monitor presents a transparent way of avoiding this situation by blocking requests that lead to an undesired outcome. Exceptional situations, where it is preferable to violate the policy with the knowledge of an administrator, can be accommodated by using a soft enforcement mode, as discussed in Section 7.4.

7.1.2 Example 2 - Mitigating vulnerabilities

TomatoCart[9] is a popular e-commerce application that implements the checkout process depicted on the top of Figure 7.3. It is composed of 5 tasks executed in sequence, where $t4$ is a sub-process that can be implemented in different ways, but must produce a data object representing a token issued by a trusted third party, that is read in $t5$.

This is an example of a multi-party web application [140], which imple-

[9]http://www.tomatocart.com/

ments the payment step by using third-party solutions such as PayPal[10]. An execution of this workflow, using PayPal Express Checkout, involves three actors, a client C, a service provider SP implementing TomatoCart and a trusted third party TTP implementing the payment provider. The execution starts with the client browsing the SP, selecting some product ($t1$), requesting checkout ($t2$), and entering shipping information ($t3$). The SP then contacts the TTP and receives a token identifying the transaction (not shown in the workflow). The user is redirected to the TTP with the token ($t4$), completes the payment (again not shown in the Figure), and is redirected back to the SP passing the token, which is verified to complete the transaction ($t5$).

In version 1.1.7, TomatoCart had a vulnerability that allowed users to replay a PayPal Express Checkout token in $t5$ of a new transaction and shop for free [140]. This vulnerability was manually fixed in a later release of the application, but Aegis could have been used to mitigate it until a patch was available (or until the patch could be applied, which is not always trivial).

To mitigate the replay vulnerability, we can generate a monitor in the configuration $\langle \mathcal{I} \rangle$, enforcing control-flow integrity and the data invariant that the token received in $t4$ is the same that is sent in $t5$. An authorization policy and authorization constraints are not specified since every user can execute the steps in the checkout process and all steps are executed by the same user. The details of the communication between SP and TTP and between C and TTP are not shown in the workflow because the monitor only needs to enforce that no user can replace the token that has been sent to him/her. Although Aegis ignores some messages, many vulnerabilities in multi-party web applications can be mitigated this way [161].

To generate the monitor, we repeat the steps presented for Example 1.

[10]`https://www.paypal.com/`

Below, there are some traces of the execution of the TomatoCart check-out process, again simplified for readability. Now the traces involve three parties, thus each request must be identified with its host.

$$\begin{aligned}\tau_1 = \{&\texttt{shop.com/select}, \texttt{shop.com/checkout}, \texttt{shop.com/shipping},\\ &\texttt{shop.com/payment --> paypal.com/webscr?token=abcd1234},\\ &\texttt{shop.com/confirm?token=abcd1234}\},\\ \tau_2 = \{&\texttt{shop.com/select}, \texttt{shop.com/checkout}, \texttt{shop.com/shipping},\\ &\texttt{shop.com/payment --> paypal.com/webscr?token=efgh5678},\\ &\texttt{shop.com/confirm?token=efgh5678}\}.\end{aligned}$$

Figure 7.3 shows the Petri net obtained for the checkout process, labeled directly with the URL of each task (where `-->` represents a redirect). The invariant annotation `<<I>>` is applied to the `token` received from PayPal, specifying that its value must be the same in `/payment` and `/confirm`.

A monitor is synthesized as before, however with neither authorization policy nor constraints. Workflow instances can be identified by the user identifier, since each user has only one checkout process at any given time.

At run-time, whenever a user tries to replay a token, the monitor blocks this request because the token sent in $t5$ is different from the one received in $t4$ (since PayPal generates unique tokens). If the user tries to bypass the monitor by skipping step $t4$ and sending the token directly in $t5$, the monitor blocks the request because of a control-flow violation.

7.2 Details

An HTTP trace (or a web session) is a sequence $S = \{(u_1 : r_1, s_1), (u_2 : r_2, s_2), \ldots, (u_n : r_n, s_n)\}$ of pairs of web requests r_i issued by users u_i (which may or may not be all distinct) and responses s_i. Each web request or response is defined as $r_i = (\mathit{headers}, \mathit{body})$ and the information we derive from a request is a tuple $(method, url, P)$, where $method \in \{\text{GET, POST}\}$, *url* is the requested URL, and P is a set of parameters of the form (k, v), which can be in the URL (in GET requests), the body (in POST requests) or in the headers (e.g., cookies or *Location* in redirects). Data values passed as JSON can also be flattened to the same representation. The parameters in P represent the data values later annotated with data-flow properties.

A workflow $W(T, U)$ is a series of tasks $(t \in T)$ in a causal order executed by a set of users $(u \in U)$, and a web application is composed of a set of workflows $\Psi = \{W_1(T_1, U_1), \ldots, W_n(T_n, U_n)\}$. We take as input sets of web sessions $WS = \{S_1, S_2, \ldots, S_n\}$ and infer from each a workflow $W_i(T_i, U_i)$ using a process mining function $\mathcal{PM}$, and a set of data property labels L_i using heuristics. We also take as input, optionally, sets of authorization constraints C_i. We then use a monitor synthesis procedure $\mathcal{MS}(W_i, L_i, C_i)$ that returns a monitor M_i. M_i is capable of answering requests of the form "can user u perform task t?"—encoded as $can\_do(u, t)$—with True iff the control-flow in W_i and the data-flow in L_i are respected, no authorization constraint in C_i is violated, the requesting user u is authorized by an authorization policy *TA* (specified at run-time), and the workflow can be executed until the end.

Attacks and enforcement

At run-time, a reverse proxy $\mathcal{RP}$ receives an incoming request $u : r$ and based on the information taken from it, tries to translate it into a query of the form $can\_do(u, t)$, for $u \in U_i$ and $t \in T_i$ of workflow $W_i(T_i, U_i)$, which can be answered by M_i.

Attacks on the application at the level of web requests are reflected on the workflows [98] as shown below. The monitor can mitigate these attacks because they do not comply with the expected workflow (naturally, they are only mitigated in the parts of the application covered by the inferred model).

A request forgery is an extra request not foreseen in a workflow ($\{r_1, r_2, \ldots, r_{forged}, \ldots, r_n\}$). A workflow bypass is a missing request ($\{r_1, r_2, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n\}$). A workflow violation is an attempt to either repeat a unique request ($\{r_1, r_2, \ldots, r_i, \ldots, r_i, \ldots, r_n\}$) or execute a request out of order ($\{r_1, r_2, \ldots, r_{i+1}, \ldots, r_i, \ldots, r_n\}$). Authorization violations happen when a request is issued by a user who is not entitled to do so by the policy or when, for two tasks t_1 and t_2 in SoD, a user who previously issued a request r_1 to execute t_1, issues a new request r_2 to execute t_2.

Adversary model

The monitor M enforces security properties related to access control and control- and data-flow integrity, ignoring vulnerabilities such as code injection. We trust the target application, as well as any third parties trusted by it. Application users are not trusted, since they can be partially or fully controlled by an adversary.

7.2.1 Step 1 - Model inference

The goal of Aegis is not to produce an accurate model of the whole application, but only workflow models composed of a sequence of critical actions. These critical actions are the requests related to workflow tasks, whose execution should be controlled by the monitor.

The definition of what is critical varies from application to application, but besides the usual noise in HTTP traces (e.g., requests loading images and other resources), any request that leaves the application state unchanged (e.g., an AJAX request triggered for auto-completion of an input field) should be filtered out. Such requests are called navigation events, as opposed to system-interaction events, which change the state of the application [136]. Not every system-interaction event should be controlled by the monitor (this should be decided by the user). However, discarding navigation events is crucial to keep the inferred models to a reasonable size and to eliminate imprecision due to variations in the process when executed by different users.

We assume that this treatment of the input traces is done before Aegis is invoked. It can be done manually by deleting unimportant actions, but there are proposals of automated black-box techniques to detect state changing requests.

Some techniques detect a state change by sending identical requests and comparing their outputs [55, 57], others are based on an abstraction of the user interface [136]. The former have limitations such as the need to isolate the application (other users cannot interact with it while a trace is collected) and to be able to reset it to its initial state. The latter cannot detect system states that are not reflected in the UI. Such techniques are usually embedded in crawlers to obtain a model of the entire application. Applying just the state-change detection part to traces of a single workflow may have sub-optimal results. Evaluating similar approaches to automate trace collection is left to future work.

Since some URLs in an application can take different parameters and different values for these parameters, while still representing the same action, and since we apply differential analysis to identify data-flow properties, we need at least two different traces as input, each containing a possible value for each of the parameters (including their presence and absence). The input traces should also represent all the possible execution paths of the process (control-flow). The number of input traces required for a precise model depends on the number of control-flow branches in the workflow being analyzed, as well as the diversity of the traces. Related works use, e.g., four traces as input [161] or traces with specific requirements for each of the parties in the process [140].

At least two login traces with distinct users must also be present, so that cookies defining the user session identifier and parameters representing user names can be mined, to map requests to concrete users at run-time.

It is possible to obtain input traces by reusing functional tests, which are common in web development and usually implemented using a framework such as Selenium. From the set of HTTP traces, we extract three artifacts: a workflow model, a task-to-URL map, and a set of data properties.

Workflow model and map

A workflow model is automatically obtained from a process mining tool. There are many well-known process mining algorithms and AEGIS uses the α-algorithm [144]. It mines workflow nets by recording all the events in a log and detecting relations between them, such as sequence ($\rightarrow$), exclusive or ($\#$), and parallel ($||$).

In the traces used in Example 1, it is possible to see that $t1$ always precedes $t2$ and $t2$ never precedes $t1$, so the algorithm infers a causal dependency between them and adds a place connecting transitions $t1$ and $t2$ in the output net (place $p1$ in Figure 7.2). It is also possible to see that $t4 \rightarrow t5$ ($t4$ precedes $t5$), $t4 \rightarrow t6$ ($t4$ precedes $t6$), and $t5\#t6$ ($t5$ and $t6$ do not happen in the same trace), thus the algorithm creates a place after $t4$ that branches the execution ($p4$ in the same Figure).

