Appeared in:
Proc. of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11).
March 22–24, 2011, Hong Kong, China.

Separation of Duties as a Service

David Basin [†]
basin@inf.ethz.ch

Samuel J. Burri [†,‡]
sbu@zurich.ibm.com

Günter Karjoth [‡]
gka@zurich.ibm.com

[†] ETH Zurich, Department of Computer Science, Switzerland
[‡] IBM Research, Zurich Research Laboratory, Switzerland

ABSTRACT

We introduce the concept of Separation of Duties (SoD) as a Service, an approach to enforcing SoD requirements on workflows and thereby preventing 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. In this paper, we describe an implementation of SoD as a Service, which extends a widely used, commercial workflow system, and discuss its performance. We present a drug dispensation workflow deployed in a hospital as case study to demonstrate the feasibility and benefits of our proof-of-concept implementation.

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. These advances enable 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 developments and new business models [22]. 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 [9]. *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 [16] 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 toward 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 be quickly adapted to changing regulations and threats. Past research has largely ignored the impact of changing authorizations on running workflows. The European fraud survey of Ernest & Young confirms that organizational changes, triggered by acquisitions and job cuts, are among the major sources of fraud [9]. In [5], we showed how to bridge the gap between workflow-independent SoD constraints, formalized as SoDA terms, and their enforcement in a general workflow model [5]. Our approach also accounts for changing authorizations and thereby generalizes the original SoDA semantics [16].

Concretely, we describe the architecture and implementation of an SaaS instance, called *SoD as a Service*, for enforcing SoD constraints on workflows. We illustrate how off-the-shelf, widespread software components can be combined and extended to improve internal controls, while remaining flexible with respect to change. Our theoretical models from [5] serve as blueprint. Although the runtime complexity of these models is exponential, our implementation achieves an acceptable runtime performance for workflows used in practice. Through an extensive and realistic case study, we test the applicability of SoD as a Service to a real-world scenario and identify critical design decisions. We carry out performance measurements that confirm the results of our complexity analysis. A more detailed description of the results presented in this paper is given in the technical report [6].

Our first contribution is the introduction of the concept of SoD as a Service, 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. In exchange for a moderate increase in 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. Our architecture and implementation pinpoint the design decisions one faces when trying to achieve this kind of modularity and flexibility. Our second contribution is an empirical validation of the theoretical models in [5].

2. BACKGROUND

In industry, workflows are specified using modeling languages such as the Business Process Modeling Notation (BPMN) [18]. Different formalisms have been used to give workflow languages a precise semantics. We use the process algebra *CSP* [20] to analyse our architecture's runtime complexity. Furthermore, CSP facilitates a precise and compact description of our architecture as the composition of simple components.

CSP describes a system as a set of communicating processes. An *event* is the smallest unit of activity; let Σ denote the set of all events. 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 t_1 and t_2, their concatenation is denoted $t_1 \hat{}\ t_2$. A *process* describes a communication pattern. The denotational semantics of CSP defines the behavior of a process P as a prefix-closed set of traces $\mathsf{T}(P) \subseteq \Sigma^*$, each describing a possible execution of P. For a trace t and an event e, if $t \in \mathsf{T}(P)$ we say that P *accepts* t and if $t\hat{}\langle e \rangle \in \mathsf{T}(P)$ we say that P *engages* in e (after P accepted t). 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, $P \underset{E}{\|} Q$ 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

A unit of work is called a *task*. Because SoD constraints are concerned with 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 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 model the execution of a task $t \in \mathcal{T}$ by a user $u \in \mathcal{U}$ by the composite event $b.t.u$, where b indicates that this is a *business event*. The set of all business events is Σ_B. In addition, 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 $i \in \Sigma^*$, the function $\mathsf{users}(i)$ returns the set of users who executed a task in i, *e.g.* $\mathsf{users}(\langle b.t_1.\mathtt{Bob}, b.t_2.\mathtt{Claire}, b.t_3.\mathtt{Bob}, done\rangle) = \{\mathtt{Bob}, \mathtt{Claire}\}$.

2.2 Authorizations

We use *Role-based Access Control (RBAC)* [10] to model authorizations. Let a set of *roles* $\mathcal{R}$, a *user-assignment (relation)* $UA \subseteq \mathcal{U} \times \mathcal{R}$, and a *permission-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 is 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. Users and their credentials, including their assignments to roles, are typically stored and administrated in a *user repository*.

