RZ 3784 (# Z1008-001) 08/16/2010
Computer Science 15 pages

Research Report

Separation of Duties as a Service

Samuel J. Burri*‡, Günter Karjoth*, David Basin‡

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

‡ETH Zurich
Department of Computer Science
Switzerland

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

Separation of Duties as a Service *

Samuel J. Burri*,†, Günter Karjoth*, and David Basin†

*IBM Research – Zurich, Switzerland
†ETH Zurich, Department of Computer Science, Switzerland

Abstract

We introduce the concept of Separation of Duties (SoD) as a Service, a new approach to enforce SoD requirements on workflows and thereby prevent fraud and errors. SoD as a Service facilitates a separation of concern between business experts and security professionals. Moreover, it allows enterprises to address the need for internal controls and to quickly adapt to organizational, regulatory, and technological changes, which are common characteristics of today's dynamic business environments. We describe our implementation of SoD as a Service, which extends a widespread, commercial workflow system. We validate our approach and implementation with a realistic case study, a drug dispensation workflow deployed in a hospital.

1 Introduction

New technologies and methodologies, such as Service-Oriented Architectures (SOAs), facilitate the integration of legacy information systems with new system components and the dynamic outsourcing of business functionality. This enables organizations to concentrate on mission-critical and value-generating business activities and to outsource less central activities. *Software as a Service (SaaS)* is a new software delivery model that is motivated by these technical advances and new business models [27]. SaaS decouples the ownership and the use of software by providing its functionality as a service and facilitates a demand-driven, late binding of system components. Along with this decomposition and distribution of work comes the need to structure and organize business tasks, which is typically done in the form of business processes, modeled as workflows.

The second trend that motivates our work is the increasing effort of organizations to enforce internal controls in order to fight fraud and to comply with regulatory requirements. For example, regulations such as the Sarbanes-Oxley Act [1] mandate companies to document their business processes, to identify fraud and security vulnerabilities, and to apply appropriate countermeasures. Most security requirements for business processes are concerned with human activities, as the most severe security risks stem from human interaction [10]. *Separation of Duties (SoD)* is a popular class of constraints on human activities that prevent a single user from executing all critical tasks in a workflow. Therefore, the collusion of at least two users is required to commit fraud. Various frameworks have been developed for specifying and analyzing authorization constraints for business processes. However, they are limited in the kinds of constraints they can handle and typically force a tight coupling of the workflow and constraint definition. The SoD algebra (SoDA) of Li and Wang [18] constitutes a notable exception. It allows one to model expressive SoD constraints decoupled from a workflow definition.

In this paper, we address both the trend towards loosely-coupled, service-oriented architectures and the increasing need for internal, process-oriented controls. A common characteristics of these two developments is change. SaaS is motivated by fast-changing business environments and internal controls must quickly adapt to changing regulations and threats. Past research has largely ignored the impact of changing authorizations on running business processes. A prominent example of the relevance of these dynamic aspects is the loss of

* The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement N° 216917.

4.9 billion euros that the French bank Société Générale suffered due to unauthorized trades of its employee Jérôme Kerviel who exploited the authorizations of his former job role, which were not revoked [8]. The European fraud survey of Ernest & Young confirms that organizational changes, triggered by acquisitions and job cuts, are among the major sources of fraud [10]. To counter such problems, Basin et. al. bridge the gap between workflow-independent SoD constraints, formalized as SoDA terms, and their enforcement in a general workflow model [5]. Their approach also accounts for changing authorizations and thereby generalizes the original SoDA semantics [18].

We report on the implementation and validation of the results of [5] in a SOA-based workflow environment. We illustrate how off-the-shelf, widespread software products can be combined and extended to improve internal control, while remaining flexible with respect to change. We compare the runtime complexity of our implementation to the runtime complexity of the underlying formal models and explain how we achieve an acceptable runtime performance in practice. Through an extensive and realistic case study, we test the applicability of our approach to a real-world scenario and we use a series of performance measurements to confirm the results of our complexity analysis.

Overall, our contributions are as follows. First, we empirical validate the applicability of the theoretical models in [5] to a realistic business setting. Second, we introduce the concept of *SoD as a Service*, which is an instance of SaaS, providing SoD enforcement as a service. SoD as a Service has a number of attractive properties. It enables a loose coupling between a workflow engine that executes the business logic, a user repository that administers users and their authorizations, and the enforcement of abstract SoD constraints. Loose coupling and the employment of the service concept in turn facilitates a separation of concerns. Business experts can focus on modeling business processes, managers on the organizational design, and security professionals on the enforcement of internal controls – each of them requiring minimal interaction with the other two. Our architecture is also well-suited for enforcing SoD constraints on legacy systems. By accepting a moderate increase of communication, our architecture allows a reduction of implementation costs and configuration efforts. At the same time, changing legal requirements or organizational changes can quickly be reflected in the IT infrastructure.

We proceed in Section 2 with a brief introduction to the formal prerequisites of this paper: CSP, workflows, authorizations, and SoDA. We describe the implementation of our prototype system in Section 3, including our design goals, a complexity analysis, and a discussion of the design decisions. We introduce a real-world scenario, the enforcement of SoD constraints on a drug dispensation workflow deployed in a hospital, together with our performance measurements in Section 4. After discussing related work in Section 5, we draw conclusions in Section 6. Supporting documentation is given in the appendix.

2 Background

Our work builds on the models of [5]. In particular, we also model workflow systems using the process algebra *CSP* [22]. In CSP, systems are described by communicating processes. An *event* is the smallest unit of activity; let Σ denote the set of all events. Events can be structured using *channels*. For a set A and a channel c, we say that c is of *type* A if $\{c.a \mid a \in A\} \subseteq \Sigma$. For a tuple $(a_1, a_2, \ldots a_n)$ we write $c.a_1.a_2.\ldots.a_n$. A sequence of events, written $\langle e_1, e_2, \ldots, e_n \rangle$, is called a *trace* and Σ^* is the set of all traces over Σ. For two traces i_1 and i_2, we denote their concatenation by $i_1 \hat{} \; i_2$.

A *process* describes a communication pattern. The denotational semantics of CSP defines the behavior of a process in terms of a set of traces. Formally, for a process P, $\mathsf{T}(P) \subseteq \Sigma^*$ is a prefix-closed set of traces, each describing a possible execution of P. For a trace i and an event e, if $i \in \mathsf{T}(P)$ we say that P *accepts* i and if $i \hat{} \langle e \rangle \in \mathsf{T}(P)$ we say that P *engages* in e after P accepted i. Processes can be parametrized. For a variable v, $P(v)$ describes a class of processes, where the behavior of an instance of $P(v)$ depends on the value of v. Finally, for two processes P and Q and a set of events E, $P \underset{E}{\parallel} Q$ is the parallel process that engages in an event $e \in E$ if both P and Q engage in e, and it engages in an event $e \notin E$ if P or Q engages in e.

2.1 Workflows

We call a unit of work a *task*. Because SoD constraints are defined with respect to human activities, we concentrate on tasks that are executed by humans, either directly or through the execution of a program on their behalf. A *workflow* models the temporal ordering and causal dependencies of a set of tasks that together implement a business objective.

