Dynamic Enforcement of Abstract Separation of Duty Constraints[⋆]

David Basin[1], Samuel J. Burri[1,2], and Günter Karjoth[2]

[1] ETH Zurich, Department of Computer Science, Switzerland
[2] IBM Research, Zurich Research Laboratory, Switzerland

Abstract. Separation of Duties (SoD) aims to prevent fraud and errors by distributing tasks and associated privileges among multiple users. Li and Wang proposed an algebra (SoDA) for specifying SoD requirements, which is both expressive in the requirements it formalizes and abstract in that it is not bound to any specific workflow model. In this paper, we both generalize SoDA and map it to enforcement mechanisms. First, we increase SoDA's expressiveness by extending its semantics to multisets. This better suits policy enforcement over workflows, where users may execute multiple tasks. Second, we further generalize SoDA to allow for changing role assignments. This lifts the strong restriction that authorizations do not change during workflow execution. Finally, we map SoDA terms to CSP processes, taking advantage of CSP's operational semantics to provide the critical link between abstract specifications of SoD requirements by SoDA terms and runtime-enforcement mechanisms.

1 Introduction

Most information-security mechanisms protect resources from external threats. However, threats often reside within organizations where authorized users may intentionally or accidentally misuse information systems. Examples are the scandals [1] that led to regulations such as the Sarbanes-Oxley Act [2]. These regulations require companies to document their processes, to identify conflicts of interests, to adopt countermeasures, and to audit and control those activities. *Separation of Duties (SoD)* is a well-established extension of access control that aims to ensure data integrity, in particular the prevention of fraud and errors [3,4]. The main idea behind SoD is to split critical processes into multiple actions and to ensure that no single user can execute all actions. Therefore, at least two users must be involved in the process and fraud requires their collusion.

Existing specification formalisms and enforcement mechanisms for SoD are limited in the kinds of constraints they can handle. Moreover, they are typically bound to specific workflow models. The SoD algebra (SoDA) of Li and Wang [5] constitutes a notable exception. It allows the modeling of SoD constraints at

[⋆] 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.

a high level of abstraction, combining quantification and qualification requirements. As an example, consider the SoD policy that requires a user other than `Bob` that acts in the role of a `Manager` and one or two additional users, acting as `Accountant` and `Clerk`. Using SoDA, this policy can be modeled by the term

$$(\texttt{Manager} \sqcap \neg\{\texttt{Bob}\}) \otimes (\texttt{Accountant} \odot \texttt{Clerk}) .$$

The term's left side is satisfied by any `Manager` other than `Bob`. Under the semantics of the $\odot$-operator, the right side is satisfied by a single user that acts as `Accountant` and `Clerk` or by two users, provided one of them acts as `Accountant` and the other as `Clerk`. Finally, the $\otimes$-operator requires that the users in the two parts are disjoint. It thereby separates their duties. As this example shows, SoDA terms specify both the number and kinds of users who must take part in the workflow, independent of the details of the workflow itself. Separating concerns this way allows business processes and security requirements to be developed independently. Moreover, it permits the definition and enforcement of SoD constraints on running business processes without changing the processes' description or deployment.

Until now, no general mapping from SoDA terms onto workflows or to dynamic enforcement mechanisms existed. In particular, a link between the satisfaction of subterms and the actions executed in workflows was missing. Moreover, previous work did not address how changing role assignments affect the enforcement of SoD constraints during workflow execution. We provide solutions to these problems in this paper. Using the process algebra CSP, we construct formal models of workflows, access-control enforcement, and SoD constraints, as well as their combination.

We extend the original SoDA semantics [5] to multisets of users and interpret SoDA terms over workflow traces, allowing for changing role assignments (or, equivalently, sessions). The resulting semantics is well-suited for policy enforcement over workflows, where users may execute multiple tasks and authorizations may change during workflow execution. We further bridge the gap between the specification of high-level SoD constraints and their enforcement in a workflow environment by defining a mapping from SoDA terms to CSP processes. A correctness proof for this mapping establishes that every execution accepted by an SoD-enforcement process complies with its corresponding SoD policy.

2 Background

CSP. We briefly describe CSP [7,8] and the notation used in this paper. Let Σ be a set of *events*. Events can be structured using *channels*. Given a channel c and a set A, we can define c to be *of type* A. This means that for all $a \in A$, events of the form $c.a$ belong to Σ and represent the communication of a on the channel c. By $\{| c |\}$, we denote the set of all possible events involving channel c, i.e., $\{| c |\} := \{c.a \mid a \in A\}$. For a tuple $(a_1, ..., a_n)$, we write $c.a_1. \dots .a_n$.

Let $\mathcal{I}$ be the set of *process identifiers* and $i \in \mathcal{I}$. The set of *processes* $\mathcal{P}$ is inductively defined by the grammar $\mathcal{P} ::= e \to \mathcal{P} \mid STOP \mid i \mid \mathcal{P} \mathbin{\Box} \mathcal{P} \mid \mathcal{P} \underset{E}{\parallel} \mathcal{P}$, where $e \in \Sigma$ and $E \subseteq \Sigma$. Let $P, Q \in \mathcal{P}$ be two processes. The *assignment* of P to i is denoted by $i = P$ and can be *parametrized*. For example $i(v) = P$ defines a process parametrized by the variable v.

The process $e \rightarrow P$ *engages* in the event e first and behaves like the process P afterward. When using channels, this notation can be extended. For $A' \subseteq A$, the expression $c?a : A' \rightarrow P$ represents a process that waits for an $a \in A'$ to be *received* on channel c of type A and afterwards behaves like P. Similarly, $c!a \rightarrow P$ represents a process that *sends* a on channel c and afterwards behaves like P. $STOP$ represents the process that does not engage in any further events. For an assignment $i = P$, the process i behaves like P. $P \,\Box\, Q$ denotes a process that lets the environment choose whether it behaves like P or Q.The process $P \underset{E}{\parallel} Q$ represents the parallel execution of the processes P and Q *synchronized* on $E \subseteq \Sigma$. This means, whenever one of the two processes engages in an event $e \in E$, the other process must also engage in e.

A *trace*, denoted $\langle e_1, ..., e_n \rangle$, is a sequence of events. $\langle\rangle$ denotes the *empty* trace and $t\hat{\ }t'$ denotes the *concatenation* of two finite traces t and t'. Moreover, E^* denotes the set of all finite traces over E and E^+ denotes the set of all finite traces over E that contain at least one event. A process is described as a set $\mathcal{T}(P) \subseteq \Sigma^*$ of finite traces. When $t \in \mathcal{T}(P)$, P *accepts* t; each such trace t describes a sequence of events that P can engage in with the environment. For example, $\mathcal{T}(STOP) := \{\langle\rangle\}$, $\mathcal{T}(e \rightarrow P) := \{\langle\rangle\} \cup \{\langle e \rangle\hat{\ }t \mid t \in \mathcal{T}(P)\}$, and $\mathcal{T}(P \,\Box\, Q) := \mathcal{T}(P) \cup \mathcal{T}(Q)$. Q *refines* P, denoted $P \sqsubseteq_{\mathcal{T}} Q$, if and only if $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$.

Multisets. We will make extensive use of multisets in the paper and briefly review their notation. A *multiset*, or *bag*, is a collection of objects where repetition is allowed [9]. Formally, given a set A, a multiset $\mathbf{M}$ of A is a pair (A, f), where the function $f : A \rightarrow \mathbb{N}_0$ (where $\mathbb{N}_0$ is the set of natural numbers, including zero) defines how often each element $a \in A$ occurs in $\mathbf{M}$. We write $\mathbf{M}(a)$ as shorthand for $f(a)$. We say that a is an *element* of $\mathbf{M}$, written $a \in \mathbf{M}$, if $\mathbf{M}(a) \geq 1$. We use standard set notation to define multisets, but allow duplicated elements, e.g., $\mathbf{M} := \{a_1, a_1\}$ is the multiset where $\mathbf{M}(a_1) = 2$ and for all other $a \in A$, $\mathbf{M}(a) = 0$. For a finite multiset $\mathbf{M}$, $|\mathbf{M}|$ denotes the *cardinality* of $\mathbf{M}$ and is defined as $\sum_{a \in A} \mathbf{M}(a)$. Given the multisets $\mathbf{M}$ and $\mathbf{N}$, their *intersection*, denoted $\mathbf{M} \cap \mathbf{N}$, is the multiset $\mathbf{O}$, where for all $a \in A$, $\mathbf{O}(a) := \min(\mathbf{M}(a), \mathbf{N}(a))$. Similarly, their *union*, denoted $\mathbf{M} \cup \mathbf{N}$, is the multiset $\mathbf{O}$, where for all $a \in A$, $\mathbf{O}(a) := \max(\mathbf{M}(a), \mathbf{N}(a))$, and their *sum*, denoted $\mathbf{M} \uplus \mathbf{N}$, is the multiset $\mathbf{O}$, where for all $a \in A$, $\mathbf{O}(a) := \mathbf{M}(a) + \mathbf{N}(a)$. The *empty multiset* $\emptyset$ of A is the multiset where $\emptyset(a) := 0$, for all $a \in A$.

