On Process Rewriting for Business Process Security
(Extended Abstract)

Rafael Accorsi

University of Freiburg, Germany
`accorsi@iig.uni-freiburg.de`

Abstract. This paper reports on ongoing work towards a framework to automatically rewrite business process models and, thereby, correctively enforce adherence to regulatory compliance and security policies. Specifically, the paper first motivates the need for rewriting mechanisms as a means to enforce a particular class of properties. Second, it describes the main building blocks of ReWrite, the framework to automatically rewriting process specifications. Third, in order to preserve the functional goals of the processes upon rewriting, a set of congruence relations is defined and their appropriateness is discussed. The presentation of the formal framework and rewriting algorithms is deferred to the full paper version.

1 Introduction

Ensuring the compliance of business processes to regulations and security policies is of utmost importance in business process management [1, 31]. Approaches to assess compliance can be classified into [8, 25]: (1) *forward compliance checking* aims to design and implement processes where conforming behavior is enforced and (2) *backward compliance checking* aims to detect and localize non-conforming behavior. This paper focuses on forward compliance checking based on business process models and policies. In this setting, previous work addresses the annotation of business processes [15, 23], requirements elicitation [11, 10, 26] and formal verification [2–4, 6]. None of the previous work has considered the intersection of rewriting techniques for programs and the corrective enforcement of business processes compliance. This paper sets out to investigate this connection.

Specifically, this paper starts by motivating the need for rewriting mechanisms as a means to enforce a class of complex safety properties which encompass compliance requirements. In contrast to previous works, which merely present the challenge of enforcing compliance, this paper formally substantiates the need for rewriting using formal enforcement theories of computer security. By doing so, one establishes the relationship between the class of properties and the corresponding enforcement mechanisms needed to guarantee these properties.

Based on this, the paper further presents the building blocks and examples of the framework ReWrite to automatically rewrite business processes specifications and, thereby, ensure compliance with pertinent policies. The overall approach is depicted in Fig. 1. The key insight is to split responsibilities during the modeling phase and focus on the core competencies of each actor: the business experts design a process model (e.g. using BPMN or BPMN) and compliance experts design the compliance policy (i.e. description of applicable controls) and denote the processes to which they apply. The goal of ReWrite is to take process and policy specifications as input and produce an executable process model which complies to the policy.

Clearly, modifying the structure of the process may lead to a specification that does not achieve the business goals designated for that process. To circumvent this issue while still allowing for rewriting, one must define similarity relations between the original and the modified processes in terms of execution traces their produce: only modifications that preserve the similarity relation are appropriate. These relations are, however, not characterized in the literature. This paper defines and investigates the adequacy of four congruence relations. Initial investigations carried out with ReWrite show that rewriting operators – append, delete and replace activities in process models – can be supported by such relations, thereby allowing for effective rewriting algorithms.

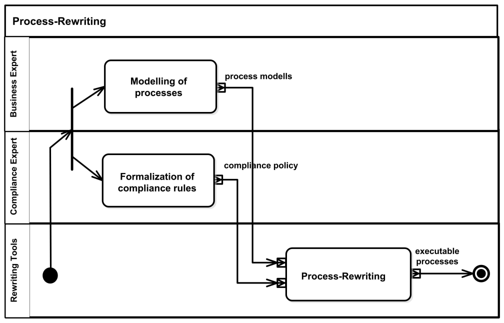


Fig. 1. ReWrite approach.

Taking stock, this paper provides the following contributions:

– Establishes the relationship between the compliance requirements, the underlying formal properties and the enforcement mechanisms. In particular, it shows a class of properties that can only be enforced using process rewriting and that process rewriting can enforce all other kinds of properties.
– Presents the main building blocks of ReWrite. Specifically, it presents μBPMN, a Turing complete fragment of BPMN to express business processes and a meta-model for the expression of compliance policies. An example will illustrate the interplay of these requirements for rewriting.
– Defines four congruence relations to guarantee the compliance with requirements while preserving the business (functional) goal of the process and discusses their adequacy for rewriting.
– Reports on ongoing implementation of the ReWrite framework "as-a-service" and describes how its evaluation will be conducted.