Since the input traces contain only relevant URLs and each unique URL becomes a transition after process mining, the task-to-URL map is trivial to obtain.

Data-flow properties

Identification and annotation of data properties has been used initially in [155] and later in other works [161, 119, 140]. Each work proposes their own annotations.

We use five annotations, namely *constant*, *don't care*, *invariant*, *instance identifier*, and *user identifier*, which are used for three goals. *Constants* and *don't cares* are used to restrict and generalize, respectively, the input traces by fixing or hiding given values that are used to match incoming requests at run-time. A *user identifier* is used to detect the user issuing a request and an *instance identifier* to detect the workflow instance that the request is related to. This is because several instances of the same workflow may be running at the same time and they may have different execution histories (e.g., an instance 1 of the invoicing process where only $t1$ was executed by $u1$ and an instance 2 where both $t1$ was executed by $u1$ and $t2$ was executed by $u2$). *Invariants* indicate values that should not be modified during a workflow instance execution.

Data-flow properties are obtained by using differential analysis, i.e., comparing the differences in the data values between traces, as is done in related work (e.g., [161, 140]). For each trace, the analysis compares the values of all parameters in each request in relation to (i) the same parameter in other requests of the same trace, (ii) the same parameter in other traces, (iii) other parameters in the same trace, and (iv) other parameter in other traces.

Aegis does not apply syntactic annotation (as, e.g., [140]) to identify the data type of each parameter, and does not try to discover possible values or intervals for data elements, because it does not enforce particular values that were seen in the traces (except for *constants*). Below, we describe the differential analysis used to identify each kind of data-flow property.

Let WS be the set of traces τ_i used for analysis, each τ_i be composed of requests r_j and responses s_j, and each request or response have a set P of parameters (k, v). Considering the same request r_j in every trace $\tau \in WS$, if a parameter (k, v) appears in only a strict subset $\tau' \subset \tau$ of the traces, it is considered optional and ignored, i.e. dropped from the URL in the labeling function L. *Constants* are parameters that are present in every trace $\tau \in WS$ for the same URL of a request r_j and whose key k and value v never change. An example is the parameter `action=create`, which is in $t1$ of traces τ_1, τ_2, and τ_3 in Example 1. *Don't cares* are parameters that appear in every trace $\tau \in WS$ for the same URL of a request r_j and whose key k remains constant, but whose value v is different in at least one of the requests. One example is `prod=abc`, `prod=def` and `prod=ghi` in $t1$ of Example 1 annotated as `prod=<<DC>>`. An *instance identifier* is a key k whose value v is present in every request r of a trace τ, with different v's in every trace. In Example 1, the parameter `id` is an instance identifier, since it has the value 1 in every request of τ_1, the value 2 in every request of τ_2, and the value 3 in every request of τ_3. Notice that what must remain constant is the value and not the key, so it is possible to have an instance identifier called, e.g., `id` in one request and `iid` in another request. A *user identifier* is a parameter that comes from a response issued by the server, is stored in a cookie, sent in every request of a trace and whose value changes in every trace in WS. In Example 1, only URLs are shown in the traces, but the cookie `sid` is sent with every request, as can be seen towards the end of the example. *Invariants* are values v that remain constant during a trace, change between traces in WS and are not present in every request of a trace (as opposed to instance identifiers). Two examples are the `value` parameter in $t1$ and $t3$ in Example 1 and the `token` in $t4$ and $t5$ of Example 2. Like instance identifiers, invariant *values* should not change, but their *keys* might, so that an invariant can be called, e.g., `price` in one request and `amount` in another. There may be many invariants in a workflow, so they are annotated as `<<I_1>>`,`<<I_2>>`, for run-time enforcement (there may be several don't cares too, but they are not enforced and do not need separate annotations).

The result of Step 1 is a tuple (PN, L), where PN is a Petri net obtained from process mining and L is a labeling function that associates to each transition in the net a URL annotated with the identified data properties.

Although the inferred model (PN, L) is obtained automatically, it can be edited by a user before being sent for monitor synthesis. Control-flow constraints can be changed by graphically adding or removing places or transitions in the Petri net (or tasks and gateways in BPMN), while data properties can be modified by adding or removing annotations on the URLs.

7.2.2 Step 2 - Monitor synthesis

Monitor synthesis takes as input the tuple (PN, L) obtained from Step 1 and, optionally, augments it with security properties given by the user. The user can specify a set of authorization constraints and indicate whether the monitor should enforce an authorization policy, which will be specified at run-time.

A symbolic transition system is obtained from the augmented tuple and sent to a model checker, which computes the reachability graph containing all valid executions of the workflow.

Security properties specification

All behaviors of the web application that satisfy the specified security properties (namely those deriving from authorization policies and constraints) are represented by the executions of a symbolic transition system $S = (V, Tr)$, where V is a set of state variables and Tr is a set of transitions (as described in Chapter 3). The enforced properties, and as a consequence the variables in V, fall into four categories. Therefore, V can be seen as the union of four disjoint sets V_{CF}, V_{DF}, V_C, and V_A, explained below.

First, *control-flow* constraints (involving state variables from the set V_{CF}) are automatically derived from the Petri net PN by using Boolean variables p_i's, one for each place in PN, indicating the presence or absence of tokens in these places; and Boolean variables d_{ti}'s, one for each task, representing the fact that a task has been executed or not.

Second, data values of parameters annotated as invariants are represented by variables v_i and g_i (g stands for ghost) in V_{DF}. Data types are abstracted, so every v_i and g_i is represented by a natural number.

Third, the set V_C contains Boolean functions h_t's, one for each task, keeping track of which user has executed task t. The functions start with a value False for every transition and user and are updated after each task execution. *Authorization constraints* of the form $(tx, ty, \neq)$ are represented by an enabling condition $\neg h_{tx}(u)$ in transition ty. BoD constraints can be encoded in a similar way as SoD.

Fourth, an *authorization policy* is represented by constraints on Boolean functions a_t's, one for each task, that involve state variables from the set V_A and return True iff a user is authorized to perform task t. The functions a_t's are an interface to the authorization policy that is provided at run-time.

Transitions in Tr have the shape

$$t(u) : en_{CF} \wedge en_C \wedge en_A \rightarrow act_{CF} || act_C || act_A || act_{DF}$$

where $t(u)$ is an identifier, the en's are predicates on the state variables in V representing the *enabling conditions* of the transitions (in terms of control-flow, constraints, and authorization policy, respectively), and the act's are parallel ($||$) assignments to the variables in V representing the *effects* of executing a transition (again, for each security property). Data variables in V_{DF} are not used in the conditions, only in the assignments of transitions that contain data invariants, as $g_i := v_i$. The values of v_i are taken as input at run-time.

As an example, the transition for task $t2$ in Example 1 is

$$t2(u) : p1 \land \neg d_{t2} \land \neg h_{t1}(u) \land a_{t2}(u) \rightarrow p1, p2, d_{t2}, h_{t2}(u) := F, T, T, T$$

indicating that, for this transition to be executed, there must be a token in $p1$, $t2$ should not have been executed ($\neg d_{t2}$), the user u should not have executed $t1$ ($\neg h_{t1}(u)$) and the same user u should be authorized to execute $t2$ ($a_{t2}(u)$); the result of its execution is that a token is removed from $p1$, placed in $p2$ and the functions d_{t2} and h_{t2} are updated to record that $t2$ has been executed and user u has executed $t2$, respectively. Since $t2$ does not contain invariants, there is no assignment to data values. However, $t1$ contains $g_1 := v_1$ in the update, where the value of v_1 will be taken as the value of parameter `value` of the incoming request at run-time.

Monitor synthesis

The transition system S is fed to a symbolic model checker, which is responsible for computing a reachability graph RG representing all possible executions of the workflow by a set of symbolic users.

A procedure $\mathcal{MS}$ to compute this graph is described in Chapter 3. A Datalog program M is derived from RG by generating a clause of the form $can\_do(u, t) \leftarrow \beta_n$ for each node n in the graph. An invariant $d_t \Rightarrow v_i = g_i$, for every v_i in the assignments of transition t, is conjoined with each clause β_n. This invariant specifies that after the execution of each transition the value of a variable remains the same as the value of its respective ghost variable.

M is then translated to SQL and the SQL program is capable of answering—after being instantiated with a concrete authorization policy—user requests to execute tasks in a workflow in such a way that the authorization and execution constraints are not violated, the authorization policy is respected and termination of the workflow is guaranteed, thus enforcing the specified security properties and solving the run-time WSP.

The result of the monitor synthesis step is a tuple (M, L), where M is the monitor generated from RG and L is the labeling function, which now maps from transitions in the system to annotated HTTP requests.

7.2.3 Step 3 - Run-time monitoring

Step 3 takes as input (M, L) and, if previously specified, an authorization policy TA specifying which users can execute which tasks. The authorization policy is used to populate a database queried by M, resulting in a concrete monitor.

A reverse proxy intercepts all incoming requests to the application and decides, for each request, whether it is part of a workflow or not. To do so, it tries to match the URL and parameters in the request to annotated URLs and parameters stored in L, taking into account the constant, ignored and don't care values. If there is no match, the proxy forwards the request to the application, as it is not part of any workflow. If there is a match, the proxy associates the request to a task t of a workflow $W(T, U)$ and checks the annotated URL for `<<IID>>` and `<<UID>>` values, extracting the instance i and the user u. The user identifier is a cookie value that must be mapped to a user name in the policy. This is done by capturing login actions, storing the cookies issued to each user, and later retrieving the user names based on the cookie.

To enforce data invariants, when the proxy receives a request for the first URL containing the annotation `inv=<<I_i>>`, it stores the value of the parameter `inv` as v_i. When any subsequent task containing `<<I_i>>` is accessed, the value of the incoming annotated parameter `inc` is compared to the stored value ($v_i = \texttt{inc}$). In Example 1, for instance, $t1$ sets the value of parameter `value`, while $t3$ checks that this value is unchanged. In these requests, the monitor query $can\_do(u, t) \leftarrow \beta$ is dynamically conjoined with the data invariant condition, becoming $can\_do(u,t) \leftarrow \beta \wedge v_i = \texttt{inc}$. The WSP solution remains unchanged because the monitor still guarantees termination if the invariants are respected.