3 Secure Workflow Processes

3.1 Modeling Workflows

We call a unit of work an *action*. The temporal ordering of actions and the causal dependencies between them, which together implement a business objective, are called a *workflow*. There are various formalisms for modeling workflows. We use CSP.

For the rest of this paper, let $\mathcal{U}$ be a set of *users* and $\mathcal{A}$ a set of *actions*. We model a workflow as a CSP process with a channel bc of type $\mathcal{U} \times \mathcal{A}$ that we call the *business channel*. Let $\mathcal{E}_B := \{|bc|\}$, and we call an element of $\mathcal{E}_B$ a *business event*. For a user u and an action a, the business event $bc.u.a$ describes the execution of the action a by the user u.

We introduce the event *done*, which states that a workflow has finished.[1] We further define the auxiliary predicate done on traces where, for all $t \in \Sigma^*$, $\mathsf{done}(t)$ if and only if t contains exactly one event *done* in the end. Formally, $\mathsf{done}(t) := \exists t' \in (\Sigma \setminus \{done\})^* \,.\, t = t'^\frown\langle done\rangle$.

For a workflow w modeled by a process W, a trace $t \in \mathcal{T}(W)$ corresponds to a *workflow run* (or *workflow instance*) of w. A trace t represents a *finished* workflow run if $\mathsf{done}(t)$; otherwise t represents an *unfinished* workflow run. Note that given a trace t and a process W, it is straightforward to check, using CSP's operational semantics, whether $t \in \mathcal{T}(W)$.

For a process W that models a workflow, we require the set of traces $\mathcal{T}(W)$ to contain at least one trace that corresponds to a finished workflow run. This ensures that each workflow can be completed in at least one way.

We define two auxiliary functions that extract users from traces. First, the projection function $\mathsf{user} : \mathcal{E}_B \to \mathcal{U}$, given a business event $business.u.a$, returns u. Second, the function users, given a trace t, returns the multiset of users that are contained in business events in t.

$$\mathsf{users}(t) := \begin{cases} \emptyset & \text{if } t = \langle\rangle, \\ \{\mathsf{user}(b)\} \uplus \mathsf{users}(t') & \text{for } t = \langle b\rangle^\frown t' \text{and } b \in \mathcal{E}_B, \\ \mathsf{users}(t') & \text{for } t = \langle e\rangle^\frown t' \text{and } e \notin \mathcal{E}_B. \end{cases}$$

To illustrate these notions, we introduce a running example of a payment process, similar to the one used in [4].

Example 1 (Payment workflow). Fig. 1 describes a payment workflow where invoices are payed by check. For now, all users can execute all actions. Only in later refinements do we restrict the set of authorized users. First, an invoice is received and afterwards a payment check is prepared. Next, the payment is either directly approved, it is approved but at least one further approval is required, or it is rejected. In the third case, the payment must be prepared again. If the payment is finally approved, the check is issued and the workflow terminates, which is denoted by the event *done*. Fig. 1a models the workflow as a process W and Fig. 1b depicts the workflow as a labeled transition system. The edge $s_1 \xrightarrow{\{l_1,\ldots,l_n\}} s'$ denotes the set of labeled transitions $s \xrightarrow{l_i} s'$, for $i \in \{1, \ldots, n\}$.

[1] We do not use CSP's special event ✓ and the process *SKIP* because later we synchronize on *done* with most, but not all, involved processes. By the semantics of CSP, all processes must synchronize on ✓.

$$\begin{aligned}
W &= W_1\\
W_1 &= bc?u : \mathcal{U}.\texttt{receive invoice} \rightarrow W_2\\
W_2 &= bc?u : \mathcal{U}.\texttt{prepare check} \rightarrow W_3\\
W_3 &= (bc?u : \mathcal{U}.\texttt{reject payment} \rightarrow W_2)\\
&\quad \Box\ (bc?u : \mathcal{U}.\texttt{approve payment} \rightarrow W_3)\\
&\quad \Box\ (bc?u : \mathcal{U}.\texttt{approve payment} \rightarrow W_4)\\
W_4 &= bc?u : \mathcal{U}.\texttt{issue check} \rightarrow W_5\\
W_5 &= done \rightarrow STOP
\end{aligned}$$

a) In CSP notation

$$\begin{tikzcd}
{} \arrow[r] & \textcircled{1} \arrow[r, "RI"] & \textcircled{2} \arrow[r, "PC"] & \textcircled{3} \arrow[r, "AP"] \arrow[l, bend left, "RP"] \arrow[loop below, "AP"] & \textcircled{4} \arrow[r, "IC"] & \textcircled{5} \arrow[r, "done"] & \textcircled{6}
\end{tikzcd}$$

$$\begin{aligned}
RI &:= \{bc.u.\texttt{receive invoice} \mid u \in \mathcal{U}\}\\
PC &:= \{bc.u.\texttt{prepare check} \mid u \in \mathcal{U}\}\\
AP &:= \{bc.u.\texttt{approve payment} \mid u \in \mathcal{U}\}\\
RP &:= \{bc.u.\texttt{reject payment} \mid u \in \mathcal{U}\}\\
IC &:= \{bc.u.\texttt{issue check} \mid u \in \mathcal{U}\}
\end{aligned}$$

b) As labeled transition system

Fig. 1. Payment Workflow

3.2 Access Control

We use *role-based access control (RBAC)* [10,6] to describe access-control policies. We only make use of RBAC's core feature, which is the decomposition of the user-permission-assignment relation into a user-role and a role-permission-assignment relation. For the reminder of this paper, let $\mathcal{R}$ be a set of *roles*.

Definition 1 (RBAC configuration). *An* RBAC configuration *is a tuple* (UA, PA)*, where* $UA \subseteq \mathcal{U} \times \mathcal{R}$ *is the* user-assignment relation *and* $PA \subseteq \mathcal{R} \times \mathcal{A}$ *is the* permission-assignment relation*.*

We say that the user u *acts in the role* r if $(u, r) \in UA$. Furthermore, the user u is *authorized to execute the action* a if $\exists r \in \mathcal{R}\,.\ (u, r) \in UA$ and $(r, a) \in PA$.

In contrast to the RBAC standard of NIST [6], we omit the concept of sessions. This is without loss of generality as the activation and deactivation of roles within a session can be modeled by changing RBAC configurations, where all assigned roles are always implicitly activated. Note that what we call actions are called *permissions* in [6].

Administrative actions $\mathcal{A}_A \subseteq \mathcal{A}$ are the subset of actions that modify RBAC configurations. For a user u, a role r, and a user-assignment relation UA, the action $\texttt{addUA}.u.r$ adds the tuple (u, r) to UA and the action $\texttt{rmUA}.u.r$ removes (u, r) from UA. In this paper, we do not discuss administrative actions that change permission-assignment relations. We describe a configuration's evolution and the enforcement of the resulting access-control policy in terms of a process that we call the *RBAC process.*

$$\begin{aligned}
RBAC(UA, PA) = {}& \big(bc?(u.a) : \{u.a \mid \exists r \in \mathcal{R}\,.\,(u, r) \in UA \wedge (r, a) \in PA\} \rightarrow RBAC(UA, PA)\big)\\
&\Box\ \big(ac.\texttt{addUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow RBAC(UA \cup \{(u, r)\}, PA)\big)\\
&\Box\ \big(ac.\texttt{rmUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow RBAC(UA \setminus \{(u, r)\}, PA)\big)
\end{aligned}$$

The RBAC process is parametrized by a user-assignment relation UA and a permission-assignment relation PA, which together represent an RBAC configuration. Besides the channel bc, introduced in Sec. 3.1, the RBAC process also has a channel called ac of type $\mathcal{A}_A$ that we call the *admin channel.* Let $\mathcal{E}_A := \{| ac |\}$, and we call an element of $\mathcal{E}_A$ an *admin event.* Note that the RBAC process does not terminate, i.e., it never behaves like $STOP$. This is consistent with our view of access-control monitors that outlive workflow execution.