Let (UA, PA) be an RBAC configuration. The *RBAC process* $RBAC(UA, PA)$ models the enforcement of role-based authorizations and the administration of UA. $RBAC(UA, PA)$ engages in a business event $b.t.u$ if u is authorized to execute t with respect to (UA, PA). Furthermore, we model the administration of user-assignment relations with a set of *administrative events* Σ_A. For every user $u \in \mathcal{U}$ and every role $r \in \mathcal{R}$, Σ_A contains the administrative events $a.\mathtt{rmUA}.u.r$ and $a.\mathtt{addUA}.u.r$, where a indicates that they are administrative events. $RBAC(UA, PA)$ 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 $RBAC(UA \backslash \{(u, r)\}, PA)$ afterward. In other words, $a.\mathtt{addUA}.u.r$ adds the tuple (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 [16] 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 assignments to roles in UA comply with the SoD constraint described by ϕ. As examples, 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 may 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.

In [5], we generalize the original SoDA semantics to a trace semantics that also accounts for changing authorizations. We thereby 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, we 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 administrative events that modify UA. The relation between the satisfaction of ϕ by a set of users and the acceptance of a trace by $SOD_\phi(UA)$ is 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,

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

If $SSW_\phi(UA, PA)$ engages in a business event $b.t.u$, then t is one of the next tasks in the workflow modeled by 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 administrative event and change their user assignments accordingly.

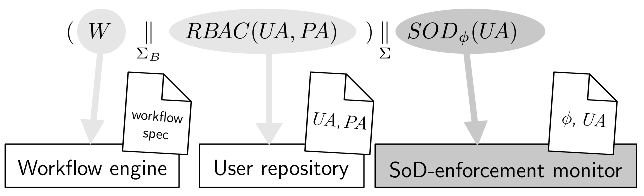
3. IMPLEMENTATION

In the following, we describe an implementation of SoD as a Service. Our goal is to demonstrate the flexibility of this approach, to analyze its scalability, and to identify performance-critical parameters. We use the SoD-secure process as blueprint for our implementation because its subprocesses naturally map to components of a SOA as illustrated in Figure 1. The components' interfaces can be inferred from the sets of events on which the respective processes synchronize. We proceed by implementing W by a workflow engine, $RBAC$ by a user repository, and SOD_ϕ by a program called an *SoD-enforcement monitor*. Workflow engines and user repositories are well-established concepts. We use off-the-shelf components to realize them. The standalone SoD-enforcement monitor, however, is something fundamentally new. Hence, we implement it 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.



$(W \underset{\Sigma_B}{\parallel} RBAC(UA, PA)) \underset{\Sigma}{\parallel} SOD_\phi(UA)$
workflow spec
UA, PA
ϕ, UA
Workflow engine
User repository
SoD-enforcement monitor

Figure 1: From theory to practice

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 of mapping the processes W, $RBAC$, and SOD_ϕ to three individual system components. The concrete software tools we use and their intercommunication are 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) [15] as workflow engine. WPS runs on top of the IBM WebSphere Application Server (WAS) [14], IBM's Java EE application server.

User repository: The IBM Tivoli Directory Server (TDS) [13] serves as a user repository. TDS is an LDAP server 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 [21] running on top of Apache Tomcat.

Along with the various web-service standards, many semi-formal business process modeling languages have emerged. 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. BPEL4People [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 encode $\mathcal{U}$, $\mathcal{R}$, and UA in LDAP's export format LDIF and send it to TDS, or we administer them directly through 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 [6].

By adopting a service-oriented architecture, we achieve a loose coupling between our three main system components.

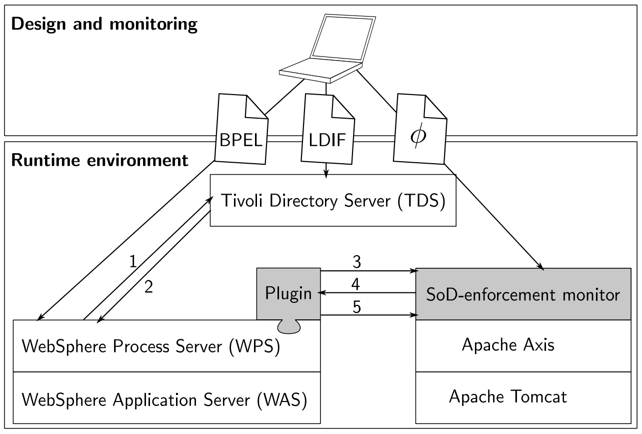


Figure 2: Architecture

This allows us to integrate two off-the-shelf components and our 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. 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 and the performance analysis in Section 4.3 show that 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 an 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, administrative events, and the event *done*. The implementation and handling of administrative 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 the execution of 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, an RBAC configuration (UA, 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. 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. We refer to an arrow labeled with n in Figure 2 as An.

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

RBAC Authorization: In SSW_ϕ, authorization decisions are made by the $RBAC$ and the SOD_ϕ process and W simply defines the order in which 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 executes the respective query against the user repository. The returned users 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 (A1). 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 (A2).