Finally, the proxy issues a request $can\_do(u,t)$ to the monitor of instance i of W and acts based on its response by either forwarding the request or dropping it.

7.3 Experiments

Aegis was implemented in Python 2.7. We capture execution traces as Zest[11] scripts exported by OWASP ZAP, extract data properties from them, aggregate them into an XES [144] file and use ProM[12] to output a PNML [76] file containing the mined Petri net. The implementation of the monitor synthesis algorithm is the one from Chapter 3, which takes as input a PNML file and outputs the synthesized SQL monitor. We use mitmproxy[13] as the reverse proxy and instantiate it with the generated monitor as in the example below:

```
mitmdump -R http://localhost:80 -p8080 -s httpmonitor.py
```

where `mitmdump -R` starts the proxy in the reverse mode to accept requests on port `8080`, process them using the `httpmonitor.py` script and, possibly, forward them to the application (`http://localhost:80`). The `httpmonitor.py` script intercepts requests and responses using a proxy API, performs the URL matching, queries a MySQL database (where the authorization policies are stored) by using the synthesized queries, and either forwards or drops the request. The proxy also supports HTTPS connections.

[11] `https://goo.gl/jNyFK4`

[12] `http://www.promwebtools.org/`

[13] `https://mitmproxy.org/`

7.3.1 Experimental setup

We tested Aegis on popular open-source applications (shown in Table 7.1), synthesizing monitors in the configurations $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ and $\langle \mathcal{I} \rangle$. Applications 1-4 are ERP platforms, 5-6 are e-health applications and 7-10 are e-commerce applications. Column 'Application' contains the name of each application; 'Language' shows the language in which it was developed; 'Params' describes the predominant method used for parameter passing (although an application can use several methods) and 'Downloads' reports the number of downloads (applications 1-6) or public installations (applications 7-10).

The different languages show the versatility of the black-box approach, which has to be tailored to support each parameter passing method (to annotate and match URLs). Supporting new applications that use the same method is straightforward, whereas supporting new methods (e.g., OData [121]) requires new functionality for model inference. The number of downloads and installations is a measure of the popularity of the applications and it comes from official repositories (applications 2, 3, 5, and 6), data in the web page of the project (applications 1 and 4), or related work [119] (applications 7-10). The number of actual deployments for applications 1-6 is not available as they are usually internal to an organization and not indexed by search engines. The numbers shown for applications 7-10 were obtained using Google dorks and are from 2014 [119].

We installed and pre-configured the applications using demo data (e.g., financial accounts for ERP, products for e-commerce, patients for e-health) either available during installation or generated by us, which required some manual effort. We then captured four execution traces for each workflow (as in [161]) and two login traces for each application.

To compare Aegis in different ERP applications, we used workflows offered by all of them: *Purchase order* (PO), *Sales order* (SO), *Purchase invoice* (PI), and *Sales invoice* (SI). They are slightly different in each application, varying from 4 to 6 tasks, usually with a gateway defining 2 to 3 alternative execution branches. Figure 7.4 shows at the top the patient visit workflow mined from OpenEMR (where the lines labeled by = represent BoD constraints added by us). The same Figure shows at the bottom the lab analysis workflow mined from BikaLIMS. In these 6 applications, we added the authorization constraints and specified policies with 10 users assigned to each task, generating $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$ monitors. The number of users was arbitrarily chosen because it influences the time taken to answer queries (discussed in Section 7.3.2).

The workflows for e-commerce applications are similar to the one shown in Figure 7.3. For these applications, we use the $\langle \mathcal{I} \rangle$ configuration, thus neither constraints nor authorization policies were defined. Applications 7 and 8 have a vulnerability allowing attackers to shop for free due to improper validation of PayPal Express Checkout tokens, which can be replayed from previous transactions, as explained in Section 7.1.2 (CVE-

Table 7.1: Applications used in the experiments

#	Application	Language	Params	Downloads
1	Odoo	Python	JSON	2M
2	Dolibarr	PHP	GET	850k
3	WebERP	PHP	GET	617k
4	ERPNext	Python	JSON	25k
5	OpenEMR	PHP	GET	382k
6	BikaLIMS	Python	REST	111k
7	OpenCart 1.5.3.1	PHP	GET	9M
8	TomatoCart 1.1.7	PHP	GET	119k
9	osCommerce 2.3.1	PHP	GET	80k
10	AbanteCart 1.0.4	PHP	GET	21k

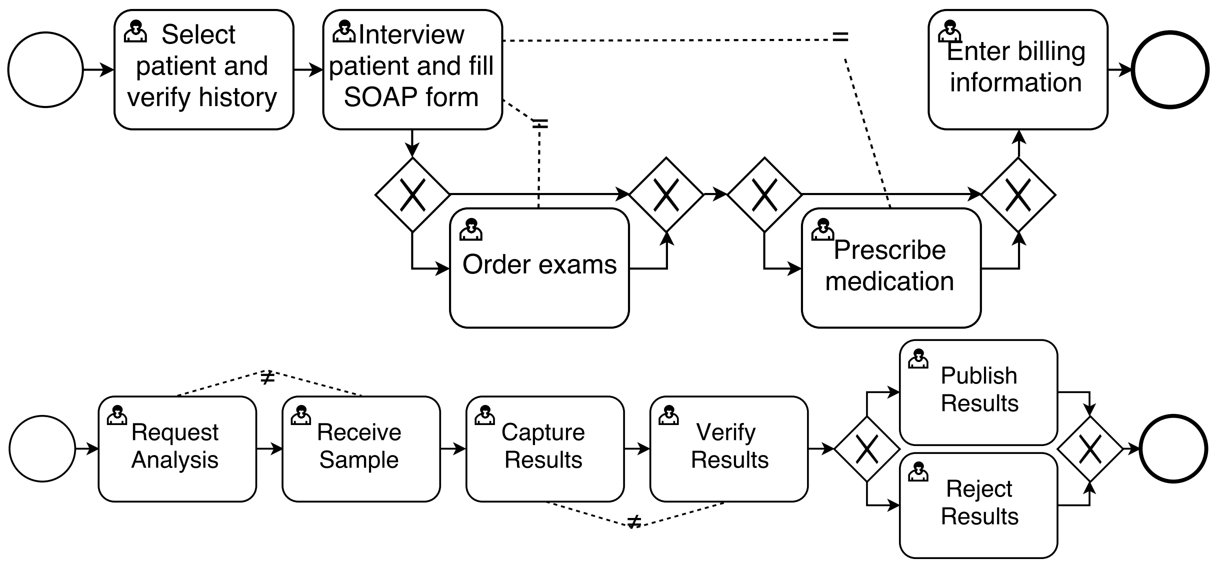


Figure 7.4: Patient visit workflow mined from OpenEMR (top) and lab analysis workflow mined from BikaLIMS (bottom)

2012-4934 for TomatoCart). Applications 9 and 10 allow an attacker to buy products and pay to himself, by tampering with a parameter that indicates who should receive the payment for a PayPal Payments Standard transaction (CVE-2012-2991 for osCommerce).

All applications were deployed as Docker [107] containers and the tests as Selenium scripts, using the architecture described in [44], which allows us to achieve repeatable experiments by automatically testing the applications in five steps: (i) start a new container with the application; (ii) run the workflow in the Selenium script without monitoring; (iii) start the monitor; (iv) run the workflow with monitoring; (v) capture results and destroy the container. The experiments ran on a MacBook Air 2014 laptop with a 1.3GHz dual-core Intel Core i5 processor and 8GB of RAM.

7.3.2 Results

The enforcement of security properties and mitigation of vulnerabilities was successful in all applications, which was confirmed by manual inspection.

In applications 1-6, we tested the enforcement of policies and constraints by trying the attacks described in Section 7.2 (workflow bypass, workflow violations, and authorization violations). The monitor was able to block situations such as the same user executing an entire workflow (SoD violation), and users trying to access tasks that were not assigned to them. In applications 7-10, we tried to exploit the vulnerabilities described above. In applications 7-8, the attacks were unsuccessful because `token` was detected as an invariant and automatically enforced. In applications 9-10, the `PayeeId` parameter was detected as a constant, since every trace in the input was related to the same shop. Constant values are usually not enforced, only used to match URLs (details in Section 7.2). For applications 9-10, we edited the inferred model by annotating `PayeeId` with `<<I>>`, so that requests with any value of `PayeeId` are controlled by the monitor, and used invariant enforcement with a constant, instead of with the first received value, to check that in every request containing `PayeeId`, its value is equal to *ShopId* (the constant obtained in the traces). Manual editing could be avoided by doing inference from a dataset containing execution traces of different shops.

We measured the overhead of the monitors by comparing the execution of each workflow with and without monitoring. Each execution was repeated 10 times and Table 7.2 shows the results (all times are in milliseconds). Column 'Application' shows the application under test (and the specific workflow tested for ERP applications); 'Original' reports the median time between receiving a request and sending a response with no monitor (measured by mitmproxy without the `httpmonitor.py` script); 'Query' reports the median time taken by the monitor to answer to a query (ignoring the time taken by the proxy to invoke the script, translate an incoming request to a monitor query, forward the request, etc); 'Aegis' reports the median time of a response with the monitor script (the time taken by the application, plus the translation time, plus the querying time); and 'Overhead' shows the overhead incurred by the use of the monitor as seen by a user (the difference between 'Aegis' and 'Original').