The execution of a workflow by a *workflow engine* is a called a *workflow instance*. A workflow engine may execute multiple instances of the same workflow in parallel. The execution of a task in a workflow instance is called a *task instance*. Let $\mathcal{U}$ be a set of *users* and $\mathcal{T}$ a set of *tasks*. We use the channel b of type $\mathcal{T} \times \mathcal{U}$ to model the execution of tasks. For a task $t \in \mathcal{T}$, and a user $u \in \mathcal{U}$, we call the event $b.t.u$ a *business event*. It describes the execution of t by u, i.e. a task instance of t. Let Σ_B be the set of all business events.

In addition to business events, we use the event *done* to denote that a workflow has finished. Given a workflow w, we model w as a process W. Every trace $i \in \mathsf{T}(W)$ corresponds to a workflow instance of w. For a trace $i \in \Sigma^*$, the function $\mathsf{users}(i)$ returns the set of users contained in business events in i – that is, the function users extracts the users who executed a task in a workflow instance. For example, $\mathsf{users}(\langle b.t_1.\mathtt{Bob}, b.t_2.\mathtt{Claire}, b.t_3.\mathtt{Bob}, \mathit{done}\rangle) = \{\mathtt{Bob}, \mathtt{Claire}\}$.

2.2 Authorizations

We use *Role-based Access Control (RBAC)* [11] to model authorizations. Let a set of *roles* $\mathcal{R}$, a *user-assignment (relation)* $UA \subseteq \mathcal{U} \times \mathcal{R}$, and a *role-assignment (relation)* $PA \subseteq \mathcal{R} \times \mathcal{T}$ be given. We call a tuple (UA, PA) an *RBAC configuration*. A user u is *authorized* to execute a task t if there exists a role $r \in \mathcal{R}$ such that $(u, r) \in UA$ and $(r, t) \in PA$. We say that u *acts* in role r if $(u, r) \in UA$. We do not consider sessions but they could be modeled with the administrative commands introduced below. In an enterprise environment, users and their credentials, including their role assignments, are typically stored and administrated in a *user repository*.

For a user assignment UA and a permission assignment PA, the *RBAC process* $RBAC(UA, PA)$ models the enforcement of role-based authorizations and the administration of UA. The process $RBAC(UA, PA)$ engages in a business event $b.t.u$ if u is authorized to execute t with respect to UA and PA. Furthermore, we model the administration of user-assignment relations with a set of *admin events* Σ_A that are communicated over a channel a. For every user $u \in \mathcal{U}$ and every role $r \in \mathcal{R}$, Σ_A contains the admin events $a.\mathtt{rmUA}.u.r$ and $a.\mathtt{addUA}.u.r$. The RBAC process $RBAC(UA, PA)$ engages in $a.\mathtt{addUA}.u.r$ and behaves like $RBAC(UA \cup \{(u, r)\}, PA)$ afterward. Similarly, it engages in $a.\mathtt{rmUA}.u.r$ and behaves like $RBAC(UA \setminus \{(u, r)\}, PA)$ afterward. In other words, $a.\mathtt{addUA}.u.r$ adds the user assignment (u, r) to UA and $a.\mathtt{rmUA}.u.r$ removes it from UA.

2.3 Separation of Duty Algebra

Li and Wang's *separation of duty algebra (SoDA)* describes SoD constraints independent of workflows. This decouples workflow definitions and SoD enforcement and naturally fits our SoD as a Service approach. In this paper, we merely motivate SoDA by giving a few examples. See [18] for a complete language definition.

SoDA formalizes SoD constraints as terms. Let a term ϕ and a user assignment UA be given. A set of users U *satisfies* ϕ with respect to UA, written $U \vdash_{UA} \phi$, if the users in U and their role assignments contained in UA comply with the SoD constraint described by ϕ.

For example, consider the terms: $\phi_1 = \mathsf{All} \otimes \mathsf{All} \otimes \mathsf{All}$, $\phi_2 = \mathtt{Pharmacist} \sqcup (\mathtt{Nurse} \otimes \mathtt{Nurse})$, and $\phi_3 = (\mathtt{Therapist} \otimes \mathtt{Nurse}) \sqcap (\neg\mathtt{Patient} \sqcap \neg\{\mathtt{Bob}, \mathtt{Claire}\})^+$. The term ϕ_1 is satisfied by every set containing three users; i.e., ϕ_1 requires the separation of duties between three arbitrary users. The term ϕ_2 is satisfied by either a user acting as `Pharmacist` or two different users, both acting as `Nurse`. Under the assumption that a `Pharmacist` has more medical knowledge than a `Nurse`, this constraint could be used to ensure that the medical decisions of a `Nurse` are double-checked by another `Nurse` while a `Pharmacist`'s decision need not be checked by a second user. The term ϕ_3 requires a user acting as `Therapist` and another user acting as `Nurse`. In addition, both users must not act as `Patient` and be neither `Bob` nor `Claire`. These examples illustrate how SoDA can be used to express quantitative and qualitative restrictions. Terms define both the number of users and the kinds of users that are required for the execution of a set of tasks.

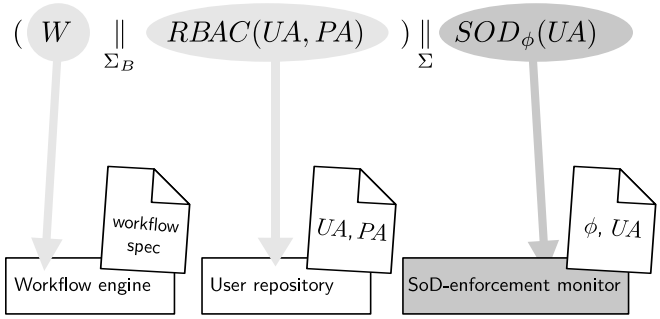

Figure 1: From theory to practice

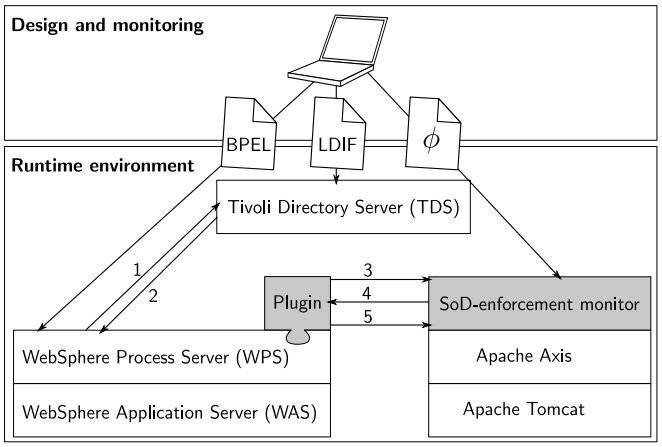

Figure 2: Architecture

Basin et. al. generalize the original SoDA semantics to a trace semantics that also accounts for changing authorizations [5]. Thereby, they close the gap between the workflow-independent, abstract specification of SoD constraints and their enforcement on workflows. Given a user assignment UA and a term ϕ, they describe the construction of a process $SOD_\phi(UA)$, called SoD-enforcement process, that engages in all business events that correspond to a satisfying set of users for ϕ with respect to UA. Additionally, $SOD_\phi(UA)$ also engages in admin events that modify its user assignment UA. The relation between the satisfaction of ϕ by a set of users and the acceptance of a trace by $SOD_\phi(UA)$ can be summarized as follows: For all terms ϕ, all user-assignment relations UA, and all traces $i \in \Sigma_B^*$, if $i\hat{}\langle done\rangle \in \mathsf{T}(SOD_\phi(UA))$, then $\mathsf{users}(i) \vdash_{UA} \phi$.