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. Namely, we compute the set of users $U_2 = \{u \in U_1 \mid i\,\hat{}\,\langle b.t.u\rangle \in \mathsf{T}(SOD_\phi(UA'))\}$.

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, their assignments to roles $UA'_1 = \{(u, r) \in UA' \mid u \in U_1\}$, and the identifiers of w and i to the SoD-enforcement monitor (A3). We refer to this web service call as a *refinement call*. See [6] for a detailed interface description.

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. The output, U_2, is returned to WPS (A4).

Display: A user can interact with WPS through a personalized, web 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.

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 (A5). 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.

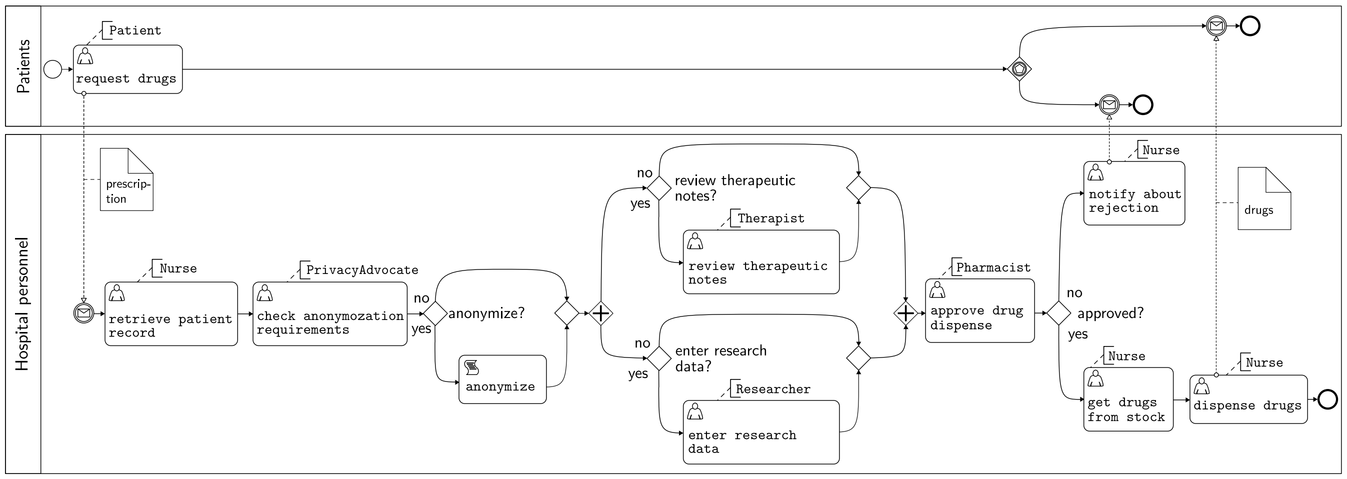


Figure 3: Case study: Drug dispensation workflow in BPMN

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

Due to space limitations, we present the complexity results for our SoD-enforcement monitor implementation without proof and refer to [6] for more details. In particular, we discuss the runtime 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 [16]. 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 in $O(|U_1|\, m\, 2^e\, p(|\mathcal{U}|, |\mathcal{R}|))$ where m is the number of operators in ϕ and p is a polynomial function. If ϕ contains no $+$-operators then $e \leq m$ and if ϕ contains $+$-operators then $e \leq |\mathcal{U}|$. The number of operators is typically linear in the number of tasks of the workflow. Our experience with business process catalogs, such as the IBM Insurance Application Architecture (IAA) [12], 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 is acceptable for most workflows.

3.5 Design Decisions and Assumptions

An SoD-enforcement process $SOD_\phi(UA)$ is parametrized by the user assignment UA and keeps track of administrative changes by engaging in administrative events and modifying UA accordingly. Our SoD-enforcement monitor, however, does not store all tuples of UA. It receives all relevant tuples as call parameters and stores only those 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. 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 user assignments. The service must therefore keep track of the users who execute task instances and the roles they act in at that time. Workflow engines such as WPS may store the users who executed task instances but they do not store the history of their user 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 to the workflow engine or trigger the abortion of a workflow instance. Further design considerations are elaborated in [6].

4. CASE STUDY

4.1 Scenario

We illustrate SoD as a Service with a drug dispensation workflow from [17]. This workflow defines the tasks that must be executed to dispense drugs to patients within a hospital. The drugs dispensed in this 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. We use BPMN annotations to define the permission-assignment relation, *e.g.* a `Pharmacist` is authorized to execute instances of the task `approve drug dispense`. 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 human tasks. If therapeutic 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 may jeopardize the patients' health, may violate regulations, and could severely impact the hospital's finances and reputation. We assume that the hospital enforces SoD constraints 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` 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.