The time in column 'Query' varies with the size of a workflow and

Table 7.2: Monitoring overhead

#	Application	Original	Query	Aegis	Overhead
1	Odoo				
1.1	PO	112 ms	6 ms	132 ms	20 ms (17.8%)
1.2	SO	170 ms	7 ms	213 ms	43 ms (25.2%)
1.3	PI	174 ms	7 ms	213 ms	39 ms (22.4%)
1.4	SI	104 ms	7 ms	116 ms	12 ms (11.5%)
2	Dolibarr				
2.1	PO	93 ms	5 ms	103 ms	10 ms (10.7%)
2.2	SO	92 ms	4 ms	104 ms	12 ms (13%)
2.3	PI	89 ms	5 ms	97 ms	8 ms (8.9%)
2.4	SI	90 ms	5 ms	105 ms	15 ms (16.6%)
3	WebERP				
3.1	PO	51 ms	6 ms	59 ms	8 ms (15.6%)
3.2	SO	50 ms	5 ms	57 ms	7 ms (14%)
3.3	PI	30 ms	6 ms	37 ms	7 ms (23.3%)
3.4	SI	32 ms	4 ms	39 ms	7 ms (21.8%)
4	ERPNext				
4.1	PO	222 ms	7 ms	251 ms	29 ms (13%)
4.2	SO	327 ms	14 ms	411 ms	84 ms (25.8%)
4.3	PI	263 ms	10 ms	327 ms	64 ms (24.3%)
4.4	SI	272 ms	13 ms	318 ms	46 ms (16.9%)
5	OpenEMR	95 ms	7 ms	112 ms	17 ms (17.8%)
6	BikaLIMS	306 ms	7 ms	326 ms	20 ms (6.5%)
7	OpenCart	65 ms	6 ms	77 ms	12 ms (18.4%)
8	TomatoCart	63 ms	4 ms	71 ms	8 ms (12%)
9	osCommerce	79 ms	7 ms	95 ms	16 ms (20.2%)
10	AbanteCart	117 ms	8 ms	127 ms	10 ms (6.8%)

the number of users and constraints, as reported in Section 6.3.2, which describes a linear growth due to the *LogSpace* complexity of the queries used. The time in column ‘Aegis’ adds, to the time in *Query*, the time to process and match URLs, which depends on the data structures used.

As shown in Table 7.2, the overhead varied from 8ms to 84ms, with a median of 13.5ms, out of which less than 10ms in most cases is spent in querying the monitor. The overhead variability is due to the complexity of the workflows and the time taken to translate a request to a monitor. For instance, applications 1 and 4 have a large overhead because of the time to flatten JSON requests. Monitor synthesis is computationally much more expensive, but it is run off-line and only once for each workflow (unless there is a change in the application). Synthesis times are similar to what was reported in Chapter 6. For the workflows used in the experiments (3 to 6 tasks and 0 to 2 constraints), monitor synthesis takes less than 1 minute.

We did not test the performance of monitoring concurrent executions of workflows. Since there is no interaction between instances, we believe that any additional overhead would be related to request processing in the proxy and database access to answer monitor queries.

7.4 Discussion and limitations

Aegis targets workflow-driven web applications. These applications are common in domains such as ERP, e-health, and e-commerce, where control- and data-flow, and optionally authorization constraints and policies, need to be enforced.

There are solutions to design web applications using workflow modeling and frameworks that allow their declarative description (e.g., Spring Web Flow [151]). However, model-driven development of web applications is not common practice. This highlights the need for model inference, which can be based on static analysis [24, 72] or dynamic analysis [156, 73]. Using dynamic analysis by relying on an existing process mining tool allows us to develop a black-box approach. However, we believe that a hybrid solution, combining static and dynamic techniques, enriched with the knowledge of the user, is a direction to be investigated.

Our approach provides an easy way to enforce authorization policies and constraints over the actions a user can trigger when interacting with a service (via URL requests). Obtaining the same behavior within an application is not trivial. In fact, it must be done differently for each application and the granularity of the permissions therein offered may not be easily related to URLs requests. As an example, the granularity of permissions in the applications we experimented with varied from actions in a module (e.g., creating or approving an invoice in the finance module of Dolibarr) to binary module access. Though the former case provides permissions over actions, it may be the case that not all actions triggered in the process are covered, e.g., *Send by email* of Figure 7.2. In the latter case, it would not even be possible to create SoD constraints between actions triggered by different URLs implemented within the same module, since users that can access a module can perform all actions therein. None of the applications we analyzed offered support for authorization based on individual URLs nor authorization constraints (although ERPNext allows the definition of approval steps in predefined workflows). Moreover, the policy enforced by AEGIS can be applied on top of the existing one (if any) and can be easily specified by connecting tasks (obtained from the HTTP traces) to users (obtained from the application).

Limitations

AEGIS only sees the traffic between users and the target application, ignoring messages between a third party and a user, and between the application and a third party. The model inference step ignores some formats that can be used to exchange messages, such as XML in the body of a request or response. The first limitation is architectural; the second is an implementation issue. It is possible to have a parser for each format that returns (k, v) pairs and searches for them in incoming requests. The invariants that AEGIS detects and enforces are only exact matches, which is the most common case in web applications [161]. Adding support for more complex relations between data values (e.g., values in a list) requires new analysis and new data annotations.

These limitations, coupled with the complexity of model inference, may cause inferred models to be imprecise and not match all executions of an application, leading to false positives. Therefore, the user may not want AEGIS to block incoming requests, which could prevent legal executions. Instead, it can be used for soft enforcement, where denied requests represent deviations that must be logged so that a human agent can later examine them.

AEGIS synthesizes monitors that work in isolation, disregarding any possible inter-workflow and inter-instance dependencies and constraints. Related works consider such constraints when executing applications in several tabs [19].

Part III

Discussion

Chapter 8

Industrial impact

The work in this thesis is partly motivated by industrial applications because of the collaboration with SAP in the context of the SECENTIS project. In this Chapter, we discuss the industrial impact of our work by describing possible use cases of CERBERUS and AEGIS in an industrial setting.

8.1 Use cases

8.1.1 Cerberus

Business processes and workflow management systems are widespread in industry [158] and the commercial availability and use of Governance, Risk, and Compliance (GRC) software to enforce access control and compliance, highlight the need for supporting authorization policies and constraints for industrial workflow systems.

GRC systems (e.g., SAP GRC[1]) combine access control enforcement and auditing for business processes. They are often implemented as standalone tools, i.e. support for the specification and enforcement of authorization constraints is not integrated into workflow management systems.

[1] `http://go.sap.com/solution/platform-technology/analytics/grc.html`

Despite the importance of solutions to the WSP in industrial scenarios to reconcile business compliance and business continuity, to the best of our knowledge there are no commercial tools that support solutions to this problem, or related problems, e.g., resiliency. On the other hand, most academic approaches to solve the WSP are not viable as industrial applications because of their heavy computational cost at run-time. By splitting the problem in two phases, off-line and on-line, Cerberus is able to provide a solution that not only is efficient at run-time, but also supports the reuse of models and can be easily integrated in existing WFMS.

Cerberus creates a potential new commercial opportunity for existing BPM solutions by complementing them with an automated technique to ensure that authorization policies and constraints are enforced at run-time while ensuring business continuity. We validated our prototype (described in Chapter 6) on a real use case involving payment approval workflows provided by a business unit in SAP.

8.1.2 Aegis

Industrial tools for web application security are mostly focused on the development and testing of applications, using secure coding guidelines, Static Application Security Testing (SAST), and Dynamic Application Security Testing (DAST) [60]. Some properties enforced by Aegis can be achieved with other tools, e.g., Web Application Firewalls (WAFs) [13], which filter incoming requests based on user-defined rules. Although the use of WAFs is a best practice, they are typically incapable of handling logical flaws [49]. Indeed, most web application security tools protect against injection vulnerabilities rather than business logic flaws [45] and a defense-in-depth approach, combining multiple tools, is recommended [4]. We are unaware of any single tool that encompasses all the properties enforced by Aegis.

As discussed in Chapter 7, there is a trend of exposing business processes to customers as web applications, which leads to the need for tools that can enforce, at run-time, security properties related to workflow execution. AEGIS can be used as a complement to the tools described in the previous paragraph, providing transparent policy enforcement based on authorization, control-flow, and data-flow integrity and mitigating classes of zero-day logic vulnerabilities before a patch is made available.

AEGIS has not yet been validated in an industrial application, but we envision that it could be applied in three scenarios:

- New or existing workflow-driven applications running on, e.g., SAP HANA Extended Application Services[2], SAP's application server, which supports languages such as Java and server-side JavaScript. These applications may need to enforce any combination of the three properties $\langle \mathcal{P}, \mathcal{C}, \mathcal{I} \rangle$.
- ERP web applications such as S/4 HANA[3] that already offer support for authorization policies could be complemented with authorization constraint specification and enforcement ($\langle \mathcal{C}, \mathcal{I} \rangle$).
- E-commerce applications developed with Hybris[4], SAP's framework that integrates e-commerce websites with other business processes running on SAP systems. These applications typically need to implement control- and data-flow integrity ($\langle \mathcal{I} \rangle$).

In all scenarios, the basic use is the same: capture execution traces, select properties to enforce, generate a monitor, and deploy it (as described in the examples of Sections 7.1.1 and 7.1.2).

[2]`http://scn.sap.com/docs/DOC-60322`

[3] `http://go.sap.com/product/enterprise-management/s4hana-erp.html`

[4]`https://www.hybris.com/`

8.1.3 TestREx

As part of the development and testing of Aegis, we created another tool called TestREx [44], a testbed for repeatable exploits[5]. We did not detail the usage of TestREx in this thesis because it was out of scope. The tool has as main features: packing and running applications in different environments, using Docker containers; injecting exploits developed as Selenium scripts and monitoring their success; and generating security reports.

TestREx assumes that a security researcher has an exploit X that can subvert the execution of an application A running on an environment E, composed of operating system, database, and other supporting applications. The tool helps this researcher to answer (semi) automatically to the following questions:

1. Is X successful on application A running on a new environment E'?
2. Is X successful on a new version of the application, A', running on environment E?
3. Is X successful on a new version of the application, A', running on a new environment E'?

Figure 8.1 shows an overview of the tool. The main component is the *Execution Engine*, which takes as input a *Configuration* and an *Exploit* and outputs a *Report*. An *Application* is a set of files containing the source code of the system under test. A *Container* is the representation of the execution environment. It is an image of the system on which the application must be run, containing an operating system and supporting applications, like an application server and a database management system. A *Configuration* is a set of files used to bind an *Application* to a *Container*. It describes the setup required for a given application to run on a given container, like pre-

[5]TestREx is the result of a joint work with Stanislav Dashevskyi, another student in the SECENTIS project, and his advisors Fabio Massacci and Antonino Sabetta.