Let a process W that models a workflow be given. Let ϕ be a term that formalizes an SoD constraint, UA a user assignment, and PA a permission assignment. We call the parallel, partially synchronized composition of W, the RBAC process, and the SoD-enforcement process SOD_ϕ the *SoD-secure (workflow) process* SSW_ϕ. Formally,

$$SSW_\phi(UA, PA) = (W \underset{\Sigma_B}{\|} RBAC(UA, PA)) \underset{\Sigma}{\|} SOD_\phi(UA)\,.$$

If $SSW_\phi(UA, PA)$ engages in a business event $b.t.u$, then t is one of the next tasks in the workflow W, u is allowed to execute t with respect to UA and PA, and u is also authorized to execute a task according to the SoD-policy ϕ with respect to UA. In addition, $RBAC$ and SOD_ϕ can synchronously engage in an admin event and change their user assignments accordingly.

3 Implementation

As motivated in the introduction, we want to empirically validate that an SoD-secure workflow process can be mapped to a scalable implementation that is employable for a realistic use case. Furthermore, our implementation demonstrates the concept of SoD as a Service and serves as basis for its assessment.

The mapping from an SoD-secure workflow process to software components is illustrated in Figure 1. We proceed by implementing W by a workflow engine, $RBAC$ by a user repository, and SOD_ϕ by a program that we call an *SoD-enforcement monitor*. Workflow engines and user repositories are well-established concepts and therefore we use off-the-shelf components to realize them. The standalone SoD-enforcement monitor, however, is a new concept, which we implemented from scratch (indicated by dark gray in Figure 1).

3.1 Technical Objectives

We aim at realizing an effective, practical, and efficient implementation. By effective we mean that the implementation fulfills its purpose. Namely, it should support the execution of arbitrary workflows, facilitate changing RBAC configurations, and correctly enforce SoD constraints that are specified as SoDA terms.

We understand practicability in the sense that the integration and configuration effort is moderate. The main components of our system should be loosely coupled in order to reduce the cost of integration and to allow the integration of pre-existing components, such as a legacy workflow system. Furthermore, the system should be configurable using standard means, e.g. a workflow definition, an RBAC configuration, and an SoD policy, rather than requiring additional, labor-intensive settings.

The performance of our implementation is a critical success factor for this work. We call the runtime of a system with a workflow engine and a user repository but without an SoD-enforcement monitor the *runtime baseline*. Our objective is to enforce SoD constraints efficiently, that is with a low overhead compared to the runtime baseline.

3.2 Architecture

Figure 1 shows our general approach to mapping the processes W, $RBAC$, and SOD_ϕ to three individual system components. The concrete software tools we use and their intercommunication is illustrated in Figure 2. Gray boxes again indicate the components that we developed versus those that are standardly available.

Workflow engine: We use the IBM WebSphere Process Server (WPS) [17] as workflow engine. WPS runs on top of the IBM WebSphere Application Server (WAS) [16], IBM's Java EE application server [13].

User repository: The IBM Tivoli Directory Server (TDS) [15] serves as a user repository. TDS is an LDAP server [28] whose LDAP schema we configured to support the RBAC relations.

SoD enforcement monitor: We implemented the SoD-enforcement monitor in Java and wrapped it as a web service, using Apache Axis [25], which runs on top of Apache Tomcat [26].

Along with the various web-service standards, many semi-formal business processes modeling languages have emerged. Among the high-level languages, the Business Process Modeling Notation (BPMN) [20] has gained considerable attention. Backed by numerous software vendors, the Web Service Business Process Execution Language (WS-BPEL) [3], or BPEL for short, is a popular standard for describing business processes at the implementation-level. A BPEL process definition can be directly executed by a workflow engine. The BPEL4People standard [2] is an extension of BPEL for describing human tasks. At design time, we define a workflow in BPEL, including BPEL4People extensions, and deploy it to WPS.

LDAP supports RBAC with the object class `accessRole`. Instances of this class represent a role and store the distinguished name of their members, typically instances of `inetOrgPerson`, in the field `member`. We send $\mathcal{U}$, $\mathcal{R}$, and UA, encoded in the LDIF format [28], to TDS or we administer them directly through the TDS web interface.

Using an ASCII version of the SoDA grammar, we encode SoDA terms as character strings. We send them to the SoD-enforcement monitor with a standalone client. The interface of our SoD-enforcement monitor implementation is described in detail in Appendix A.

By adopting a service-oriented architecture, we achieve a loose coupling between our three main system components. This allows us to integrate two off-the-shelf components and the newly developed SoD-enforcement monitor. Hence, we achieve the flexibility described in Section 3.1.

The downside of the SOA approach is the increased communication and serialization overhead. In order to determine whether a user is authorized to execute a task instance with respect to an SoD constraint, the SoD-enforcement monitor requires context information, which must be sent across the network. Our design decisions in this regard are explained in Section 3.5. As the performance measurements in Section 4.3 will show, the communication overhead is acceptable.

Similar trade-offs between flexible, distributed architectures with an increased communication overhead versus monolithic architectures with a smaller communication overhead have been made in the past. For example, the Hierarchical Resource Profile for XACML [4] proposes sending the hierarchy, based on which an access control decision is made, to the access control monitor along with the access request. As with our architecture, the access control monitor needs considerable context information to compute the access decision.

3.3 Enforcement of SoD Constraints

In this section, we explain how our prototype system implements an SoD-secure workflow process SSW_ϕ. The process SSW_ϕ engages in three kinds of events: business events, admin events, and the event *done*. The implementation and handling of admin events and the event *done* is straightforward and therefore not discussed. We take a closer look at business events and explain why every execution of a task instance in our system corresponds to a business event that is accepted by SSW_ϕ. A business event corresponds to a sequence of steps in our implementation.

Consider the SoD-secure workflow process

$$SSW_\phi(UA, PA) = (W \underset{\Sigma_B}{\|} RBAC(UA, PA)) \underset{\Sigma}{\|} SOD_\phi(UA),$$

for a SoDA term ϕ, a user assignment UA, a permission assignment PA, and a workflow process W that models a workflow w. Assume that $i \in \mathsf{T}(SSW_\phi(UA, PA))$ corresponds to an unfinished workflow instance of w. Let UA' be the user assignment after executing the administrative events in i.[1] Assume that t is a task in w. Furthermore, assume that t_i, an instance of t, is the next task instance that is executed. We now look at the state transitions of t_i.

Instantiation The creation of t_i is either triggered by the termination of the preceding task instance, corresponding to the rightmost business event in i or by the creation of the workflow instance i itself.

RBAC Authorization In SSW_ϕ, authorization decisions are only made by the $RBAC$ and the SOD_ϕ process and W simply defines the order that tasks must be executed. This is different in our system and also in most commercial workflow systems. For example, BPEL4People requires the definition of a query, called *people link*, for every task. When the workflow engine instantiates the task, it also executes the respective query against the user repository. The users who are returned are candidates for executing the newly created task instance.

For a user u, the process $RBAC(UA', PA)$ accepts the business event $b.t.u$ if u is assigned to one of the roles in $R_t = \{r \in \mathcal{R} \mid (r, t) \in PA\}$ according to UA'. Therefore, during design time, we specify t's people link in such a way that the user repository returns all users who are assigned to a role in R_t. In other words, the user repository keeps track of the user-assignment relation UA and the workflow definition specifies the permission-assignment relation PA. Implicitly, we assume a one-to-one relation between permissions and tasks.