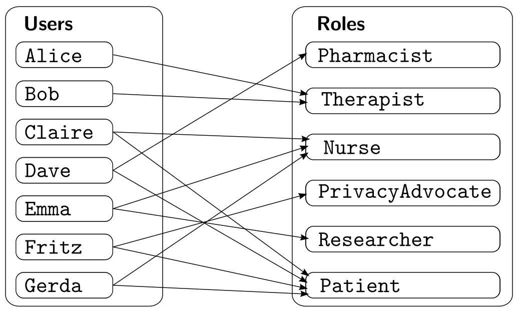


Figure 4: Initial user-assignment relation *UA*

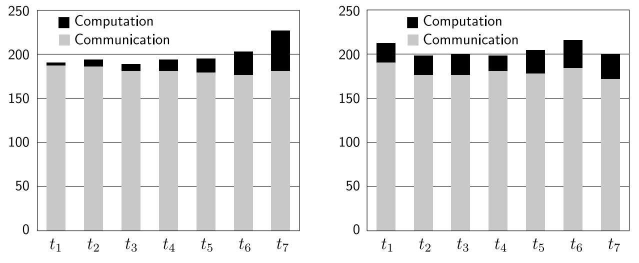
We represented the SoD term ϕ as a string and sent it to the SoD-enforcement monitor. In addition, we configured the plugin interface of WPS to use our plugin to post-process user repository requests, *i.e.* to send them to the SoD-enforcement monitor and to inform it about users who claimed task instances.

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 require additional research data. However, we review the therapeutic 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 an assignment of `Alice` to `Pharmacist` 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.

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 *communication 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

We executed ten workflow instances like the one outlined in the previous section and measured the total runtime for each refinement and claim call. We refer to an instance of `request drugs` as t_1, to an instance of `retrieve patient record` as t_2, etc. Figure 5 illustrates the averaged communication and computation time in milliseconds per task.



(a) Refinement calls (b) Claim calls

Figure 5: Average service call times in ms

The communication time 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.[1] The communication time averages between 150ms and 200ms per call.

The computation time for claim calls was always around 24ms. The computation time of refinement calls, however, increased with the number of executed task instances. As explained in Section 3.4, the operators in ϕ cause this time to increase exponentially.

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 on 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

[1]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.

caused by the SoD-enforcement monitor calls is an order of magnitude smaller than the runtime baseline.

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 achieved all the objectives described in Section 3.1.

5. RELATED WORK AND CONCLUSIONS

A classification of SoD constraints is given in [11]. In general, SoD mechanisms are tightly coupled with the workflow to be controlled, *e.g.* [7]. Li and Wang's SoD algebra [16] 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 [16]. Furthermore, they developed algorithms for the static enforcement of high-level SoD constraints, formalized in SoDA [23]. However, their approach is only applicable to a subset of terms.

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 [19] based on the work of Crampton [8]. Authorizations, including SoD constraints, 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.

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 organizational structures.

Concretely, our work bridges the gap between the theoretical models of [5] and a realistic implementation in an enterprise workflow environment. 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. Furthermore, the computation time during refinement calls may grow exponentially with the number of tasks that are executed. 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 service and allow task-specific constraints.

Acknowledgments: The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement N° 216917. This work is partially supported by the EU FP7-ICT-2009.1.4 Project N° 256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

6. REFERENCES

[1] Sarbanes-Oxley Act of 2002. Public Law 107-204 (116 Statute 745), United States, 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 ESORICS*, pp. 250–267, 2009.

[6] S. J. Burri, and G. Karjoth, and D. A. Basin. Separation of duties as a service. IBM Research – Zurich, TR RZ 3784 (`http://bit.ly/gAbQUy`), 2010.

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

[8] J. Crampton. A reference monitor for workflow systems with constrained task execution. *Proc. of SACMAT*, pp. 38–47, 2005.

[9] European fraud survey 2009. Ernest & Young, TR, 2009.

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

[11] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. On the formal definition of separation-of-duty policies and their composition. *Proc. of S&P*, pp. 172–183, 1998.

[12] J. Huschens and M. Rumpold-Preining, M. Bernus. IBM Insurance Application Architecture (IAA) – An overview of the Insurance Business Architecture. *Handbook on Architectures of Information Systems*, pp. 669–692, 2006.

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

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

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

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

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

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

[19] 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.

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

[21] Apache Axis2, v1.5. `http://ws.apache.org/axis2`, 2009.

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

[23] Q. Wang and N. Li. Direct static enforcement of high-level security policies. *Proc. of ASIACCS*, pp. 214–225, 2007.