Given a process W that models a workflow, we define the *secure (workflow) process SW* as the parallel composition of W and $RBAC$, synchronized on all business events. Like the RBAC process, a secure process is parametrized by an RBAC configuration.

$$SW(UA, PA) = W \underset{\mathcal{E}_B}{\|} RBAC(UA, PA)$$

A secure process models a workflow that only executes actions authorized under the configuration. By synchronizing only on business events, arbitrary admin events can be interleaved with business events and *done* in any order. Thus, the RBAC configuration can change between workflow actions. Having introduced all the kinds of events that we need, specifically, $\Sigma = \mathcal{E}_B \cup \mathcal{E}_A \cup \{done\}$, we now refine the workflow from Example 1 into a secure workflow process.

Example 2 (Secure workflow process). Assume $\mathcal{U} := \{$`Alice`,`Bob`,`Claire`$\}$, $\mathcal{R} := \{$`Accountant`, `Clerk`, `Manager`$\}$, and $\mathcal{A} := \{$`receive invoice`, `issue check`, `prepare check`, `approve payment`, `reject payment`$\}$. Also, let the RBAC configuration (UA, PA) be initially given as depicted by the solid arrows in Fig. 2.

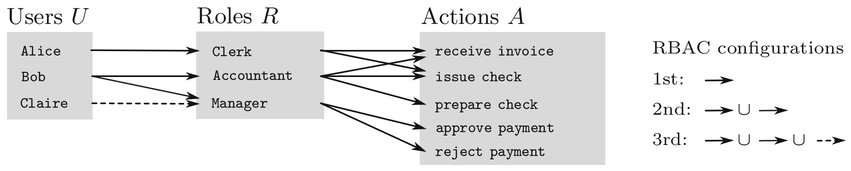

Fig. 2. Example RBAC Configuration

Consider the following trace, corresponding to a completed workflow run.

$$t := \langle bc.\texttt{Alice.receive invoice},\ bc.\texttt{Bob.prepare check},\ bc.\texttt{Bob.approve payment},\ bc.\texttt{Alice.issue check},\ done\rangle$$

This trace represents a workflow run of our payment workflow, modeled by W. In contrast, $t \notin \mathcal{T}(SW(UA, PA))$ because no user is authorized to execute `approve payment`. This can be overcome by placing `Bob` in the `Manager` role.

$$t' := \langle bc.\texttt{Alice.receive invoice},\ bc.\texttt{Bob.prepare check},\ ac.\texttt{addUA.Bob.Manager},\ bc.\texttt{Bob.approve payment},\ bc.\texttt{Alice.issue check},\ done\rangle$$

The new admin event adds the user-role assignment (`Bob`, `Manager`) to SW's RBAC configuration as indicated by the dotted arrow in Fig. 2. Therefore,

$t' \in \mathcal{T}(SW(UA, PA))$. However, it is risky to allow `Bob` to execute both the actions `prepare check` and `approve payment` as he could then approve his own fraudulent payments. Our next refinement of this example solves this problem by enforcing an appropriate SoD constraint.

4 Abstract Separation of Duty Constraints

4.1 Separation of Duty Algebra Syntax

Our work builds on Li and Wang's *separation of duty algebra* [5], *SoDA*. We present below the syntax of SoDA terms.

Definition 2 (SoDA grammar $\mathfrak{G}$). *A* SoDA grammar $\mathfrak{G}$ *with respect to a set of users* $\mathcal{U} := \{u_1, \ldots, u_n\}$ *and a set of roles* $\mathcal{R} := \{r_1, \ldots, r_m\}$ *is a quadruple* (N, T, P, S) *where:*

- $N := \{S, CT, UT, AT, US, UR, U, R\}$ *is the set of nonterminal symbols,*
- $T := \{',', (,), \{, \}, \otimes, \odot, \sqcup, \sqcap, ^+, \neg, \mathsf{All}\} \cup \mathcal{U} \cup \mathcal{R}$ *are the terminal symbols,*
- *the set of productions* $P \subseteq (\, N \times (N \cup T)^* \,)$ *is given by:*

$$\begin{array}{llllll} S & ::= & CT \mid UT & CT & ::= & (CT \sqcup S) \mid (CT \sqcap S) \mid (S \otimes S) \mid (S \odot S) \mid (UT)^+ \\ AT & ::= & \{UR\} \mid R \mid \mathsf{All} & UT & ::= & AT \mid (UT \sqcap UT) \mid (UT \sqcup UT) \mid \neg\, UT \\ UR & ::= & U \mid U,\, UR & U & ::= & u_1 \mid \ldots \mid u_n \\ R & ::= & r_1 \mid \ldots \mid r_m & & & \end{array}$$

- *and* $S \in N$ *is the start symbol.*

The terminal symbols $\otimes$, $\odot$, $\sqcup$, $\sqcap$, $^+$, and $\neg$ are called *operators*. Without loss of generality, we omit the productions $CT ::= (S \sqcap CT)$ and $CT ::= (S \sqcup CT)$. Li and Wang showed in [5] that $\sqcap$ and $\sqcup$ are commutative with respect to their semantics and this is also the case for our semantics. Therefore, each term that could be constructed with these additional productions can be transformed to a semantically equivalent term constructed without them.

Let $\rightarrow^1_{\mathfrak{G}} \in (N \cup T)^+ \times (N \cup T)^*$ denote one derivation step of $\mathfrak{G}$ and $\rightarrow^*_{\mathfrak{G}}$ the transitive closure of $\rightarrow^1_{\mathfrak{G}}$. We call an element of $\{s \in T^* \mid S \rightarrow^*_{\mathfrak{G}} s\}$ a *term*. Furthermore, we call an element of $\{s \in T^* \mid AT \rightarrow^*_{\mathfrak{G}} s\}$ an *atomic term*. These are either a non-empty set of users, e.g. $\{$`Alice`,`Bob`$\}$, a single role, e.g. `Clerk`, or the keyword All. We call an element of $\{s \in T^* \mid UT \rightarrow^*_{\mathfrak{G}} s\}$ a *unit term*. These terms do not contain the operators $\otimes$, $\odot$, and $^+$. Finally, a *complex term* is an element of $\{s \in T^* \mid CT \rightarrow^*_{\mathfrak{G}} s\}$. In contrast to unit terms, they contain at least one of the operators $\otimes$, $\odot$, or $^+$. For a term ϕ, we call a unit term ϕ_{ut} a *maximal unit term of* ϕ if ϕ_{ut} is a subterm of ϕ and if there is no other unit term ϕ'_{ut} that is also a subterm of ϕ, where ϕ_{ut} is a subterm of ϕ'_{ut}.

4.2 SoDA Semantics for Multisets of Users

Li and Wang define the satisfaction of SoDA terms for sets of users [5]. We refer to their semantics as $\text{SoDA}^{\mathcal{S}}$, which allows for quantitative constraints whereby terms define how many different users must participate in a workflow. However, it does not express how many actions each of these users must execute. Consider the policy P that requires `Bob` to execute two actions, modeled by the SoDA term $\phi := \{\texttt{Bob}\} \odot \{\texttt{Bob}\}$. Under $\text{SoDA}^{\mathcal{S}}$, ϕ is satisfied by the set $\{\texttt{Bob}\}$. There is no satisfactory mapping of ϕ to a process that accepts all traces that correspond to satisfying assignments of ϕ. If we define the correspondence between sets and traces in a way that $\{\texttt{Bob}\}$ maps to the set of traces containing *exactly one* business event executed by `Bob`, this would not satisfy P. Alternatively, if we map $\{\texttt{Bob}\}$ to the set of traces containing *arbitrarily many* business events executed by `Bob`, this set would also include traces that do not satisfy P, for example, the trace containing three business events executed by `Bob`. The problem here is that sets of users are too restrictive: users cannot be repeated and hence information is lost on how many actions a user (here `Bob`) must perform.