WPS evaluates t's people link after every instantiation of t. Initially, the people link is sent to TDS (Arrow 1 in Figure 2). Afterwards, TDS returns the set of users $U_1 = \{u \in \mathcal{U} \mid \exists r \in R_t \,.\, (u, r) \in UA'\}$ to WPS (Arrow 2 in Figure 2).

Refine to SoD-compliant Users Next, we select those users from U_1 who are allowed to execute t_i with respect to ϕ and i. Formally, we compute the set of users $U_2 = \{u \in U_1 \mid i \,\hat{}\, \langle b.t.a \rangle \in \mathsf{T}(SOD_\phi(UA'))\}$.

WPS provides a plugin interface that allows one to post-process the sets of users returned by a user repository. We wrote a plugin for this interface that sends U_1, the role assignments of these users, $UA'_1 = \{(u, r) \in UA' \mid u \in U_1\}$, and the identifiers of w and i to the SoD-enforcement monitor (Arrow 3 in Figure 2). We refer to this web service call as a *refinement call*. A detailed interface definition is provided in Appendix A.

For every workflow, the SoD-enforcement monitor stores the corresponding SoDA term. Furthermore, it keeps track of the users who execute task instances (see step *Claim*). Together with the above mentioned inputs, this allows the computation of U_2. In Section 3.4, we discuss this computation in more detail. The output, U_2, is returned to WPS (Arrow 4 in Figure 2).

Display A user can interact with WPS through a personalized, web-based interface. Once a user has successfully logged into the system, WPS displays a list of task instances that the user is authorized to execute. We call this list the user's *inbox*. For every user $u \in U_2$, $i \,\hat{}\, \langle b.t.u \rangle \in SSW_\phi(UA, PA)$. Therefore, WPS displays t_i in the inbox of every user in U_2.

[1] In [5], this is formalized using an update function upd.

Claim In the workflow terminology, if a user requests to execute a task he is said to *claim* the task. One of the users in U_2 must claim t_i by clicking on t_i in his inbox. Assume the user u claims t_i. Instantaneously, t_i is removed from the inboxes of all other users. At this point, we must communicate to the SoD-enforcement monitor that u is executing t_i. In addition, we send the roles assigned to u to the monitor (Arrow 5 in Figure 2). We refer to this web service call as a *claim call*.

Termination Afterwards, u is prompted with a form whose completion constitutes the work associated with t_i. The work is completed when the form is submitted. If t_i is not a task instance that terminates the workflow instance, its termination triggers the instantiation of another task.

Summarizing, our system effectively enforces abstract SoD constraints as specified in Section 3.1. Arbitrary workflows, constrained by a possibly changing RBAC configuration and an abstract SoD policy, can be executed on WPS. The applicability of our approach is further demonstrated with the case study in Section 4.

3.4 Complexity

We now analyze the runtime complexity of our SoD-enforcement monitor implementation. In particular, we analyze the complexity of a refinement call. The complexity of claim calls are negligible compared to the complexity of refinement calls and is therefore not discussed.

In general, the problem of deciding whether a term is satisfied by a set of users is **NP**-complete [18]. The SoD-enforcement monitor must solve this decision problem for every user received through a refinement call. Therefore, it comes as no surprise that refinement calls have a worst-case exponential runtime complexity. However, we can show that the exponent remains small for moderate size workflows.

The parameters of a refinement call are a set of users, U_1, their role assignments, UA'_1, and the identifiers of i and w. Using the identifier of w, the monitor retrieves ϕ. With the identifier of i, it retrieves all the users who have executed task instances in i and their role assignments at that time. The monitor-internal data model is described in Appendix B.

For every $u \in U_1$, the SoD-enforcement monitor then computes whether $i \,\hat{}\, \langle b.t.u \rangle \in \mathsf{T}(SOD_\phi(UA'_1))$. This computation is executed $|U_1| = n$ times. Consider the $[\![.]\!]$-mapping given in Appendix C. The evaluation of a unit term can be performed in polynomial time in the size of $|\mathcal{U}|$ and $|\mathcal{R}|$; i.e. $p(|\mathcal{U}|, |\mathcal{R}|)$ for a polynomial p. In the worst case, $SOD_\phi(UA'_1)$ branches $2^{|\mathcal{U}|}$ times per operator in ϕ. If m is the number of operators, the worst-case runtime is therefore in $O(n\, m\, 2^{|\mathcal{U}|}\, p(|\mathcal{U}|, |\mathcal{R}|))$.

The exponential factor originates from the $\otimes$-operator, which causes $SOD_\phi(UA'_1)$ to branch for all disjoint subsets of $\mathcal{U}$. Let $U_{i+u} = \mathsf{users}(i) \cup \{u\}$, i.e. the set of users in business events in i and u. If we check whether $i \,\hat{}\, \langle b.t.u \rangle \in \mathsf{T}(SOD_\phi(UA'_1))$, the users in $\mathcal{U} \setminus U_{i+u}$ are not relevant. Therefore, we need not branch over all partitions of $\mathcal{U}$ but only over those of U_{i+u}. If ϕ does not contain a $^+$-operator, then the maximal number of users in business events in i is $m+1$ and therefore $|U_{i+u}| \leq m+2$. If ϕ does contain a $^+$-operator, then $|U_{i+u}| \leq |\mathcal{U}|$. Our implementation exploits these observations. Hence, its runtime complexity is in $O(n\, m\, 2^{|U_{i+u}|}\, p(|\mathcal{U}|, |\mathcal{R}|))$ for $|U_{i+u}|$ as discussed above.

Our experience with business process catalogs, such as the IBM Insurance Application Architecture (IAA) [14], is that workflows contain a good dozen human tasks on the average. Furthermore, most workflow languages allow the decomposition of workflows into sub-workflows. Given these numbers and observations, we conclude that the performance penalty imposed by the SoD as a Service approach remains acceptable for most workflows. We further investigate these issues and provide runtime measurements in Section 4.3.

3.5 Design Decisions and Assumptions

An SoD-enforcement process $SOD_\phi(UA)$ contains the user assignment UA and keeps track of administrative changes by engaging in admin events and modifying UA accordingly. Our SoD-enforcement monitor does not store the entire user-assignment relation. It receives all relevant assignments as call parameters and stores only those assignments of users who claimed a task instance. Although this approach increases the communication overhead between WPS and the SoD-enforcement monitor, it reduces unnecessary replication.