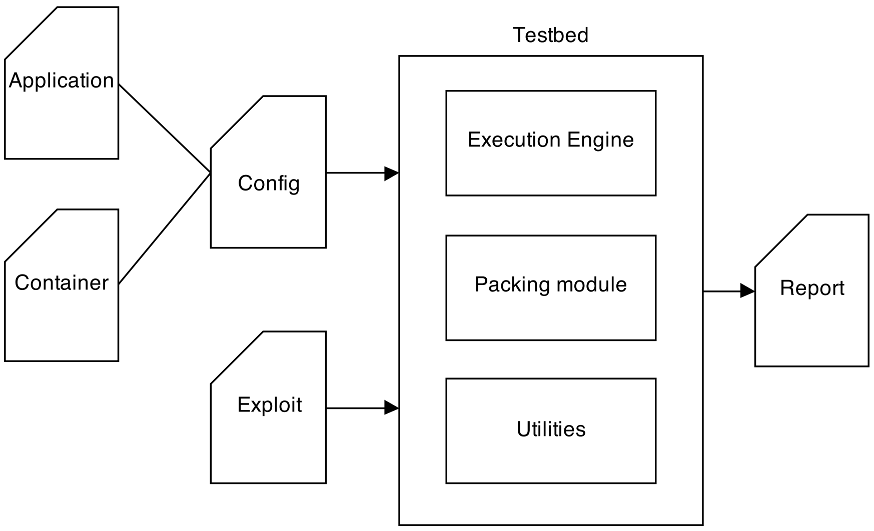


Figure 8.1: Overview of the testbed architecture

loading a database, creating users and starting a server. An *Exploit* is a sequence of steps that must be executed to take advantage of a vulnerability in the application. A *Report* contains a result of the execution of an exploit on a configuration and logs of the execution to provide more details. The *Packing Module* allows testers to package the applications and execution environments in a compressed archive file that can be deployed in another system running the testbed. The *Utilities* are a collection of scripts to import applications and exploits from other sources and to manage the containers.

There are several possible uses of TestREx in an industrial setting, covering different phases of the software development lifecycle. It can be used for automated software validation and regression testing, using a corpus of exploits stored in a corporate-wide repository to check the absence of known vulnerabilities or to perform regression tests to verify that a previously fixed vulnerability is not introduced again. TestREx exploit scripts can also be thought of as executable descriptions of an attack, replacing the current practice of using a combination of natural language and script-ing to describe the process and the configuration necessary to reproduce an attack.

8.2 Discussion and Perspectives

We presented some use cases of the tools developed in this thesis as industrial applications. None of the tools has yet reached commercial production, they are all research prototypes. Despite that, we believe that our experience with the process of technology transfer was already successful for the following reasons.

- First, two of the developed tools, TestREx and Cerberus, have been submitted as patent applications to the USPTO. The third tool, Aegis is under evaluation to follow the same steps. These patent applications show that there is some interest in the commercial usage of the techniques described in this thesis.

- Second, we were able to communicate research problems to an audience of software developers and business people. Our goal was to make them understand the importance of these problems. Even if solutions adopted in the future are not exactly the ones described in this work, we are confident that these discussions contribute to raise awareness about security issues and help making software, especially workflow applications, more secure. Although these discussions did not impact the techniques and implementations of our tools, we learned about real scenarios and requirements from customers.

- Third, we learned valuable lessons about technology transfer on the way. There is a big gap between early research prototypes and commercial products; bridging the gap involves convincing people to spend time and resources to turn a prototype into something usable and useful to customers. To convince the people who can invest the time and resources, there must a compelling use case. Just because a research problem is hard or interesting, it does not mean that the solution will have immediate impact on industrial practice.

Over the last decades there has been a growing interest in BPM [149, 145]. There are nowadays many well-known commercial tools to model, enact, and improve business processes, such as SAP NetWeaver[6], Oracle BPM Suite[7], and IBM BPM[8]. Process mining has been receiving a lot of attention and is being touted as a bridge between process management and data science [146]. There are also commercial process mining tools available, e.g., Disco[9] and Celonis[10]. A recent report on the state of the BPM market in 2015 can be found in [75]. As discussed in Chapter 1, the BPM field is experiencing a shift towards cloud-based, API-enabled solutions using web technologies, as evidenced by tools such as Bonita BPM[11]—used by, e.g., Sony, Cisco, and Orange—and Camunda[12]—used by, e.g., Allianz, Lufthansa, and T-Mobile.

Security is increasingly important in IT as a whole and is considered a key challenge in workflow-based applications [92]. It becomes even more critical if we consider the transition to web-based systems, which are notoriously lacking in terms of security [45]. The use of formal methods in many computer science disciplines, e.g., hardware verification and software testing, has improved the quality and reliability of computer systems. The participants of a recent NSF workshop on formal methods for security [27] concluded that

[6]`https://help.sap.com/netweaver`
[7]`http://www.oracle.com/us/technologies/bpm/overview/index.html`
[8]`https://www-03.ibm.com/software/products/en/business-process-manager-family`
[9]`https://fluxicon.com/disco/`
[10]`http://www.celonis.com/`
[11]`http://www.bonitasoft.com/`
[12]`https://camunda.com/`

> "formal methods can provide similar improvement in the security of computer systems.
>
> Moreover, formal methods are in a period of rapid development and significantly broadening practical applications. (...)
>
> Without broad use of formal methods, security will always remain fragile. Attackers have a clear advantage in what is currently a match between the cleverness of the attacker and the vigilance of the defender."

The same researchers anticipated that

> "Formal methods for security will have an enormous effect in the coming years. Recent advances now enable their use at scales that were previously impossible. The resulting security improvements will spur new investments in formal tools and techniques. This interplay will produce a virtuous circle of capital investments in the methods and increases in both the quality of secure systems and the productivity of security-minded developers."

We strongly support the statements above and believe that the techniques described in this thesis can pave the way towards formal tools to enforce security in workflow-based applications that are at the same time efficient and easy to use. Formal method techniques based on Satisfiability Modulo Theories (SMT) have shown their efficiency and scalability in many industrial settings. Some of them have been integrated into: dynamic symbolic execution (such as SAGE and Pex [70]), symbolic model checking (such as SLAM [8]), static analysis (such as PREfix [23]) and program verification (such as Boogie [46]). We believe that our monitor synthesis technique, being based on SMT model checking, can benefit from the power and flexibility of SMT solvers. Nevertheless, in the same report, it is written that

> "There are many open and compelling research problems, including: (...) What is required to enable formal methods for security at industrial scales and make them compatible with common industry processes?"

In our view, the efficiency and usability mentioned above are key issues for the adoption of such tools. However, as discussed at the beginning of this section, there is still a long way to go before the research prototypes described so far can become commercial products available to customers.

Chapter 9

Conclusions and Future work

In this thesis, we have motivated, described, implemented, and validated techniques and tools to solve the Workflow Satisfiability Problem and related problems, such as Scenario Finding and Workflow Resiliency, in workflow management systems and workflow-driven web applications. Solutions to these problems help users to avoid situations where a choice has to be made between business compliance and business continuity.

After introducing the work, in Chapter 1, and the state of the art, in Chapter 2, we have described, in Chapter 3, a precise technique to automatically synthesize run-time monitors capable of ensuring the successful termination of workflows while enforcing authorization policies and constraints, thus solving the run-time version of the WSP. It consists of an off-line phase in which we compute a symbolic representation of all possible behaviors of a workflow and an on-line phase in which the monitor is derived from such a symbolic representation. An advantage of the technique is that changes in the policies can be taken into account without re-running the off-line phase since only an abstract interface to policies is required. The interface is refined to the concrete policy only in the on-line phase. We have also described the assumptions for the correctness of the technique (cf. Theorem 4 in Chapter 3).

This technique was extended, in Chapter 4, to modular workflow specifications. This extension is crucial for two applications. First, to improve the scalability of monitor synthesis by adopting a divide-and-conquer approach where workflows are decomposed into smaller components for which monitors can be generated and later combined (cf. Theorem 6 in Chapter 4). Second, to enable the reuse of workflow models stored in repositories that can be retrieved and combined with new models, specifying control-flow and authorization constraints spanning multiple models and avoiding the burden of running monitor synthesis from scratch.

In Chapter 5, we have introduced four Scenario Finding Problems, discussed their relationships with the WSP, and argued that solving them supports the deployment of business processes and the activity of model reuse. We have also described algorithms to solve the SFPs, based on the monitor synthesis technique for the WSP.

In Chapter 6, we have described and implemented CERBERUS, a tool to integrate monitor synthesis, scenario finding, and run-time enforcement into workflow management systems. We have shown the use of the tool on a simple example and experimented with real-world workflows and synthetic benchmarks. The experiments show the scalability of monitor synthesis and run-time monitoring because of the modular approach to synthesis. The experiments also show that the scenario finding techniques can be used in practice at deployment time since they perform the computationally heaviest part (namely, computing the set of eligible scenarios) once and for all when the workflow is added to a repository and reuse it for any possible authorization policy.

In Chapter 7, we have described, implemented, and evaluated AEGIS, a technique and tool to enforce authorization policies and constraints, control- and data-flow integrity and ensure the satisfiability of workflow-driven web applications. We have tested our implementation with relevant open-source applications. Our findings confirm the validity of the approach in enforcing the desired properties and mitigating related vulnerabilities. The performance results show an overhead incurred at run-time that is negligible in many cases and acceptable in all the others.

We have also discussed the industrial impact of this thesis in Chapter 8, describing possible use cases of the developed research prototypes in an industrial context.

Our contributions are significant because we provide techniques (and describe their implementation) that can be used to solve the WSP and related problems in realistic, industrially relevant, scenarios. Our work has many differences with respect to related approaches described in Chapter 2. Our solution to the WSP is innovative because it focuses on the run-time and because it is split in two phases, allowing us to reuse the heavy computation of the reachability graph and to accommodate different authorization policies. The modular workflow specification and monitor synthesis allows us to support users in reusing workflows and monitors from repositories. The scenario finding approach allows us to define and solve a series of problems that users may face when trying to deploy a workflow in their organization with a specific authorization policy. Cerberus is innovative for allowing easy and transparent monitor synthesis and run-time enforcement for end-users, ensuring that workflow instances can be executed until successful termination without damaging neither business compliance nor continuity. Finally, Aegis is capable of enforcing security policies composed of many properties related to control-flow, data-flow, and authorization under the same umbrella and with a black-box approach.