To address this problem, we introduce a new semantics, $\text{SoDA}^{\mathcal{M}}$, that defines term satisfaction based on multisets of users. This allows us to make finer distinctions concerning repetition (quantification requirements) than in $\text{SoDA}^{\mathcal{S}}$. As shown below, under $\text{SoDA}^{\mathcal{M}}$, ϕ is only satisfied by the multiset $\{\texttt{Bob}, \texttt{Bob}\}$. Mapping multisets to traces is straightforward and the corresponding traces include exactly two business events that are executed by `Bob`. In this respect, $\text{SoDA}^{\mathcal{M}}$ allows a more precise mapping to traces than $\text{SoDA}^{\mathcal{S}}$.

Definition 3 (Multiset Satisfaction $\text{SoDA}^{\mathcal{M}}$). *Let $U \subseteq \mathcal{U}$ be a non-empty set of users and $r \in \mathcal{R}$ a role. For a multiset of users $\mathbf{U}$, a term ϕ, and a user-assignment relation UA,* multiset satisfiability *is the smallest ternary relation between multisets of users, user-assignment relations, and terms, written $\mathbf{U} \models^{\mathcal{M}}_{UA} \phi$, that is closed under the following rules:*

$$(1)\ \frac{}{\{u\} \models^{\mathcal{M}}_{UA} \mathsf{All}}\ \exists r \in \mathcal{R} \,.\, (u,r) \in UA \qquad (2)\ \frac{}{\{u\} \models^{\mathcal{M}}_{UA} r}\ (u,r) \in UA$$

$$(3)\ \frac{}{\{u\} \models^{\mathcal{M}}_{UA} U}\ u \in U \textit{ and } \exists r \in \mathcal{R} \,.\, (u,r) \in UA \qquad (4)\ \frac{\{u\} \not\models^{\mathcal{M}}_{UA} \phi}{\{u\} \models^{\mathcal{M}}_{UA} \neg\phi}$$

$$(5)\ \frac{\{u\} \models^{\mathcal{M}}_{UA} \phi}{\{u\} \models^{\mathcal{M}}_{UA} \phi^{+}} \qquad (6)\ \frac{\{u\} \models^{\mathcal{M}}_{UA} \phi,\ \mathbf{U} \models^{\mathcal{M}}_{UA} \phi^{+}}{(\{u\} \uplus \mathbf{U}) \models^{\mathcal{M}}_{UA} \phi^{+}}$$

$$(7)\ \frac{\mathbf{U} \models^{\mathcal{M}}_{UA} \phi}{\mathbf{U} \models^{\mathcal{M}}_{UA} (\phi \sqcup \psi)} \qquad (8)\ \frac{\mathbf{U} \models^{\mathcal{M}}_{UA} \psi}{\mathbf{U} \models^{\mathcal{M}}_{UA} (\phi \sqcup \psi)}$$

$$(9)\ \frac{\mathbf{U} \models^{\mathcal{M}}_{UA} \phi,\ \mathbf{U} \models^{\mathcal{M}}_{UA} \psi}{\mathbf{U} \models^{\mathcal{M}}_{UA} (\phi \sqcap \psi)} \qquad (10)\ \frac{\mathbf{U} \models^{\mathcal{M}}_{UA} \phi,\ \mathbf{V} \models^{\mathcal{M}}_{UA} \psi}{(\mathbf{U} \uplus \mathbf{V}) \models^{\mathcal{M}}_{UA} (\phi \odot \psi)}$$

$$(11)\ \frac{\mathbf{U} \models^{\mathcal{M}}_{UA} \phi,\ \mathbf{V} \models^{\mathcal{M}}_{UA} \psi}{(\mathbf{U} \uplus \mathbf{V}) \models^{\mathcal{M}}_{UA} (\phi \otimes \psi)}\ (\mathbf{U} \cap \mathbf{V}) = \emptyset \,.$$

We say that $\mathbf{U}$ *satisfies* ϕ *with respect to* UA if $\mathbf{U} \models^{\mathcal{M}}_{UA} \phi$. Informally, a user u satisfies the term All if u is in the domain of UA. A user u satisfies a role r if there is a role assignment (u, r) in UA, and u satisfies a set of users U if u is member of U and is in the domain of UA. A unit term $\neg\phi$ is satisfied by u if u does not satisfy ϕ. A non-empty multiset of users $\mathbf{U}$ satisfies a complex term ϕ^+ if each user $u \in \mathbf{U}$ satisfies the unit term ϕ. A multiset of users $\mathbf{U}$ satisfies a term $\phi \sqcup \psi$ if $\mathbf{U}$ satisfies either ϕ or ψ, and $\mathbf{U}$ satisfies a term $\phi \sqcap \psi$ if $\mathbf{U}$ satisfies both ϕ and ψ. A term $\phi \otimes \psi$ is satisfied by a multiset of users $\mathbf{W}$, if $\mathbf{W}$ can be partitioned into two disjoint multisets $\mathbf{U}$ and $\mathbf{V}$, and $\mathbf{U}$ satisfies ϕ and $\mathbf{V}$ satisfies ψ. Because every user in $\mathbf{W}$ must be in either $\mathbf{U}$ or $\mathbf{V}$, but not both, the $\otimes$ operator separates duties between two multisets of users. In contrast, a term $\phi \odot \psi$ is satisfied by a multiset of users $\mathbf{W}$, if there are two multisets $\mathbf{U}$ and $\mathbf{V}$, which may share users, and $\mathbf{U}$ satisfies ϕ, $\mathbf{V}$ satisfies ψ, and $\mathbf{W}$ is the sum of $\mathbf{U}$ and $\mathbf{V}$. Thus, the $\odot$ operator allows overlapping duties where a user is in both $\mathbf{U}$ and $\mathbf{V}$.

We now provide two examples. The first illustrates many of the operators whereas the second illustrates the difference between SoDA$^{\mathcal{M}}$ and SoDA$^{\mathcal{S}}$.

Example 3. Suppose we have the term $\phi = (\texttt{Accountant} \otimes (\texttt{Manager} \sqcup (\texttt{Accountant} \otimes \texttt{Accountant}))) \odot \mathsf{All}^+$ and the third user-assignment relation shown in Fig. 2,

$$UA'' := \{(\texttt{Alice}, \texttt{Clerk}), (\texttt{Bob}, \texttt{Accountant}), (\texttt{Bob}, \texttt{Manager}), (\texttt{Claire}, \texttt{Manager})\}.$$

It follows that $\{\texttt{Alice}, \texttt{Alice}, \texttt{Bob}, \texttt{Claire}\}$ satisfies ϕ with respect to UA''. In contrast, $\{\texttt{Alice}, \texttt{Claire}\}$ does not satisfy ϕ with respect to UA'', because ϕ least one `Accountant`. Moreover, $\{\texttt{Alice}, \texttt{Bob}\}$ does not satisfy ϕ either, because ϕ requires also a `Manager` or a second user who acts as `Accountant`.

Example 4. Under SoDA$^{\mathcal{M}}$, the term $\{\texttt{Bob}\} \odot \{\texttt{Bob}\} \odot \{\texttt{Bob}\}^+$ is satisfied by all multisets that contain `Bob` three or more times, i.e. `Bob` must execute at least three actions. Under SoDA$^{\mathcal{S}}$, this term is only satisfied by the set $\{\texttt{Bob}\}$ and therefore does not define how many actions `Bob` must actually execute.

We conclude by relating SoDA$^{\mathcal{M}}$ and SoDA$^{\mathcal{S}}$. Under SoDA$^{\mathcal{S}}$, $X \models^{\mathcal{S}}_{(U,UR)} \phi$ denotes the satisfaction of a term ϕ by a set of users X with respect to a tuple (U, UR), where $U \subseteq \mathcal{U}$ and $UR \subseteq U \times \mathcal{R}$. Because actions can only be executed by users who have at least one role assignment, we simplify this tuple and extract the available users from UA, as one can see in Rule (3) of Def. 3. For a user-assignment relation UA, the function $\mathsf{lwconf}(UA) := (\{u \in \mathcal{U} \mid \exists r \in \mathcal{R} \,.\, (u, r) \in UA\}, UA)$ maps UA to the corresponding tuple in SoDA$^{\mathcal{S}}$. Moreover, given a multiset of users $\mathbf{U}$, the function $\mathsf{userset}(\mathbf{U}) := \{u \mid u \in \mathbf{U}\}$ returns the set of users contained in $\mathbf{U}$. We prove the following lemma in [11], showing that SoDA$^{\mathcal{M}}$ generalizes SoDA$^{\mathcal{S}}$ in the following sense.