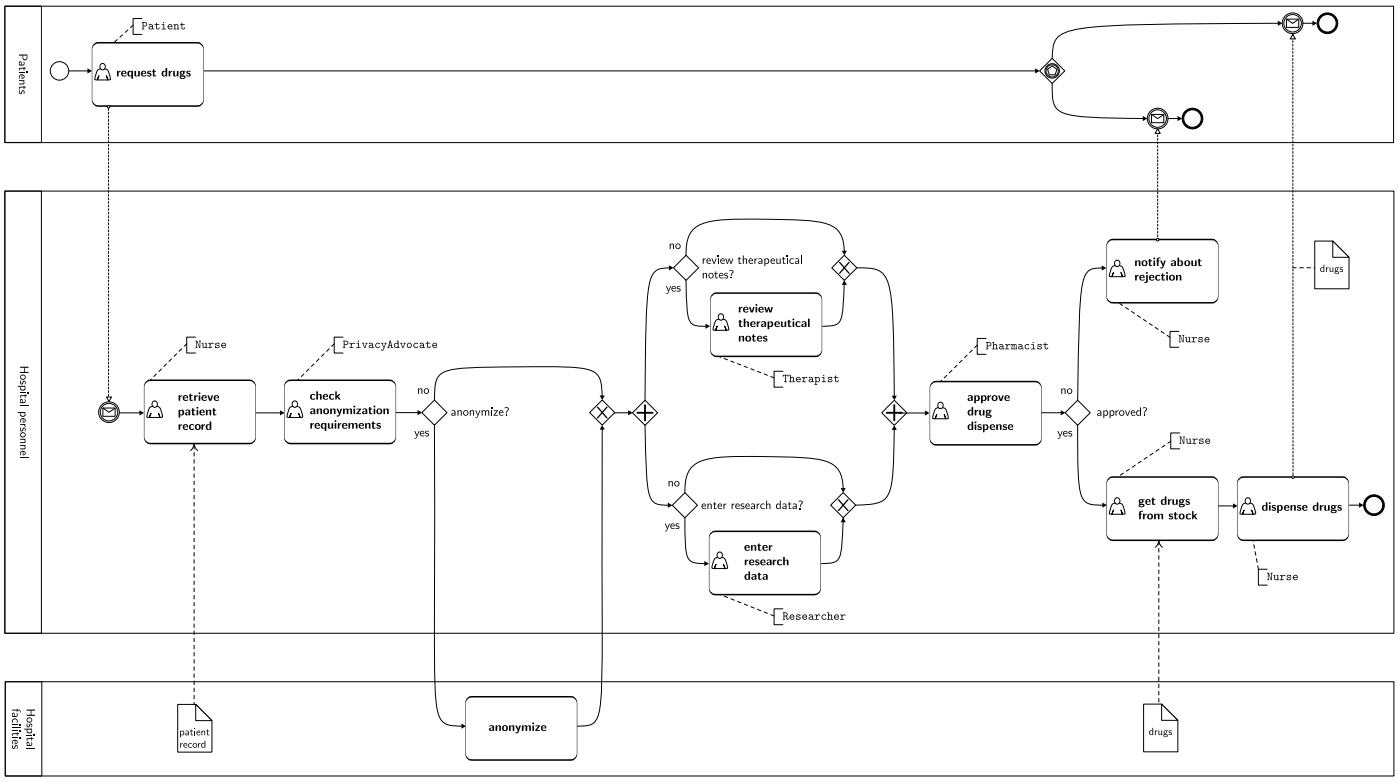

Figure 3: Case study: Drug dispensation workflow in BPMN

In large enterprises, a user repository may contain thousands of entries and only a few of them may be relevant with respect to a given workflow.

Our SoD-enforcement monitor is stateful because the enforcement of SoD constraints ranges over multiple tasks and may depend on role assignments. The service must therefore keep track of the users who execute task instances and the roles they act in at that time. Standard workflow engines such as WPS may store the users who executed task instances but they do not store the history of their role assignments. This information is stored in the SoD-enforcement monitor; the workflow engine and the user repository remain unchanged.

For simplicity, our SoD-enforcement monitor cannot cope with the abort or suspension of task instances. In practice, however, WPS users can return unfinished task instances or trigger the abortion of a workflow instance. Furthermore, we enforce exactly one term per workflow. This is not a limitation as two or more terms can be combined into a single term with the appropriate SoDA operators; e.g. $\sqcap$ for a conjunction or $\sqcup$ for a disjunction. If no SoD constraint has to be enforced, the term All^+, which is satisfied by every non-empty set of users, can be used.

The integration of an SoD-enforcement monitor with a workflow engine requires a means to refine the set of users that are authorized to execute a task. WPS' plugin interface allowes one to register a function for this purpose. Other workflow engines may not provide such an extensibility mechanism. Alternatively, users could also be filtered by intercepting the user repository's reply on the messaging level, e.g. by extending the enterprise service bus.

4 Case Study

4.1 Scenario

We illustrate SoD as a Service with a drug dispensation workflow taken from [19]. This workflow defines the tasks that must be executed to dispense drugs to patients within a hospital. The drugs dispensed within this

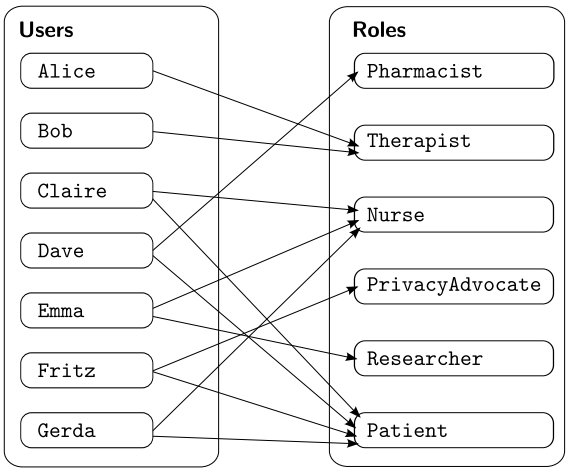

Figure 4: Initial user assignment *UA*

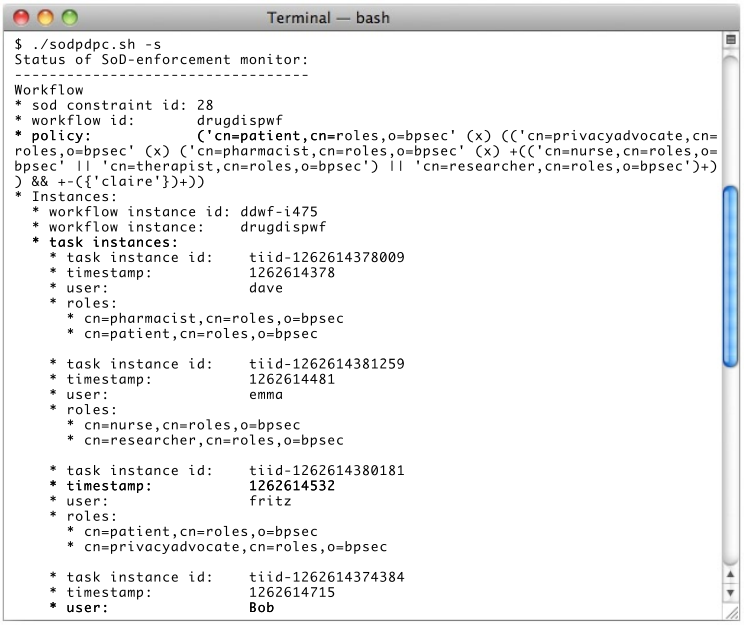

Figure 5: Screenshot of SoD-enforcement monitor client

process are either in an experimental state or very expensive and therefore require special diligence.

A BPMN model of the dispensation workflow is shown in Figure 3. Because BPMN does not provide a notation for specifying which kinds of users are allowed to execute a given task, we use BPMN annotations to augment the workflow definition with this information.

A workflow instance is started by a `Patient` who requests drugs by handing his prescription to a `Nurse`. The `Nurse` retrieves the patient's record from the hospital's database and forwards all data to a `PrivacyAdvocate` who checks whether the patient's data must be anonymized. If anonymization is required, this is done by a computer program. We ignore this task in our forthcoming discussion as we focus on tasks that are executed by humans. If therapeutical notes are contained in the prescription, they are reviewed by a `Therapist`. In parallel, research-related data is added by a `Researcher`, if the requested drugs are in an experimental state. Finally, a `Pharmacist` either approves the dispensation and a `Nurse` collects the drugs from the stock and gives them to the patient, or he denies the dispensation and a `Nurse` informs the `Patient` accordingly.