9.1 Future work

There are many possible future research directions based on the work done in this thesis, but we want to highlight four: the extension of monitor synthesis with more authorization constraints (Section 9.1.1), modularity and reuse of workflow patterns (Section 9.1.2), generalizations of scenario finding (Section 9.1.3), and improved web application monitoring and security testing (Section 9.1.4).

9.1.1 Monitor extensions

In this thesis, we described and presented a few examples of authorization constraints, e.g., SoD and BoD. These are the most common in practice, but they are relatively simple. Other, more complex, constraints could also be supported by the monitor synthesis technique. Examples of such complex constraints include data-based policies and instance-spanning constraints.

There is a long line of works discussing data authorizations (see, e.g., [88, 3]) and the interplay between data and processes (see, e.g., [109]). The main reasons for specifying constraints on data objects are increased expressiveness and simplified specification and enforcement, since the absence of data-based constraints can lead to an explosive growth in the number of task-based constraints. One example of the need for data authorization is managing conflicts of interests [113], e.g., when a user has access to the data of two competing organizations. To avoid this kind of conflict, in a Chinese Wall model [21], data is separated into sets representing classes of conflict and when a user has access to the data of one of the elements of the set, he/she cannot access the data of the other elements.

Instance-spanning constraints [91] restrict what users can do across several instances of the same workflow (inter-instance), across several instances of different workflows (inter-process), or across workflows in different organizations (inter-organization). The most usual case is inter-instance authorization constraints, which have been studied in, e.g., [157]. Since we adopt the approach of having one monitor for each instance, support for inter-instance constraints would require a global synchronization of the states of each monitor, possibly using a global execution history. A possibility would be to design a central entity to which selected parts of the state of each monitor are communicated so that it can take the right decision to avoid that some inter-instance constraint is violated. Indeed, each monitor should ask the decision of the central entity before taking a decision. Although the design of this central entity may be challenging, we could take inspiration from cache-coherence protocols (see, e.g., [47]).

9.1.2 Modularity and reuse of workflow patterns

Since workflows are built from basic control-flow patterns (see, e.g., [148, 93]), a corollary of Theorem 6 in Chapter 4 is the possibility to compute reachability graphs once for each basic security-sensitive workflow component, store the result, and modularly combine it with others along the lines of Section 4.4. In the following, we elaborate a bit on this idea according to which a workflow is seen as a combination of basic components that can be expressed by the gluing operator $\oplus_G$ introduced in Section 4.3 (e.g., sequential, alternative and exclusive execution, as shown in Figure 9.1, where bidirectional arrows represent $\Leftrightarrow$ and dashed circles represent places of components that are not shown).

For the sake of simplicity, we consider the *sequential*, *parallel*, and *alternative* composition of just two security-sensitive workflow components (S_1, Int_1) and (S_2, Int_2). The generalization to n components is straightforward. We also assume that there is just one input and just one output place in both components (this is satisfied for the important class of workflow

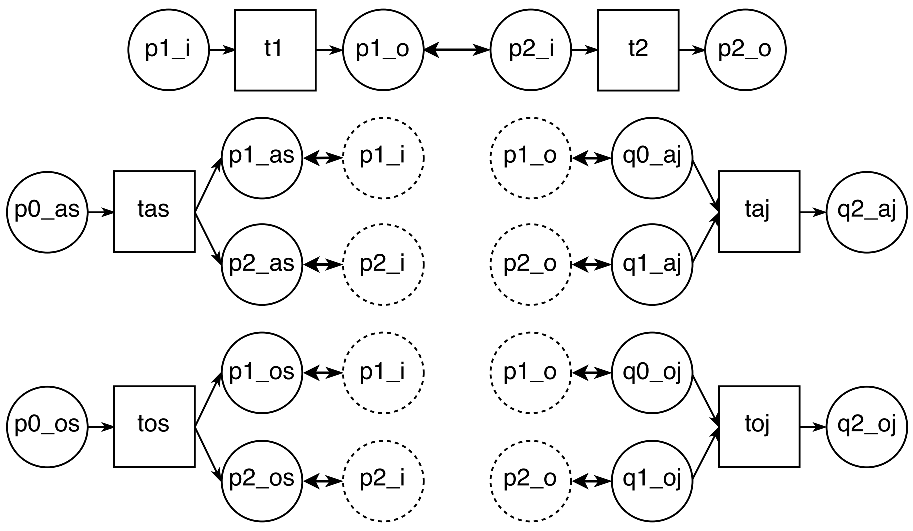


Figure 9.1: Workflow pattern composition. Sequential (top), parallel (middle), and alternative (bottom) composition.

nets [143]).

Sequential composition. It is sufficient to consider a set $G = G_{\mathrm{EC}} \cup G_{\mathrm{Auth}}$ of gluing assertions over Int_1 and Int_2 such that $G_{\mathrm{EC}} = \{p_2^i \Leftrightarrow p_1^o\}$. Notice that $(S_1, Int_1) \oplus_G (S_2, Int_2) = (S_2, Int_2) \oplus_G (S_1, Int_1)$ but because the gluing assertion in G_{EC} is $p_2^i \Leftrightarrow p_1^o$, and not $p_2^o \Leftrightarrow p_1^i$, the process specified by component (S_1, Int_1) will always be executed before that specified by (S_2, Int_2).

Parallel composition. We need to preliminarily introduce two other components, each containing a single transition, one for splitting (*a*nd *s*plit) and one for joining (*a*nd *j*oin) the execution flow. The transitions are as follows:

$$\begin{aligned} Tr_{as} &:= \{p0_{as} \wedge \neg d_{as} \to p0_{as}, p1_{as}, p2_{as}, d_{as} := F, T, T, T\} \\ Tr_{aj} &:= \{q0_{aj} \wedge q1_{aj} \wedge \neg d_{aj} \to q0_{aj}, q1_{aj}, q2_{aj}, d_{aj} := F, F, T, T\}. \end{aligned}$$

Then, it is sufficient to consider a set $G = G_{\mathrm{EC}} \cup G_{\mathrm{Auth}}$ of gluing assertions over Int_1, Int_2, Int_{as}, and Int_{aj} (recall that the gluing operator is associative) such that $G_{\mathrm{EC}} = \{p1_{as} \Leftrightarrow p_1^i, p2_{as} \Leftrightarrow p_2^i, p_1^o \Leftrightarrow q0_{aj}, p_2^o \Leftrightarrow q1_{aj}\}$.

Alternative composition. Similarly to parallel composition, we need to introduce two other components, each containing two non-deterministic transitions (*o*r *s*plit and *o*r *j*oin) to route the execution flow in one of the two components (S_1, Int_1) or (S_2, Int_2). The transitions are

$$\begin{aligned} Tr_{os} &:= \left\{ \begin{array}{l} p0_{os} \wedge \neg d_{os} \to p0_{os}, p1_{os}, p2_{os}, d_{os} := F, T, F, T, \\ p0_{os} \wedge \neg d_{os} \to p0_{os}, p1_{os}, p2_{os}, d_{os} := F, F, T, T \end{array} \right\} \\ Tr_{oj} &:= \left\{ \begin{array}{l} q0_{oj} \wedge \neg d_{oj} \to q0_{oj}, q2_{oj}, d_{oj} := F, T, T, \\ q1_{oj} \wedge \neg d_{oj} \to q1_{oj}, q2_{oj}, d_{oj} := F, T, T \end{array} \right\} \end{aligned}$$

Then, it is sufficient to consider a set $G = G_{\text{EC}} \cup G_{\text{Auth}}$ of gluing assertions over Int_1, Int_2, Int_{os}, and Int_{oj} such that $G_{\text{EC}} = \{p1_{os} \Leftrightarrow p^i_1, p2_{os} \Leftrightarrow p^i_2, p^o_1 \Leftrightarrow q0_{oj}, p^o_2 \Leftrightarrow q1_{oj}\}$.

We intend to add to Cerberus pre-computed monitors for the basic control-flow patterns, such as the ones shown above. Hence, a monitor for a new workflow could be synthesized while the user models it, by simply reading the pre-computed blocks from the repository and gluing them according to the constraints imposed by the user.

9.1.3 Scenario finding

A generalization and extension of the scenario finding problems discussed in Chapter 5 is to generate all possible meaningful (i.e., considering dominance or other redundancy notions) configurations of an authorization policy before the execution of a workflow and then check if there is at least one authorized scenario (similar to the problem of workflow feasibility [85]). Policy configurations can be created with the use of authorization delegation [43] or administrative rules (e.g., Administrative RBAC (ARBAC) [131]). An idea of how to solve this problem is to split it in two reachability problems: first generate the reachability graph for the workflow, then use the authorization part of the formulae in the leaves (i.e.

the initial tasks) as the goal of a reachability problem for ARBAC, which can be solved by similar symbolic model checking techniques, see, e.g., [6]. Naturally, this solution disregards changes to the authorization policy at run-time, i.e. after the workflow execution has started. Supporting changes at run-time is much more complex, since the naive solution of modeling administrative actions interleaved with workflow tasks generates a huge search space and may even not terminate, depending on the conditions imposed on the administrative rules.

Another future direction is how to automatically synthesize, or suggest changes to existing, authorization policies so that solutions of a scenario finding problem are optimal with respect to some criteria, e.g., least privilege [80] or cost/risk [11]. One idea is to associate weights with each edge in the reachability graph, considering all possible applications of the injective function of symbolic users to concrete users, and then to apply standard graph algorithms for shortest paths to search for optimal solutions. This quickly explodes as the number of users in the policy grows. It would be interesting to study situations in which the graph can be pruned to achieve usable solutions, even if they may be sub-optimal. An alternative is to use Bounded Model Checking [17] with weights so that it is possible to reuse SMT solvers capable of performing optimization, see, e.g., [137].