Lemma 1. *For all terms ϕ, all user-assignment relations UA, and all multisets of users $\mathbf{U}$, if $\mathbf{U} \models^{\mathcal{M}}_{UA} \phi$, then $\mathsf{userset}(\mathbf{U}) \models^{\mathcal{S}}_{\mathsf{lwconf}(UA)} \phi$.*

5 Separation of Duty Enforcement

5.1 Approach and Requirements

As shown above, SoDA specifies SoD constraints at a high level of abstraction. However, the enforcement takes place at runtime in the context of a workflow run. Given a term ϕ, we now describe how to construct an enforcement monitor for ϕ. Our construction maps ϕ to a process $SOD_\phi(UA)$, called the *SoD-enforcement process*, parametrized by a user-assignment relation UA. $SOD_\phi(UA)$ accepts all traces corresponding to a multiset that satisfies ϕ with respect to UA.

In practice, it is critical to allow administrative events during workflow execution. If `Bob` leaves his company, it should be possible to remove all his role assignments, thereby preventing him from subsequently executing actions in currently executing workflow runs. Similarly, if `Alice` joins a company or changes positions, and as a consequence is assigned new roles, she should also be able to execute actions in workflow runs that were started prior to the organizational change. Assuming that a user-assignment relation does not change during the execution of a workflow run is therefore overly restrictive. The SoD-enforcement process defined below accounts for such changes. The function upd ("update") describes how a trace of admin events changes a user-assignment relation.

Definition 4 (UA change). *Let $a \in \mathcal{E}_A^*$ be a trace of admin events and UA a user-assignment relation. The function* upd *is defined as follows:*

$$\mathsf{upd}(UA, a) := \begin{cases} UA & \text{if } a = \langle\rangle, \\ \mathsf{upd}(UA \cup \{(u, r)\}, a') & \text{if } a = (ac.\mathtt{addUA}.u.r)\hat{}\; a', \\ \mathsf{upd}(UA \setminus \{(u, r)\}, a') & \text{if } a = (ac.\mathtt{rmUA}.u.r)\hat{}\; a', \end{cases}$$

where u ranges over $\mathcal{U}$, r over $\mathcal{R}$, and a' over $\mathcal{E}_A^*$.

Let ϕ be a term, UA a user-assignment relation, and $SOD_\phi(UA)$ the SoD-enforcement process for ϕ and UA. We postulate that $SOD_\phi(UA)$ must fulfill the following administration requirements.

- **(R1)** $SOD_\phi(UA)$ must accept every trace of admin events a, and behave like $SOD_\phi(UA')$ afterwards, for $UA' := \mathsf{upd}(UA, a)$.
- **(R2)** If $SOD_\phi(UA)$ accepts a trace t containing no admin events and reaches a final state, then $\mathsf{users}(t) \models^{\mathcal{M}}_{UA} \phi$.
- **(R3)** $SOD_\phi(UA)$ must engage in a business event $bc.u.a$, if $\{u\}$ satisfies at least one maximal unit term of ϕ with respect to UA and no restriction imposed by ϕ is violated.
- **(R4)** The semantics of the operators $^+$, $\sqcup$, $\sqcap$, $\odot$, and $\otimes$ with respect to traces must agree with their definition in SoDA$^{\mathcal{M}}$.

(R1) says that administrative events are always possible and reflected in the user-assignment relation. (R2) states that in the absence of admin events, $SOD_\phi(UA)$ agrees with the SoDA$^{\mathcal{M}}$ semantics. (R3) formulates agreement with SoDA$^{\mathcal{M}}$, where for a multiset of users $\mathbf{U}$, if $\mathbf{U} \models_{UA}^{\mathcal{M}} \phi$, then each user in $\mathbf{U}$ satisfies at least one maximal unit term of ϕ with respect to UA. Similarly, $SOD_\phi(UA)$ must not engage in a business event if the corresponding user does not contribute to the satisfaction of ϕ. As for (R4), consider for example the terms $\phi \otimes \psi$ and $\phi \odot \psi$. It must be possible to partition a trace satisfying $\phi \otimes \psi$ or $\phi \odot \psi$ into two subtraces, one satisfying ϕ and the other one satisfying ψ. In the case of $\phi \otimes \psi$, the users who execute business events in one trace must be disjoint from the users executing business events in the other trace. In contrast, for $\phi \odot \psi$, the multisets of users need not be disjoint.

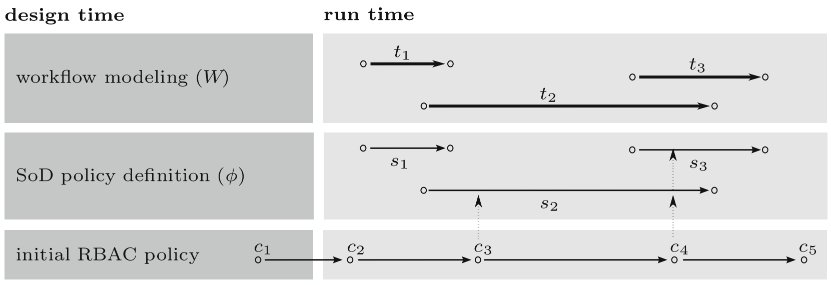

Fig. 3. Relations between a workflow process, an SoD-enforcement process, and the RBAC process

Fig. 3 illustrates how an SoD-enforcement process relates to the processes introduced so far. The X-axis represents time and the Y-axis lists a workflow process, the RBAC process, and an SoD-enforcement process. We distinguish between two time periods. At *design time*, a business officer defines a workflow using a workflow language that can be modeled as a process W, a security officer specifies the initial RBAC configuration c_1, and a compliance officer formulates SoD constraints as a term ϕ, which is mapped to the SoD-enforcement process SOD_ϕ. At *run time*, the workflow corresponding to W is executed an arbitrary number of times. Each workflow run, t_1, t_2 and t_3, corresponds to a trace of W. An instance of SOD_ϕ executes in parallel with each workflow run, e.g., s_1 in parallel with t_1. Each instance of SOD_ϕ tracks who has previously executed actions in the associated workflow run and ensures that no SoD constraint is violated. The execution of the RBAC process is modeled as a single trace. Admin events change the configuration of the RBAC process. In Fig. 3, the RBAC process evolves from c_1 to c_2, then to c_3, and so forth. Furthermore, RBAC configuration changes also affect the currently running instances of SOD_ϕ. For example, when the RBAC configuration of the process changes to c_4, this is reflected in s_2 and s_3 as indicated by the dotted arrows.

Without loss of generality, in the remainder of this paper, we look only at the execution of one instance of W, the RBAC process, and one instance of SOD_ϕ. Furthermore, we describe the traces of W, $RBAC$, and SOD_ϕ as the single trace of the partially synchronized, parallel composition of W, $RBAC$, and SOD_ϕ. The formal definition follows.

5.2 SoDA Semantics for Traces

The following example shows that $\text{SoDA}^{\mathcal{M}}$ is not expressive enough to capture the administration requirements (R1)–(R4).

Example 5. Consider the policy P that requires one action to be executed by a user acting as `Manager` and another action to be executed by a user who is not acting as `Manager`. We model P by the term $\phi := \texttt{Manager} \odot \neg\texttt{Manager}$. Under $\text{SoDA}^{\mathcal{M}}$, ϕ can only be satisfied by a multiset of users that contains two different users. Now, consider the trace

$$t := \langle ac.\texttt{addUA.Bob.Manager},\ bc.\texttt{Bob}.a,\ ac.\texttt{rmUA.Bob.Manager},\ bc.\texttt{Bob}.a' \rangle\ ,$$

for two arbitrary actions a and a'. From (R1)–(R4), it follows that $SOD_\phi(\emptyset)$ must accept t. By (R1), $SOD_\phi(\emptyset)$ engages in $ac.\texttt{addUA.Bob.Manager}$ and afterwards behaves like $SOD_\phi(UA)$, for $UA = \{(\texttt{Bob}, \texttt{Manager})\}$. Next, $SOD_\phi(UA)$ engages in $bc.\texttt{Bob}.a$ by (R3) and (R4) because `Bob` acts as `Manager`. Again by (R1), $SOD_\phi(UA)$ engages in $ac.\texttt{rmUA.Bob.Manager}$ and afterwards behaves like $SOD_\phi(\emptyset)$. Finally, by (R3) and (R4), $SOD_\phi(\emptyset)$ engages in $bc.\texttt{Bob}.a'$ because `Bob` does not act as `Manager`. In the end, SOD_ϕ engaged in a business event with a user that acted as `Manager` and in another one with a user not acting as `Manager`, satisfying the policy P. However, we have $\mathsf{users}(t) = \{\texttt{Bob}, \texttt{Bob}\}$, which contradicts the previous statement that ϕ is only satisfied by multisets containing two different users.