Fraudulent or erroneous drug dispensations could jeopardize patients' health, may violate regulations, and could severely impact the hospital's finances and reputation. We assume that the hospital imposes SoD constraints on this workflow in order to reduce these risks. A `Pharmacist` may not dispense drugs to himself; i.e. he should not act as `Patient` and `Pharmacist` within the same workflow instance. Similarly, the `Nurse` who prepares the drugs should not be the same user as the `Pharmacist` who approves the dispensation. Furthermore, the `PrivacyAdvocate` who checks whether the patient's prescription and data should be anonymized to protect patient privacy must be different from any other user involved in the same workflow instance. Finally, the nurse `Claire` may not be involved in the dispensation due to her drug abuse history. However, as a `Patient` she may receive drugs. All these constraints are encoded by the term $\phi = \texttt{Patient} \otimes ((\neg\{\texttt{Claire}\})^+ \sqcap (\texttt{PrivacyAdvocate} \otimes \texttt{Pharmacist} \otimes (\texttt{Nurse} \sqcup \texttt{Researcher} \sqcup \texttt{Therapist})^+))$.

4.2 Configuration and Execution

We defined the drug dispensation workflow in BPEL, extended by BPEL4People, and deployed it on WPS. Using the web interface of TDS, we set up an initial user assignment *UA* as depicted in Figure 4. Furthermore, we configured WPS to use TDS as user repository.

We represented the SoD term ϕ as string and sent it to the SoD-enforcement monitor. In addition, we

	Refinement calls			Claim calls		
	T_t	T_c	T	T_t	T_c	T
t_1	187.5	3.1	190.6	190.6	22.0	212.6
t_2	186.0	7.8	193.8	176.6	22.0	198.6
t_3	181.2	7.8	189.0	176.4	23.5	199.9
t_4	181.4	12.5	193.9	181.2	17.4	198.6
t_5	179.8	15.5	195.3	178.3	26.4	204.7
t_6	176.7	26.5	203.2	184.4	31.2	215.6
t_7	181.3	45.3	226.6	171.9	27.9	199.8

Figure 6: Service call times in ms

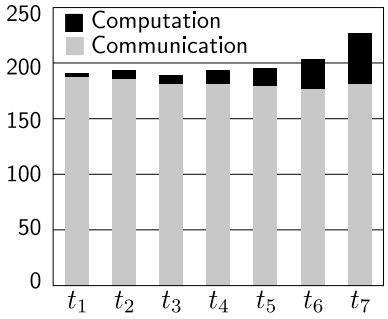

Figure 7: Refinement call

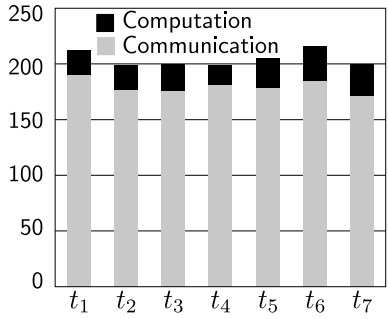

Figure 8: Claim call

configured the plugin interface of WPS to use our plugin to post-process user repository requests, i.e. to send them to the SoD-enforcement monitor and to inform the SoD-enforcement monitor about users who claimed task instances.

We executed instances of the workflow using the default web interface of WPS. For example, we log into WPS as `Dave` and start a workflow instance by submitting a form that corresponds to the task `request drugs`. Next, we log into the system as `Emma`, claim the newly created instance of the task `retrieve patient record`, and execute it by filling in the corresponding form. As `Fritz`, we claim and execute the instance of `check anonymization requirements`. The drugs requested by `Dave` do not required additional research data. However, we review the therapeutical notes included in `Dave`'s prescription as `Bob`. Because a `Patient` may not dispense drugs to himself, `Dave` must not approve the dispensation. Because there is no other user available who acts in the role `Pharmacist`, which is required for the approval, we add a `Pharmacist`-assignment for `Alice` to *UA* by executing the corresponding administrative command in TDS. Now, we can approve the dispensation as `Alice`. Finally, acting as `Gerda`, we get the drugs from the stock and dispense them. This workflow instance corresponds to the trace

$$
\begin{aligned}
i = \langle\ & b.\texttt{request drugs.Dave},\ b.\texttt{retrieve patient record.Emma},\\
& b.\texttt{check anonymization requirements.Fritz},\ b.\texttt{review therapeutical notes.Bob},\\
& a.\texttt{addUA.Alice.Pharmacist},\ b.\texttt{approve drug dispense.Alice},\\
& b.\texttt{get drugs from stock.Gerda},\ b.\texttt{dispense drugs.Gerda}\ \rangle.
\end{aligned}
$$

Figure 5 shows the status message of our SoD-enforcement monitor client after executing i in WPS. By invoking the function `status` of the SoD-enforcement monitor, the client retrieves an overview of the currently stored SoD constraints, executed workflow and task instances, and the users who executed them with their roles.

4.3 Performance

Compared to the runtime baseline, the runtime of our prototype system is increased by a refinement and a claim call for every task instance. In the following, we discuss the performance penalty imposed by these calls. We call the time it takes to call a web service and to retrieve its return values the *total runtime* of a web service. We decompose this runtime into two parts: the *transmission time* encompasses the time to serialize, transmit, and deserialize the exchanged data and the *computation time* is the time to execute the service's functionality

Consider the workflow instance presented in Section 4.2. We executed ten equivalent workflow instances in our prototype system and measured the total runtime for each refinement and claim call. We refer to a task instance of `request drugs` as t_1, to an instance of `retrieve patient record` as t_2, etc. Figure 6 shows the average transmission time T_t, computation time T_c, and the total runtime T per task instance in milliseconds (ms). Figures 7 and 8 illustrate the same results graphically.

The transmission time required for the SoD-enforcement monitor depends on various factors including the network throughput, the network latency, the payload size, and also the time taken to serialize Java objects to SOAP message parameters with the Apache Axis framework. We run the service client and the SoD-enforcement monitor on two different computers at the same geographical location, connected by a standard enterprise

network with an average latency of 1ms. Both computers have off-the-shelf configurations.[2] The transmission time averages between 150ms and 200ms per call.

The computation time for claim calls was always around 24ms during our measurements. The computation time of refinement calls, however, increased with the number of executed task instances. As explained in Section 3.4, the $\otimes$-operators in ϕ cause this time to increase exponentially (see Figure 7).

Finally, we compare the total runtime of these additional calls to the time it takes to execute a task instance in a system without an SoD-enforcement monitor. The refinement call increases the time between the termination of a preceding task instance and the moment the new task instance is ready to be claimed by a user. The durations for these steps range between 2 and 15 seconds, depending on the load of WPS and the latest patches installed on it. Claiming a new task instance takes only 1–3 seconds. A user clicks on the instance in his inbox and the corresponding form is displayed on his screen. In both cases, the additional runtime caused by the SoD-enforcement monitor calls is an order of magnitude smaller than the runtime of a task instance without SoD-enforcement monitor calls.

Given the observations made in Section 3.4 and the times reported here, we conclude that the integration of our SoD as a Service implementation into an existing workflow system imposes a performance penalty below 10%. Consequently, we have achieved all the technical objectives described in Section 3.1.