9.1.4 Web application monitoring and testing

The inference part of AEGIS can be greatly improved in two directions. First, remaining black-box, we could automatically filter the input traces to remove non-critical tasks, using state-changing detection algorithms, e.g., [55, 57] or detecting patterns in requests that must be dropped. Second, having access to the source-code, we could measure the coverage of the inferred model by executing the input traces and checking the paths executed in the code.

We would also like to explore monitor inlining [67], which requires source code changes to embed the monitor into the application. This can be done with the inferred model and a mapping from tasks to source-code entry points, or using model extraction tools from the source code, such as MARBLE [24]. In both cases, the synthesized monitor can be inlined into the application using techniques such as aspect-oriented programming [86] or policy-weaving [83].

We intend to adapt the model inference and monitor synthesis components of AEGIS to perform model-based testing of security policy enforcement [60] in web applications. The overall idea is to use the synthesized monitor as a test oracle and generate abstract test cases, which correspond to execution scenarios, by adapting tools such as MISTA [162] or M[agi]C [114], which combine model-based and combinatorial testing [87]. The abstract test cases can be later concretized into executable Selenium scripts (taking into account also the events that were discarded during the inference step) and the Selenium test cases can be run on TESTREX.

Bibliography

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

[2] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal Guards, Relativization of Quantifiers, and Failure Models in Model Checking Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:29–61, 2012.

[3] B. Alhaqbani, M. Adams, C.J. Fidge, and A.H.M. ter Hofstede. Privacy-aware workflow management. In *Proc. of BPM*. Springer, 2013.

[4] N. Antunes and M. Vieira. Defending against web application vulnerabilities. *Computer*, 45(2):66–72, Feb 2012.

[5] A. Armando and S.E. Ponta. Model checking of security-sensitive business processes. In *Proc. of FAST*. Springer, 2009.

[6] A. Armando and S. Ranise. Automated analysis of infinite state workflows with access control policies. In *Proc. of STM*. Springer, 2012.

[7] M. Balduzzi, C.T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proc. of NDSS*, 2011.

[8] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN*, pages 103–122, New York, NY, USA, 2001. Springer.

[9] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proc. of CCS*. ACM, 2007.

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

[11] D. Basin, S.J. Burri, and G. Karjoth. Optimal workflow-aware authorizations. In *Proc. of SACMAT*. ACM, 2012.

[12] D. Basin, S.J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security and business objectives. *JCS*, 22(5):661–698, 2014.

[13] M. Becher. *Web Application Firewalls*. VDM Verlag, Saarbrucken, Germany, 2007.

[14] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *TISSEC*, 2(1):65–104, February 1999.

[15] C. Bertolissi, D.R. dos Santos, and S. Ranise. Automated Synthesis of Run-time Monitors to Enforce Authorization Policies in Business Processes. In *Proc. of ASIACCS*. ACM, 2015.

[16] C. Bertolissi and S. Ranise. Verification of composed array-based systems with applications to security-aware workflows. In *Proc. of FROCOS*. Springer, 2013.

[17] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS*, pages 193–207, Berlin, 1999. Springer.

[18] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V.N. Venkatakrishnan. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proc. of CCS*. ACM, 2010.

[19] B. Braun, P. Gemein, H.P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *Proc. of ESSoS*, 2013.

[20] B. Braun, C. Gries, B. Petschkuhn, and J. Posegga. Ghostrail: Ad hoc control-flow integrity for web applications. In *Proc. of IFIP SEC*, 2014.

[21] D. Brewer and M.J. Nash. The chinese wall security policy. In *Proc. of S&P*. IEEE, 1989.

[22] A. Brucker and I. Hang. Secure and compliant implementation of business process-driven systems. In *Proc. of BPM*. Springer, 2013.

[23] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, June 2000.

[24] R.P. Castillo, I.G.R. de Guzman, and M. Piattini. Business process archeology using marble. *Inf. and Soft. Tech.*, 53(10):1023 – 1044, 2011.

[25] R.P. Castillo, J.A.C. Lemus, I.G.R. de Guzman, and M. Piattini. A family of case studies on business process mining using MARBLE. *J. of Systems and Soft.*, 85(6):1370 – 1385, 2012.

[26] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *TKDE*, 1(1):146–166, 1989.

[27] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich. Report on the NSF workshop on formal methods for security, 2016. Available at: `https://arxiv.org/abs/1608.00678`.

[28] N. Clark and D. Jolly. Société générale loses $7 billion in trading fraud. *The New York Times*, 2008. Available at: `http://www.nytimes.com/2008/01/24/business/worldbusiness/24iht-socgen.5.9486501.html`.

[29] D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Iterative plan construction for the workflow satisfiability problem. *Journal of Artificial Intelligence Research*, 51:555–577, 2014.

[30] D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Algorithms for the workflow satisfiability problem engineered for counting constraints. *Journal of Combinatorial Optimization*, 32(1):3–24, 2016.

[31] L. Compagna, D.R. dos Santos, S.E. Ponta, and S. Ranise. Cerberus: Automated Synthesis of Enforcement Mechanisms for Security-sensitive Business Processes. In *Proc. of TACAS*. Springer, 2016.

[32] L. Compagna, D.R. dos Santos, S.E. Ponta, and S. Ranise. Aegis: Automatic enforcement of security policies in workflow-driven web applications. In *Proc. of CODASPY*. ACM, 2017.

[33] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proc. of RAID*, 2007.

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

[35] J. Crampton, A. Gagarin, G. Gutin, M. Jones, and M. Wahlström. On the workflow satisfiability problem with class-independent constraints for hierarchical organizations. *ACM Trans. Priv. Secur.*, 19(3):8:1–8:29, October 2016.

[36] J. Crampton and G. Gutin. Constraint expressions and workflow satisfiability. In *Proc. of SACMAT*. ACM, 2013.

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

[38] J. Crampton, G. Gutin, D. Karapetyan, and R. Watrigant. The bi-objective workflow satisfiability problem and workflow resiliency. *CoRR*, abs/1512.07019, 2015.

[39] J. Crampton, G. Gutin, and R. Watrigant. Resiliency policies in access control revisited. In *Proc. of SACMAT*. ACM, 2016.

[40] J. Crampton, G. Gutin, and A. Yeo. On the parameterized complexity of the workflow satisfiability problem. In *Proc. of CCS*, pages 857–868. ACM, 2012.

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

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

[43] J. Crampton and H. Khambhammettu. Delegation and satisfiability in workflow systems. In *Proc. of SACMAT*. ACM, 2008.

[44] S. Dashevskyi, D.R. dos Santos, F. Massacci, and A. Sabetta. TESTREX: a Testbed for Repeatable Exploits. In *Proc. of CSET*. USENIX, 2014.

[45] G. Deepa and P.S. Thilagam. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Inf. and Soft. Tech.*, 74:160–180, 2016.

[46] R. DeLine and K.R.M. Leino. Boogiepl: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[47] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. of CAV*, pages 53–68. Springer, 2000.

[48] G. Demarty, F. Maronnaud, G. Le Breton, and S. Hallé. Sitehopper: Abstracting navigation state machines for the efficient verification of web applications. In *Proc. of WS-FM*, 2013.

[49] M. Dermann, M. Dziadzka, B. Hemkemeier, A. Hoffmann, A. Meisel, M. Rohr, and T. Schreiber. OWASP Best Practices: Use of Web Application Firewalls, 2008. Available at: `https://www.owasp.org/index.php/Best_Practices:_Web_Application_Firewalls`.

[50] R. Dijkman, M. La Rosa, and H.A. Reijers. Editorial: Managing large collections of business process models-current techniques and challenges. *Computers in Industry*, 63(2):91–97, 2012.

[51] R.M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Inf. and Soft. Tech.*, 50(12):1281 – 1294, 2008.

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

[53] D.R. dos Santos, S. Ranise, L. Compagna, and S.E. Ponta. Assisting the Deployment of Security-Sensitive Workflows by Finding Execution Scenarios. In *Proc. of DBSec*. Springer, 2015.

[54] D.R. dos Santos, S. Ranise, and S.E. Ponta. Modular Synthesis of Enforcement Mechanisms for the Workflow Satisfiability Problem: Scalability and Reusability. In *Proc. of SACMAT*. ACM, 2016.

[55] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proc. of USENIX Security*, 2012.

[56] R.G. Downey and M.R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[57] F. Duchene, S. Rawat, J. Richier, and R. Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *Proc. of WCRE*, 2013.

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

[59] J. Epstein. Security lessons learned from société générale. *IEEE Security and Privacy*, 6(3):80–82, May 2008.

[60] M. Felderer, M. Büchlein, M. Johns, A.D. Brucker, R. Breu, and A. Pretschner. Security testing: A survey. *Advances in Computers*, 101:1–51, March 2016.

[61] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proc. of USENIX Security*, 2010.

[62] P. Zhang G. Chartrand. *Chromatic Graph Theory*. 2008.

[63] A. Gambi and C. Pautasso. Restful business process management in the cloud. In *Proc. of PESOS*, 2013.

[64] P. Gaubatz, W. Hummer, U. Zdun, and M. Strembeck. Supporting customized views for enforcing access control constraints in real-time collaborative web applications. In *Proc. of ICWE*, 2013.

[65] P. Gaubatz, W. Hummer, U. Zdun, and M. Strembeck. Enforcing entailment constraints in offline editing scenarios for real-time collaborative web documents. In *Proc. of SAC*. ACM, 2014.

[66] P. Gaubatz and U. Zdun. Supporting entailment constraints in the context of collaborative web applications. In *Proc. of SAC*. ACM, 2013.

[67] G. Gheorghe and B. Crispo. A survey of runtime policy enforcement techniques and implementations. Technical report, University of Trento, 2011.

[68] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *LMCS*, 2010.

[69] S. Ghilardi and S. Ranise. Mcmt: A model checker modulo theories. In *Proc. of IJCAR*. Srpringer, 2010.

[70] P. Godefroid, P. de Halleux, A.V. Nori, S.K. Rajamani, W. Schulte, N. Tillmann, and M.Y. Levin. Automating software testing using program analysis. *IEEE Softw.*, 25(5):30–37, September 2008.