The inability to handle administrative changes motivates the introduction of a third semantics, $\text{SoDA}^{\mathcal{T}}$. In $\text{SoDA}^{\mathcal{T}}$, subterms correspond to separate traces that may interleave with each other in any order. Admin events, though, must occur in all traces in the same order. This reflects that SoDA terms do not constrain the order of executed actions but that the user-assignment relation must be consistent across all subterms at any time. We formalize this relation by the *synchronized interleaving* predicate si. For traces t, t_1, and t_2, $\mathsf{si}(t, t_1, t_2)$ holds if and only if t_1 and t_2 "partition" t such that each admin event in t is contained in both t_1 and t_2, and each business event is either in one of t_1 or t_2. More formally:

Definition 5 (Synchronized interleaving). *Let* $t, t_1, t_2 \in (\mathcal{E}_B \cup \mathcal{E}_A)^*$ *be traces. The* synchronized interleaving *predicate* $\mathsf{si}(t, t_1, t_2)$ *is defined as follows:*

$$\mathsf{si}(t, t_1, t_2) := \begin{cases} \mathit{true} & \mathit{if}\, t = \langle\rangle, t_1 = \langle\rangle \;\mathit{and}\, t_2 = \langle\rangle, \\ \mathsf{si}(t', t_1', t_2') & \mathit{if}\, t = \langle a\rangle \hat{}\, t', t_1 = \langle a\rangle \hat{}\, t_1', \mathit{and}\, t_2 = \langle a\rangle \hat{}\, t_2', \\ \mathsf{si}(t', t_1', t_2) \;\mathit{or}\; \mathsf{si}(t', t_1, t_2') & \mathit{if}\, t = \langle b\rangle \hat{}\, t', t_1 = \langle b\rangle \hat{}\, t_1', \mathit{and}\, t_2 = \langle b\rangle \hat{}\, t_2', \\ \mathsf{si}(t', t_1', t_2) & \mathit{if}\, t = \langle b\rangle \hat{}\, t', t_1 = \langle b\rangle \hat{}\, t_1', \mathit{and}\, t_2 \neq \langle b\rangle \hat{}\, t_2', \\ \mathsf{si}(t', t_1, t_2') & \mathit{if}\, t = \langle b\rangle \hat{}\, t', t_1 \neq \langle b\rangle \hat{}\, t_1', \mathit{and}\, t_2 = \langle b\rangle \hat{}\, t_2', \\ \mathit{false} & \mathit{otherwise}, \end{cases}$$

where a *ranges over* $\mathcal{E}_A$*,* b *over* $\mathcal{E}_B$*, and* t'*,* t_1'*, and* t_2' *over* $(\mathcal{E}_B \cup \mathcal{E}_A)^*$*.*

Note that the *or* in the third case arises as there are two possible interleavings. The predicate si will hold (evaluate to *true*) if either of the two interleavings hold. We illustrate si with an example.

$$\begin{array}{rl} t & := \langle\, b_1, b_2, b_3, a_1, b_4, b_4, a_2, b_5, a_3, b_6, a_4 \,\rangle \\ t_1 & := \langle\, b_1, \quad b_3, a_1, b_4, \quad a_2, \quad a_3, b_6, a_4 \,\rangle \\ t_2 & := \langle \quad b_2, \quad a_1, \quad b_4, a_2, b_5, a_3, \quad a_4 \,\rangle \end{array}$$

For these three traces, $\mathsf{si}(t, t_1, t_2)$ holds.

We now define the satisfaction of SoDA terms by traces.

Definition 6 (Trace Satisfaction SoDA$^{\mathcal{T}}$). *Let $a \in \mathcal{E}_A$ be an admin event and $b \in \mathcal{E}_B$ a business event. For a trace $t \in (\mathcal{E}_A \cup \mathcal{E}_B)^*$, a user-assignment relation UA, a term ϕ, and a unit term ϕ_{ut},* trace satisfiability *is the smallest ternary relation between traces, user-assignment relations, and terms, written $t \models^{\mathcal{T}}_{UA} \phi$, closed under the following rules:*

$$(1)\ \frac{\{\mathsf{user}(b)\} \models^{\mathcal{M}}_{UA} \phi_{ut}}{\langle b\rangle \models^{\mathcal{T}}_{UA} \phi_{ut}} \qquad (2)\ \frac{t \models^{\mathcal{T}}_{UA} \phi}{t\,\hat{}\,\langle a\rangle \models^{\mathcal{T}}_{UA} \phi} \qquad (3)\ \frac{t \models^{\mathcal{T}}_{UA\cup\{(u,r)\}} \phi}{\langle \mathtt{addUA}.u.r\rangle\hat{}\,t \models^{\mathcal{T}}_{UA} \phi}$$

$$(4)\ \frac{t \models^{\mathcal{T}}_{UA\setminus\{(u,r)\}} \phi}{\langle \mathtt{rmUA}.u.r\rangle\hat{}\,t \models^{\mathcal{T}}_{UA} \phi} \qquad (5)\ \frac{\langle b\rangle \models^{\mathcal{T}}_{UA} \phi_{ut}}{\langle b\rangle \models^{\mathcal{T}}_{UA} \phi^{+}_{ut}} \qquad (6)\ \frac{\langle b\rangle \models^{\mathcal{T}}_{UA} \phi_{ut}\ ,\ \ t \models^{\mathcal{T}}_{UA} \phi^{+}_{ut}}{\langle b\rangle\hat{}\,t \models^{\mathcal{T}}_{UA} \phi^{+}_{ut}}$$

$$(7)\ \frac{t \models^{\mathcal{T}}_{UA} \phi}{t \models^{\mathcal{T}}_{UA} \phi \sqcup \psi} \qquad (8)\ \frac{t \models^{\mathcal{T}}_{UA} \psi}{t \models^{\mathcal{T}}_{UA} \phi \sqcup \psi} \qquad (9)\ \frac{t \models^{\mathcal{T}}_{UA} \phi\ ,\ \ t \models^{\mathcal{T}}_{UA} \psi}{t \models^{\mathcal{T}}_{UA} \phi \sqcap \psi}$$

$$(10)\ \frac{t_1 \models^{\mathcal{T}}_{UA} \phi\ ,\ \ t_2 \models^{\mathcal{T}}_{UA} \psi}{t \models^{\mathcal{T}}_{UA} \phi \odot \psi} \quad \mathsf{si}(t, t_1, t_2)$$

$$(11)\ \frac{t_1 \models^{\mathcal{T}}_{UA} \phi\ ,\ \ t_2 \models^{\mathcal{T}}_{UA} \psi}{t \models^{\mathcal{T}}_{UA} \phi \otimes \psi} \quad \mathsf{si}(t, t_1, t_2)\ \textit{and}\ \mathsf{users}(t_1) \cap \mathsf{users}(t_2) = \emptyset$$

We say that t *satisfies ϕ with respect to UA*, if $t \models^{\mathcal{T}}_{UA} \phi$. SoDA$^{\mathcal{T}}$ fulfills the requirements of Sec. 5.1. (R1) follows from rules (2) to (4) of Def. 6, (R3) follows from the rule (1), and (R4) from the rules corresponding to the respective operators. The satisfaction of (R2) is shown by the following lemma that relates SoDA$^{\mathcal{M}}$ and SoDA$^{\mathcal{T}}$, which we prove in [11].