5 Related Work

There are many languages for modeling workflows. We used BPMN to specify the workflow in our case study and BPEL for its implementation. Backed by two large consortia, OASIS and OMG respectively, these two standards are both popular in industry. Different formalisms have been used to give them a precise semantics, e.g. Petri nets and process algebras. To give a concrete example, Wong and Gibbons describe BPMN using CSP [30].

A classification of SoD constraints is given in [12, 24]. In general, SoD mechanisms are tightly coupled with the workflow to be controlled [23, 6]. Li and Wang's SoD algebra [18] is the first approach that enables an abstract specification of SoD constraints, leaving open which users are allowed to perform which tasks. They proved that the complexity of checking whether a SoDA term is satisfied by a set of users is **NP**-complete [18]. Furthermore, they developed algorithms for the static enforcement of high-level SoD constraints, formalized in SoDA [29]. However, their approach is only applicable to a subset of terms.

Dynamic SoD enforcement is more flexible than static enforcement. However, static mechanisms are typically favored over dynamic mechanisms because of their lower complexity and the relative ease in integrating them with existing systems. BPEL4People supports basic dynamic SoD constraints [2]. Although not fully specified, the query language for people links in BPEL4People allows one to exclude users who have executed previous tasks from being assigned to new task instances in the same workflow instance. By using SoDA terms, our architecture supports more expressive constraints than BPEL4People.

Paci et. al. propose another access control extension for BPEL [21] based on the work of Crampton [9]. Authorizations, including SoD constraints that range over relations between users, are enforced by a web service, which pools all information that is relevant for enforcement: the history of workflow instances, the RBAC configuration, and SoD constraints. The underlying workflow model, however, does not support loops, which is in conflict with the expressiveness of BPEL. Moreover, unlike our work, their constraint language requires a tight coupling between constraints and the workflow definition and does not support changing authorizations.

Chadwick et. al. propose an SoD policy model that spans multiple sessions [7]. Furthermore, they discuss the implementation of their model in the PERMIS Privilege Management Infrastructure. Although their architecture includes LDAP directories for managing users and their roles, SoD checking is realized within the RBAC decision engine and history information is stored in memory. In contrast, our implementation achieves full decomposability by extracting the SoD enforcement monitor as an independent component. Their concept

[2] Client: MS Windows XP on Intel Core Duo 2 GHz processor with 3 GB RAM. Server: MS Windows Server 2003 on Intel Xeon 2.9 GHz processor with 4 GB RAM.

of multi-session SoD also addresses changing role assignments, but only within the scope of a business context.

6 Conclusion and Future Work

With this work, we addressed two major trends in Information Security and business computing. First, we presented a flexible mechanism for enforcing internal controls, with applications to fraud reduction and compliance with regulatory requirements. Second, we introduced the paradigm of SoD as a Service, which enables the dynamic integration and configuration of this enforcement mechanism in a service-oriented environment. Both contributions match well with the dynamics of today's business environments including changing regulations and and organizational structures.

Concretely, our work bridges the gap between the theoretical models of [5] and a realistic implementation in an enterprise workflow environment with the prototype system presented in Section 3 and the case study described in Section 4. We thereby empirically validate the work of Basin et. al. Our implementation also serves as a proof-of-concept for SoD as a Service. The SoD-enforcement monitor is configurable through web service calls and provides its SoD-enforcement functionality as a service. Furthermore, it accounts for changing authorizations and therefore also to organizational changes. The choice of software components for our architecture illustrates how SoD as a Service enables the integration of new internal controls into existing workflow environments. We discussed the challenges that inherently arise if such flexibility is pursued. An increased communication overhead needs to be balanced against duplication of contextual information.

The key component of our system is the SoD-enforcement monitor. As expected from the complexity results in [18] and the constructions in [5], the worst-case runtime is exponential in the total number of users in the system. In Section 3.4, we show that for a large class of terms the worst-case runtime complexity is exponential only in the number of task instances in a workflow instance. Because this number grows during the execution of a workflow instance, the runtime of the corresponding SoD-enforcement monitor call increases over the lifetime of a workflow instance. Thus, a careful workflow design or the decomposition of a workflow into sub-workflows reduces the performance penalty imposed by the SoD-enforcement monitor. A promising idea for future work is to decompose SoDA terms into subterms and to enforce them on critical subsets of human tasks of workflows. This would further reduce the runtime of our SoD-enforcement monitor and allow task-specific constraints.

Finally, we comment on some synchronization issues that arose in Section 3.3. There, we described the sequence of steps that constitute the execution of a task instance. We implicitly assumed that these steps do not interleave with the execution of other task instances. In practice, however, this could be the case. A simple solution would work as follows: When a task instance is claimed, all other task instances in the same workflow instance that have not yet been claimed must re-execute their refinement step. Similar problems arise if task instances are aborted. We plan to address these synchronization problems in future work.

References

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

[2] A. Agrawal, et.al. *WS-BPEL extension for people (BPEL4People), v1.0*. 2007.

[3] A. Alves, et.al. Web services business process execution language (WS-BPEL), v2.0. OASIS Standard, 2007.

[4] A. Anderson. Hierarchical resource profile of XACML, v2.0. OASIS Standard, 2005.

[5] D. A. Basin, S. J. Burri, and G. Karjoth. Dynamic enforcement of abstract separation of duty constraints. *Proc. of the 14th European Symposium on Research in Computer Security (ESORICS)*, pp. 250–267, 2009.

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

[7] D. W. Chadwick, W. Xu, S. Otenko, R. Laborde, and B. Nasser. Multi-session Separation of Duties (MSoD) for RBAC. *Proc. of the 23rd Int. Conference on Data Engineering Workshop* (ICDE), pp. 744–753, 2007.

[8] B. Cleary. Employee role changes and SocGen: Good lessons from a bad example. SC Magazine, 2008.

[9] J. Crampton. A reference monitor for workflow systems with constrained. *Proc. of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 38–47, 2005.

[10] European fraud survey 2009. Ernest & Young, Technical report, 2009.

[11] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[12] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. *Proc. of the 19th IEEE Symposium on Security and Privacy (S&P)*, pp. 172–183, 1998.

[13] S. Haugland, M. Cade, and A. Orapallo. *J2EE 1.4: The big picture*. Prentice Hall, 2004.

[14] IBM Insurance Application Architecture (IAA). `www.ibm.com/software/sw-library/en_US/detail/N440171L95655L23.html`.

[15] IBM Tivoli Directory Server (TDS) v6. `www.ibm.com/software/tivoli/products/directory-server`.

[16] IBM WebSphere Application Server (WAS) v6.1. `www.ibm.com/software/webservers/appserv/was/`.

[17] IBM WebSphere Process Server (WSP) v6.2. `www.ibm.com/software/integration/wps/`.

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

[19] D. Marino, et.al. Deliverable D1.2.1: Master scenarios. EU Project MASTER (`www.master-fp7.eu`), 2009.

[20] Business Process Modeling Notation (BPMN), v1.2. OMG Standard, 2009.

[21] F. Paci, F. E. Bertino, and J. Crampton. *An Access-Control Framework for WS-BPEL*. Int. Journal of Web Services Research, pp. 20–43, 2008.

[22] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 2005.

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

[24] R. Simon and M. E. Zurko. Separation of duty in role-based environments. *Proc. of the 10th IEEE Workshop on Computer Security Foundations (CSFW)*, pp. 183–194, 1997.