[71] N.Z. Haddar, L. Makni, and H.B. Abdallah. Literature review of reuse in business process modeling. *Software & Systems Modeling*, 13(3):975–989, 2014.

[72] W.G.J. Halfond. Identifying inter-component control-flow in web applications. In *Proc. of ICWE*, 2015.

[73] W.G.J. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proc. of ISSTA*. ACM, 2009.

[74] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proc. of ASE*. IEEE, 2010.

[75] P. Harmon. The state of business process management, 2016. Available at: `http://www.bptrends.com/bpt/wp-content/uploads/2015-BPT-Survey-Report.pdf`.

[76] L.M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves. A primer on the petri net markup language and iso/iec 15909-2. In *Proc. of CPN*, 2009.

[77] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[78] J. Holderer, R. Accorsi, and G. Müller. When four-eyes become too much: a survey on the interplay of authorization constraints and workflow resilience. In *Proc. of SAC*. ACM, 2015.

[79] V.C. Hu, D.R. Kuhn, and D.F. Ferraiolo. Attribute-based access control. *IEEE Computer*, 48(2):85–88, 2015.

[80] H. Huang, F. Shang, J. Liu, and H. Du. Handling least privilege problem and role mining in rbac. *Journal of Combinatorial Optimization*, pages 1–24, 2013.

[81] K. Jayaraman, G. Lewandowski, P.G. Talaga, and S.J. Chapin. Enforcing request integrity in web applications. In *Proc. of DBSec.* Springer, 2010.

[82] K. Jayaraman, P.G. Talaga, G. Lewandowski, S.J. Chapin, and M. Hafiz. Modeling user interactions for (fun and) profit: preventing request forgery attacks on web applications. In *Proc. of PLOP*, 2009.

[83] R. Joiner, T. Reps, S. Jha, M. Dhawan, and V. Ganapathy. Efficient runtime enforcement techniques for policy weaving. In *Proc. of FSE.* IEEE, 2014.

[84] J.-P. Katoen. Causal behaviours and nets. In *Proc. of APN*, pages 258–277. Springer, 1995.

[85] A.A. Khan and P.W.L. Fong. Satisfiability and feasibility in a relationship-based workflow authorization model. In *Proc. of ESORICS*, pages 109–126. Springer, 2012.

[86] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of aspectj. In *Proc. of ECOOP.* Springer, 2001.

[87] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, August 2009.

[88] V. Kunzle. *Object-Aware Process Management.* PhD thesis, University of Ulm, July 2013.

[89] G. Lawton. In 2016, b2b integration api strategy becomes a priority, 2016. Available at: `http://searchsoa.techtarget.com/news/4500279512/In-2016-B2B-integration-API-strategy-becomes-a-priority`.

[90] G. Lawton. New jboss bpm middleware aims to increase role of microservices, 2016. Available at: `http://searchsoa.techtarget.com/news/450299860/New-JBoss-BPM-middleware-aims-to-increase-role-of-microservices`

[91] M. Leitner, J. Mangler, and S. Rinderle-Ma. Definition and enactment of instance-spanning process constraints. In *Proc. of WISE*, pages 652–658. Springer, 2012.

[92] M. Leitner and S. Rinderle-Ma. A systematic review on security in process-aware information systems–constitution, challenges, and future directions. *Inf. and Soft. Tech.*, 56(3):273–293, 2014.

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

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

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

[96] N. Li, Q. Wang, and M. Tripunitara. Resiliency policies in access control. *TISSEC*, 12(4):20:1–20:34, April 2009.

[97] X. Li and Y. Xue. Block: a black-box approach for detection of state violation attacks towards web applications. In *Proc. of ACSAC*. ACM, 2011.

[98] X. Li, Y. Xue, and B. Malin. Detecting anomalous user behaviors in workflow-driven web applications. In *Proc. of SRDS*, 2012.

[99] D.R. Licata and S. Krishnamurthi. Verifying interactive web programs. In *Proc. of ASE*. IEEE, 2004.

[100] H. Lu, Y. Hong, Y. Yang, Y. Fang, and L. Duan. Dynamic workflow adjustment with security constraints. In *Proc. of DBSec*. Springer, 2014.

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

[102] J.C. Mace, C. Morisset, and A. van Moorsel. Modelling user availability in workflow resiliency analysis. In *Proc. of HotSoS*. ACM, 2015.

[103] J.C. Mace, C. Morisset, and A. van Moorsel. Proc. of qest. In *Impact of Policy Design on Workflow Resiliency Computation Time*. Springer, 2015.

[104] J.C. Mace, C. Morisset, and A. van Moorsel. Resiliency variance in workflows with choice. In *Proc. of SERENE*. Springer, 2015.

[105] J.C. Mace, C. Morisset, and A. van Moorsel. *WRAD: Tool Support for Workflow Resiliency Analysis and Design*, pages 79–87. Springer, 2016.

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

[107] K. Matthias and S.P. Kane. *Docker: Up & Running*. O'Reilly, 2015.

[108] M. Meucci and A. Muller. *Testing Guide 4.0*. The OWASP Foundation, 2015.

[109] A. Meyer, S. Smirnov, and M. Weske. Data in business processes. Technical report, University of Potsdam, 2011.

[110] M. Muehlen and J. Recker. How much language is enough? theoretical and practical use of the business process modeling notation. In *Proc. of CAISE*. Springer, 2008.

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

[112] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch. Flowwatcher: Defending against data disclosure vulnerabilities in web applications. In *Proc. of CCS*. ACM, 2015.

[113] N. Nassr and E. Steegmans. Mitigating conflicts of interest by authorization policies. In *Proc. of SIN*. ACM, 2015.

[114] C.D. Nguyen, A. Marchetto, and P. Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proc. of ISSTA*, 2012.

[115] OASIS. Web services business process execution language version 2.0. Technical report, OASIS, 2007. Available at: `https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[116] OMG. Business process model and notation (bpmn), version 2.0. Technical report, Object Management Group, 2011.

[117] OWASP. *OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks*. The OWASP Foundation, 2013.

[118] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *Proc. of FMICS*, pages 37–52. Springer, 2009.

[119] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *Proc. of NDSS*, 2014.

[120] S. Perera, I. Kumara, S. Weerawarana, and M. Pathirage. A multi-tenant architecture for business process executions. In *Proc. of ICWS*, 2011.

[121] M. Pizzo, R. Handl, and M. Zurmuehl. Odata version 4.0 part 1: Protocol. Technical report, OASIS, 2014. Available at: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata.

[122] N. Poggi, V. Muthusamy, D. Carrera, and Rania Khalaf. Business process mining from e-commerce web logs. In *Proc. of BPM*. Springer, 2013.

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

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

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

[126] A.W. Roscoe. A classical mind. chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[127] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Proc. of ER*, 2005.

[128] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow resource patterns: Identification, representation and tool support. In *Proc. of CAiSE*, 2005.

[129] M. Salnitri, F. Dalpiaz, and P. Giorgini. Modeling and verifying security policies in business processes. In *Proc. of BPMDS*. Springer, 2014.

[130] P. Samarati and S.C. de Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*. Springer, 2001.

[131] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *TISSEC*, 2(1):105–135, February 1999.

[132] R. Sandhu, E. Coyne, H. Feinstein, and C. Youmann. Role-based access control models. *IEEE Computer*, 2(29):38–47, 1996.

[133] S. Sankaranarayanan, H. Sipma, and Z. Manna. Petri net analysis using invariant generation. In *In Verification: Theory and Practice*. Springer, 2003.

[134] G.K. Schneider. Sap modeling handbook - modeling standards, 2010.

[135] S. Schulte, C. Janiesch, S. Venugopal, I. Weber, and P. Hoenisch. Elastic business process management: State of the art and open challenges for {BPM} in the cloud. *FGCS*, 46:36 – 50, 2015.

[136] M. Schur, A. Roth, and A. Zeller. Mining workflow models from web applications. *IEEE TSE*, 41(12):1184–1201, dec 2015.

[137] R. Sebastiani and P. Trentin. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Proc. of CAV*. Springer, 2015.

[138] A.U. Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[139] S. Son, K.S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *Proc. of NDSS*, 2013.

[140] A. Sudhodanan, A. Armando, L. Compagna, and R. Carbone. Attack patterns for black-box security testing of multi-party web applications. In *Proc. of NDSS*, 2016.

[141] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *Proc. of USENIX Security*, 2011.

[142] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8(2):129–165, March 2008.

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

[144] W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes.* Springer, 2011.

[145] W.M.P. van der Aalst. Business process management: A comprehensive survey. *ISRN Software Engineering*, 2013, 2013.

[146] W.M.P. van der Aalst. *Process Mining: Data Science in Action.* Springer, 2016.

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

[148] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.

[149] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. Business process management: A survey. In *Proc. of BPM*, 2003.

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

[151] E. Vervaet. *The Definitive Guide to Spring Web Flow.* Apress, Berkely, CA, USA, 2008.

[152] B. von Halle. *Business rules applied: building better systems using the business rules approach.* Wiley, 2001.

[153] M. Wang, K.Y. Bandara, and C. Pahl. Process as a service. In *Proc. of SCC*, 2010.

[154] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *TISSEC*, 13, 2010.

[155] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proc. of S&P*, 2012.

[156] W. Wang, Y. Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *Proc. of ICSM*, 2009.

[157] J. Warner and V. Atluri. Inter-instance authorization constraints for secure workflow management. In *Proc. of SACMAT*, pages 190–199. ACM, 2006.

[158] M. Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[159] G. Winskel. Petri nets, algebras, morphisms, and compositionality. *Inf. and Comp.*, 72(3):197 – 238, 1987.

[160] C. Wolter, A. Schaad, and C. Meinel. Task-based entailment constraints for basic workflow patterns. In *Proc. of SACMAT*. ACM, 2008.

[161] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *Proc. of NDSS*, 2013.

[162] D. Xu, M. Kent, L. Thomas, T. Mouelhi, and Y. Le Traon. Automated model-based testing of role-based access control using predicate-transition nets. *IEEE Transactions on Computers*, 64(9):2490–2505, 2015.

[163] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *Proc. of CCS*. ACM, 2013.

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