Lemma 2. *For all terms ϕ, all user-assignment relations UA, and all traces $t \in \mathcal{E}_B^*$, if $t \models^{\mathcal{T}}_{UA} \phi$, then $\mathsf{users}(t) \models^{\mathcal{M}}_{UA} \phi$.*

Example 6. Consider again the term ϕ and the trace t from Example 5. Under SoDA$^{\mathcal{T}}$, t satisfies ϕ with respect to $UA = \emptyset$. However,

$$t' := \langle ac.\mathtt{addUA.Bob.Manager},\ bc.\mathtt{Alice}.a,\ ac.\mathtt{rmUA.Bob.Manager},\ bc.\mathtt{Bob}.a'\rangle\ ,$$

does not satisfy ϕ with respect to $UA = \emptyset$, because no action in t' is executed by a user who acts as `Manager`.

5.3 Mapping Terms to Processes

First, we introduce the auxiliary process FIN that engages in an arbitrary number of admin events before it engages in *done*, and finally behaves like $STOP$.

$$FIN = (done \rightarrow STOP) \,\Box\, (ac.a : \mathcal{A}_A \rightarrow FIN)$$

Using FIN, we define the mapping $[\![.]\!]^U_{UA}$.

Definition 7 (Mapping $[\![.]\!]^U_{UA}$). *Given a set of users U, a user-assignment relation UA, and a term ϕ, the* mapping $[\![\phi]\!]^U_{UA}$ *returns a process parametrized by UA. For a unit term ϕ_{ut} and terms ϕ and ψ, the mapping $[\![.]\!]^U_{UA}$ is defined as follows.*

$$\begin{aligned}
(1)\quad [\![\phi_{ut}]\!]^U_{UA} &:= bc?u : \{u' \in U \mid \{u'\} \models^{\mathcal{M}}_{UA} \phi_{ut}\}.a : \mathcal{A} \rightarrow FIN \\
&\quad \Box\, ac.\mathtt{addUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow [\![\phi_{ut}]\!]^U_{UA \,\cup\, \{(u,r)\}} \\
&\quad \Box\, ac.\mathtt{rmUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow [\![\phi_{ut}]\!]^U_{UA \,\setminus\, \{(u,r)\}}
\end{aligned}$$

$$\begin{aligned}
(2)\quad [\![\phi^+_{ut}]\!]^U_{UA} &:= bc?u : \{u' \in U \mid \{u'\} \models^{\mathcal{M}}_{UA} \phi_{ut}\}.a : \mathcal{A} \rightarrow (FIN \,\Box\, [\![\phi^+_{ut}]\!]^U_{UA}) \\
&\quad \Box\, ac.\mathtt{addUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow [\![\phi^+_{ut}]\!]^U_{UA \,\cup\, \{(u,r)\}} \\
&\quad \Box\, ac.\mathtt{rmUA}?u : \mathcal{U}?r : \mathcal{R} \rightarrow [\![\phi^+_{ut}]\!]^U_{UA \,\setminus\, \{(u,r)\}}
\end{aligned}$$

$$(3)\ [\![\phi \sqcup \psi]\!]^U_{UA} := [\![\phi]\!]^U_{UA} \,\Box\, [\![\psi]\!]^U_{UA}$$

$$(4)\ [\![\phi \sqcap \psi]\!]^U_{UA} := [\![\phi]\!]^U_{UA} \underset{\Sigma}{\|} [\![\psi]\!]^U_{UA}$$

$$(5)\ [\![\phi \odot \psi]\!]^U_{UA} := [\![\phi]\!]^U_{UA} \underset{\{done\} \,\cup\, \mathcal{E}_A}{\|} [\![\psi]\!]^U_{UA}$$

$$(6)\ [\![\phi \otimes \psi]\!]^U_{UA} := \underset{\{(U_\phi, U_\psi) \mid U_\phi \cup U_\psi = U \, and \, U_\phi \cap U_\psi = \emptyset\}}{\Box} [\![\phi]\!]^{U_\phi}_{UA} \underset{\{done\} \,\cup\, \mathcal{E}_A}{\|} [\![\psi]\!]^{U_\psi}_{UA}$$

Note that the equations (1) and (2) require determining whether $\{u'\} \models^{\mathcal{M}}_{UA} \phi_{ut}$. This problem is analogous to testing whether a propositional formula is satisfiable under a given assignment and is also decidable in polynomial time.

Definition 8 (SoD-enforcement process). *For a term ϕ and a user-assignment relation UA, the* SoD-enforcement process *is the process $SOD_\phi(UA) := [\![\phi]\!]^{\mathcal{U}}_{UA}$.*

Before we show how an SoD-enforcement process is used together with workflows and the RBAC process, we define correctness for the mapping $[\![.]\!]^U_{UA}$.

Definition 9 (Correctness of $[\![.]\!]^U_{UA}$). *The mapping $[\![.]\!]^U_{UA}$ is* correct *if for all terms ϕ, all user-assignment relations UA, and all traces $t \in \Sigma^*$, $t \in \mathcal{T}(SOD_\phi(UA))$ and $\mathsf{done}(t)$ if and only if $t' \models^{\mathcal{T}}_{UA} \phi$, for $t = t' ^\frown \langle done \rangle$, where t' ranges over $(\mathcal{E}_B \cup \mathcal{E}_A)^*$.*

Informally, the mapping $[\![.]\!]^U_{UA}$ is correct if the following properties hold for all SoD-enforcement processes SOD_ϕ: (1) if SOD_ϕ accepts a finished workflow run, the corresponding trace satisfies ϕ under SoDA$^{\mathcal{T}}$, and (2) if a trace satisfies ϕ under SoDA$^{\mathcal{T}}$, the corresponding finished workflow run is accepted by SOD_ϕ. We prove Theorem 1 in [11].

Theorem 1. *The mapping* $[\![.]\!]^U_{UA}$ *is correct.*

Hence, if the SoD-enforcement process accepts a finished workflow run, then the corresponding SoD constraint is satisfied. We also know that no compliant workflow run is falsely blocked by the SoD-enforcement process. The following corollary relates the set of traces of SoD-enforcement processes without administrative events and their corresponding multisets of users under the multiset semantics. Its proof follows directly from Theorem 1 and Lemma 2.

Corollary 1. *For all terms ϕ, all user-assignment relations UA, and all traces $t \in \mathcal{E}_B^*$, if $t\,\hat{}\,\langle done\rangle \in \mathcal{T}(SOD_\phi(UA))$, then* $\mathsf{users}(t) \models^{\mathcal{M}}_{UA} \phi$.

Given a process W that models a workflow and a term ϕ that models an SoD policy, the *SoD-secure (workflow) process* SSW_ϕ is the parallel, partially synchronized composition of W, the RBAC process, and the SoD-enforcement process SOD_ϕ.

$$SSW_\phi(UA, PA) = (W \underset{\mathcal{E}_B}{\parallel} RBAC(UA, PA)) \underset{\Sigma}{\parallel} SOD_\phi(UA)$$

Let $b := bc.u.a$ be a business event. $SSW_\phi(UA, PA)$ engages in b if W, $RBAC(UA, PA)$, and $SOD_\phi(UA)$ each engage in b. In other words, b must be one of the next actions to be taken according to the workflow specification, the user u must be authorized to execute the action a according to the RBAC configuration (UA, PA), and u must not violate the SoD policy ϕ, given the previously executed business events and UA. Furthermore, $RBAC$ and SOD_ϕ can synchronously engage in an admin event at any time. Finally, $SSW_\phi(UA, PA)$ engages in $done$ if both W and $SOD_\phi(UA)$ synchronously engage in $done$.

Example 7 (SoD-secure workflow process). Assume that the users who execute actions in our payment workflow must comply with the SoD policy described by the term ϕ of Example 3. Example 2 shows that $t' \in \mathcal{T}(SW(UA, PA))$. In contrast, $t' \notin \mathcal{T}(SSW_\phi(UA, PA))$ because `Bob` is not authorized to execute both the actions `prepare check` and `approve payment`. Hence, SSW_ϕ reduces the risk of fraudulent payments described in Example 2. We change t' to t'' by adding the admin event ac.`addUA.Claire.Manager` and let `Claire` execute `approve payment`.