[25] Apache Axis2, v1.5. The Apache Software Foundation, `http://ws.apache.org/axis2`, 2009.

[26] Apache Tomcat, v6. The Apache Software Foundation. `http://tomcat.apache.org`, 2009.

[27] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *IEEE Computer*, 36:38–44, 2003.

[28] S. Tuttle, et.al. *Understanding LDAP - design and implementation*. IBM Redbooks, 2004.

[29] Q. Wang and N. Li. Direct static enforcement of high-level security policies. *Proc. of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 214–225, 2007.

[30] P. Y. H. Wong and J. Gibbons. A process-algebraic approach to workflow specification and refinement. *Proc. of the 6th Int. Symposium on Software Composition (SC)*, pp. 51–65, 2007.

A SoD-Enforcement Monitor Interfaces

In the following, we document the interface of our SoD-enforcement monitor by means of Java method signatures. In order to keep the serialization and communication overhead small, we used the basic Java type `String` whenever possible.

```
boolean enforce(
    String policy,
    String workflowId
)
```

```
boolean stopEnforcing(
    String workflowId
        )
```

The methods enforce and stopEnforcing are used to deploy and undeploy an SoD constraint for a workflow that is identified by its wokflowId. The SoD constraint is formalized as SoDA term and provided to the SoD-enforcement monitor through the parameter policy.

```
String[] allowedForNextTask(
    String[] users,
    String[] roles,
    UserRoleRelation[] ur,
    String taskId,
    String workflowInstanceId,
    String workflowId
)
```

A refinement call corresponds to the invocation of the method allowedForNextTask. As described in Section 3.3, the SoD-enforcement monitor receives the workflowId in order to retrieve the workflow's SoD constraint. Using workflowInstanceId, it retrieves the users who executed previous task instances in the corresponding workflow instance. The set of users U_1 and their role assignments are encoded by the arrays users, roles, and ur. For bookkeeping, we transmit also taskId, the identifier of the instantiated task. The return value is a subset of the users in users, denoted U_2 in Section 3.3.

```
boolean claim(
    String user,
    String[] roles,
    String taskId,
    String workflowInstanceId,
    String workflowId
)
```

An invocation of the method claim corresponds to a claim call as described in Section 3.3. Using the identifiers workflowId and workflowInstanceId, the SoD-enforcement monitor determines how to relate and store the user user and his roles roles to previously executed task instances. Again, taskId is transmitted for bookkeeping.

```
String getStatus()
```

This method is used to query the status of the SoD-enforcement monitor. It returns an overview of the workflows, workflow instances and executed task instances that are stored by the service.

B SoD-Enforcement Monitor Data Model

Figure 9 depicts the database schema of the SoD-enforcement monitor using the UML Data Modeling Profile. It reflects the definitions in Section 2: A workflow is instantiated an arbitrary number of times. Each of these instances contains an arbitrary number of task instances. For every task instance, we store the user who executed it and the roles he was acting in at that time.

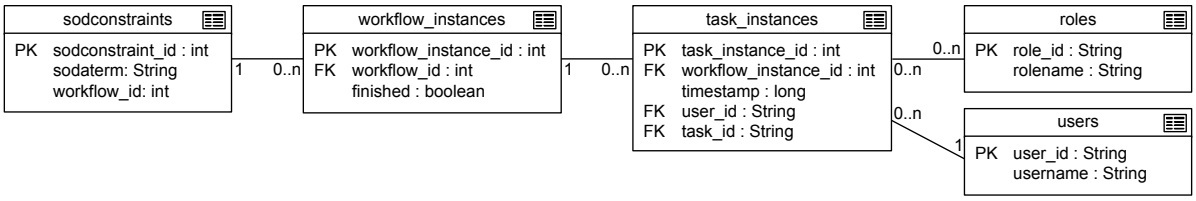

Figure 9: Database schema

C Mapping SoDA Terms to CSP Processes

We recall the mapping from SoDA terms to CSP processes [5]. A *unit term* is a term that does not contain the operators $\otimes$, $\odot$, and $^+$. More intuitively, a unit term is only satisfied by a set of users that contains only one user and is therefore mapped to one business event in a process, modeling the execution of a task instance.

With respect to the runtime complexity of a process resulting from Definition 1, Rule (6) is particularly critical as it causes an exponential branching over all disjoint subsets of U. We describe how to lower this factor in Section 3.4.

Definition 1 (Mapping $[\![.]\!]^U_{UA}$)**.** *Given a set of users U, a user assignment UA, and a term ϕ, the mapping $[\![\phi]\!]^U_{UA}$ returns a CSP process parametrized by UA. For a unit term ϕ_{ut} and terms ϕ and ψ, the mapping $[\![.]\!]^U_{UA}$ is defined in Figure 10.*

$$
\begin{aligned}
(1)\quad [\![\phi_{ut}]\!]^U_{UA} &= b?t:\mathcal{T}?u:\{u'\in U \mid \{u'\}\vdash_{UA}\phi_{ut}\}\rightarrow FIN \\
&\quad \Box\ a.\texttt{addUA}?u:\mathcal{U}?r:\mathcal{R}\rightarrow [\![\phi_{ut}]\!]^U_{UA\,\cup\,\{(u,r)\}} \\
&\quad \Box\ a.\texttt{rmUA}?u:\mathcal{U}?r:\mathcal{R}\rightarrow [\![\phi_{ut}]\!]^U_{UA\,\setminus\,\{(u,r)\}} \\
(2)\quad [\![\phi^+_{ut}]\!]^U_{UA} &= b?t:\mathcal{T}?u:\{u'\in U \mid \{u'\}\vdash_{UA}\phi_{ut}\}\rightarrow (FIN\ \Box\ [\![\phi^+_{ut}]\!]^U_{UA}) \\
&\quad \Box\ a.\texttt{addUA}?u:\mathcal{U}?r:\mathcal{R}\rightarrow [\![\phi^+_{ut}]\!]^U_{UA\,\cup\,\{(u,r)\}} \\
&\quad \Box\ a.\texttt{rmUA}?u:\mathcal{U}?r:\mathcal{R}\rightarrow [\![\phi^+_{ut}]\!]^U_{UA\,\setminus\,\{(u,r)\}} \\
(3)\quad [\![\phi\sqcup\psi]\!]^U_{UA} &= [\![\phi]\!]^U_{UA}\ \Box\ [\![\psi]\!]^U_{UA} \\
(4)\quad [\![\phi\sqcap\psi]\!]^U_{UA} &= [\![\phi]\!]^U_{UA}\ \underset{\Sigma}{\parallel}\ [\![\psi]\!]^U_{UA} \\
(5)\quad [\![\phi\odot\psi]\!]^U_{UA} &= [\![\phi]\!]^U_{UA}\ \underset{\{done\}\,\cup\,\Sigma_A}{\parallel}\ [\![\psi]\!]^U_{UA} \\
(6)\quad [\![\phi\otimes\psi]\!]^U_{UA} &= \underset{\{\,(U_\phi,U_\psi)\ \mid\ U_\phi\cup U_\psi=U \text{ and } U_\phi\cap U_\psi=\varnothing\,\}}{\Box}\ [\![\phi]\!]^{U_\phi}_{UA}\ \underset{\{done\}\,\cup\,\Sigma_A}{\parallel}\ [\![\psi]\!]^{U_\psi}_{UA}
\end{aligned}
$$

Figure 10: Mapping SoDA terms to CSP processes