$t'' := \langle bc.$`Alice.receive invoice`, $bc.$`Bob.prepare check`, $ac.$`addUA.Bob.Manager`, $ac.$`addUA.Claire.Manager`, $bc.$`Claire.approve payment`, $bc.$`Alice.issue check`, $done\rangle$

The new admin event adds the role assignment (`Claire`,`Manager`) to SSW_ϕ's RBAC configuration as shown by the dashed line in Fig. 2. The trace t'' without $done$ satisfies ϕ with respect to UA under SoDA$^{\mathcal{T}}$. Furthermore, $t'' \in \mathcal{T}(SSW_\phi(UA, PA))$.

This completes our running example and illustrates how the three kinds of processes presented in this paper interact and how each of them enforces its corresponding policy: W formalizes the workflow model, $RBAC$ formalizes a possibly changing access control policy, and $SOD_{\phi}(UA)$ formalizes the SoD policy, while accounting for changing role assignments.

5.4 From Processes to Enforcement Monitors

CSP's *operational semantics* interprets a process as a *labeled transition system (LTS).* It is straightforward to translate an LTS into a program that only allows the execution of actions as defined by the process. The program thereby constitutes an enforcement monitor for the policy specified by the process, analogous to the security automata in [12]. The mapping $[\![.]\!]^{U}_{UA}$ may yield a nondeterministic process. However, the corresponding LTS can either be determinized or the enforcement monitor can keep track of the set of reachable states after each transition, essentially performing a power-set construction, on-the-fly.

As shown in Sec. 5.3, an SoD-secure process is the parallel execution of three subprocesses, each responsible for a specific task. Due to the associativity of CSP's $||$-operator, these three processes can be grouped in any order. Furthermore, the set of events on which these processes synchronize defines the kinds of events each process engages in. Therefore, any subset of these three processes can be mapped to an enforcement monitor and the set of events synchronized with the remaining processes specifies the monitor's interface. This is of particular interest if a system already provides one of the components we model by our processes. For example, assume a system comes with a workflow engine and an access control enforcement monitor. In this case, it is sufficient to generate an enforcement monitor for the SoD-enforcement process and to synchronize all business and admin events with the existing components.

6 Related Work

There are many formalisms for modeling workflows, for example BPMN [13] and WS-BPEL [14]. Process algebras have often been used to give these a formal semantics; see for example [15]. There are also numerous models and frameworks to formalize and enforce separation of duty constraints [16,17]. Although in general more complex, dynamic SoD enforcement is more flexible than static enforcement and therefore more interesting for real-world settings. Our work is the first to model dynamic enforcement of SoD constraints with changing role assignments.

Most SoD mechanisms describe and enforce constraints between two or more explicit actions and are therefore tightly coupled with the workflow definition [4,18,19]. In contrast, our approach allows a workflow-independent specification of SoD constraints and their enforcement on different workflows. This has the advantages discussed in Sec. 1 but does not support action-specific constraints. However, if desired, such constraints could be expressed as a further refinement of our SoD-enforcement processes.

In [4], *transaction control expressions* define dynamic SoD constraints on data objects. Enforcement decisions are made at run-time, based on the history of executed actions. A workflow, associated with a data object, is defined by a list of actions, each with one or more attached roles. A user is authorized to execute an action if she acts in one of these roles. By default, all actions must be executed by different users. Constraints are less expressive than SoDA terms and they can only be defined in combination with a concrete workflow.

In [18], Bertino, Ferrari, and Atluri check the consistency of constraints defined over workflows in a logical framework. Their constraints are defined with respect to the sequence of individual workflow actions, applying (first-order) predicates to action occurrences. Schaad, Lotz, and Sohr extend SoD analysis to workflows with dynamic access rights [20]. They describe the workflow, the associated access control policy, and the delegation and revocation steps as transitions of a finite state automaton and apply model checking to verify the constraints expressed in linear temporal logic. However, neither of these papers provide a mapping to an enforcement mechanism.

Knorr and Stormer [19] map dynamic SoD constraints along with the workflow to Prolog clauses computing all workflow runs that do not violate the specified SoD constraints. In Nash and Poland's *object-based separation of duties* [21], each data object keeps track of the users who have executed actions on it. If a user requests to execute an action on an object, this is only granted if he has not executed an action on this object before. This functionality can be modeled with our formalism if every data object is protected by an SoD-enforcement process.

In [5], Wang and Li also presented an enforcement mechanism for SoDA terms. In contrast to our work, their approach is static and not applicable to all combinations of terms, roles, and permission-assignment relations. In particular, the use of the $\neg$-operator can invalidate a large subset of assignment relations.

7 Conclusions

We have showed how to map SoDA terms onto workflows in a general way that also supports administrative actions. The key ideas were (1) to extend SoDA's semantics to traces, handling both multiple actions by users and administrative actions, and (2) to map SoDA terms to processes, which interact with workflow and access control processes. Because all components are defined in CSP, we can directly employ CSP's operational semantics to map these processes to a workflow engine that performs the necessary security checks at run-time.

As future work, we will explore how to best implement our SoDA processes and integrate them with existing workflow engines. Efficiency is a central question in this regard. In our mapping to CSP, we focused on providing an abstract specification of a SoDA-enforcement mechanism, rather than an efficient one. In particular, the rule (6) of Def. 7 yields a state space that is exponential in the number of system users. We will investigate translations with improved complexity and the use of data-structures for efficiently representing extended state-machines. We will also explore optimization techniques, such as pruning the state space to eliminate the states of workflow runs from which no final state can be reached, no matter which changes are made to the RBAC configuration.

Acknowledgments. We thank Felix Klaedtke, Samuel Müller, Christoph Sprenger, and the anonymous reviewers for their helpful comments.

References

1. Enron, See you in court. The Economist, November 15 (2001)
2. Sarbanes-Oxley Act of 2002. Public Law 107-204 (116 Statute 745), United States Senate and House of Representatives in Congress (2002)
3. Saltzer, J., Schroeder, M.: The Protection of Information in Computer Systems. Proceeding of the IEEE 63(9), 1278–1308 (1975)
4. Sandhu, R.S.: Transaction Control Expressions for Separation of Duties. In: 4th IEEE Aerospace Computer Security Applications Conference, pp. 282–286 (1988)
5. Li, N., Wang, Q.: Beyond separation of duty: An algebra for specifying high-level security policies. Journal of the ACM 55(3) (2008)
6. Ferraiolo, D.F., et al.: Proposed NIST Standard for Role-Based Access Control. ACM Trans. on Information and System Security 4(3), 224–274 (2001)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
8. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
9. Syropoulos, A.: Mathematics of Multisets. In: Multiset Processing, pp. 347–358 (2000)
10. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)
11. Basin, D., Burri, S.J., Karjoth, G.: Dynamic Enforcement of Abstract Separation of Duty Constraints. IBM Research Report RZ3726 (2009), `domino.watson.ibm.com/library/cyberdig.nsf/Home`
12. Schneider, F.B.: Enforceable Security Policies. ACM Transactions on Information and System Security 3(1), 30–50 (2000)
13. Business Process Modeling Notation (BPMN). OMG Standard, v. 1.1 (2008)
14. Web Services Business Process Execution Language (WS-BPEL). OASIS Standard, v. 2.0 (2007)
15. Wong, P.Y.H., Gibbons, J.: A Process-Algebraic Approach to Workflow Specification and Refinement. In: Int. Symp. on Software Composition, pp. 51–65 (2007)
16. Gligor, V.D., Gavrila, S.I., Ferraiolo, D.: On the Formal Definition of Separation-of-Duty Policies and their Composition. In: 19th IEEE Symposium on Security and Privacy, pp. 172–183 (1998)
17. Simon, R., Zurko, M.E.: Separation of Duty in Role-based Environments. In: 10th IEEE Workshop on Computer Security Foundations, pp. 183–194 (1997)
18. Bertino, E., Ferrari, E., Atluri, V.: The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. ACM Transactions on Information and System Security 2(1), 65–104 (1999)
19. Knorr, K., Stormer, H.: Modeling and Analyzing Separation of Duties in Workflow Environments. In: 16th Int. Conf. on Information Security, pp. 199–212 (2001)
20. Schaad, A., Lotz, V., Sohr, K.: A Model-checking Approach to Analysing Organisational Controls in a Loan Origination Process. In: 11th ACM Symposium on Access Control Models and Technologies, pp. 139–149 (2006)
21. Nash, M.J., Poland, K.R.: Some Conundrums Concerning Separation of Duty. In: IEEE Symposium on Security and Privacy, pp. 201–207 (1990)