The rewriting framework proposed in this paper can be used in three dimensions and timepoints along the business process management lifecycle. Firstly, *before* the process execution to ensure compliance "by design". Here, if one assumes that process are stable and faithfully executed by the business process management system, then rewriting is capable of enforcing the designated class

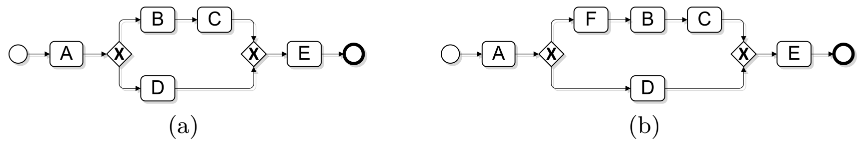


Fig. 2. Original process (a) and process after rewriting (b) .

of compliance properties. Secondly, *during* the execution for corrective enforcement. In this setting, an execution monitor with rewriting capabilities is needed to enforce properties. While this dimension comes along with a considerable performance overhead, it allows for flexible process whose compliance can be guaranteed at runtime. Thirdly, *after* the execution as a mechanism for automated process enhancement and re-design based upon process models reconstructed from event logs using process mining [30].

While the foundation of ReWrite can be used in any of these perspectives and timepoints, this paper focuses on the "by design" perspective – especially in terms of implementation. We currently apply these techniques to re-design reconstructed processes. Carrying over these concepts to in-line process monitoring is subject of future work. Further, it should be remarked that the focus of this paper is the presentation of the ReWrite framework, its building blocks and application. Pertinent proofs for properties – e.g., the Turing completeness μBPMN – are deferred to a longer version of the paper.

Paper structure. Section 2 establishes the relationship between classes of compliance properties and the corresponding enforcement mechanisms, thereby motivating the need for rewriting. Section 3 introduces the main building blocks of the rewriting approach, i.e. the process language and policies. Section 4 introduces the congruence relations and the corresponding rewriting operations. Section 5 reports on the implementation and evaluation of the approach.

Example 1. We illustrate the high-level operation of ReWrite using an example. Consider the process model in Fig. 2(a). It stands for a medical workflow, namely updating the patient record. Activity A denotes a staff member inputting the `patient_ID`. If the `ID` exists, B queries the database and C updates the database with the updated patient record; otherwise, if there is no such an `ID`, D issues an error message. In both cases, E deletes local copies of query and record. One security policy may require that, *before* showing the query results, the staff member must be authenticated and authorized to do so, which is encoded with the activity F. Given that information, rewriting would inspect the original model to detect whether F is at the correct place. If not, ReWrite will add F to the process, which produces the process model depicted in Fig. 2(b). The remainder of this paper shows how to carry out such a rewriting. $\dashv$

2 Properties and Enforcement Mechanisms

Non-functional properties – be it security or privacy properties, or properties arising from compliance requirements – can be classified according to two hierarchies: (1) the *Safety-Progress* based on languages used to *formalize* the properties [12] and (2) the *Safety-Liveness* based on mechanisms used to *enforce* the properties [7]. This section argues that the enforcement of compliance requirements demands process rewriting. It does so by revisiting these two theories.

2.1 Safety-Progress Hierarchy

Chang et al. [12] define four operators and, based upon them, six formal languages organized in a hierarchy. Let Σ^* be the set of all finite words over an alphabet Σ, Σ^+ the set of non-empty, finite words and Σ^w the set of all infinite words over the alphabet Σ. Let Φ be a finite language, the operators are:

– $A(\Phi)$ encompasses all σ, s. t. *every* prefixes of σ are in Φ.
– $E(\Phi)$ encompasses all σ, s. t. *some* prefixes of σ are in Φ.
– $R(\Phi)$ encompasses all σ, s. t. *infinite many* prefixes of σ are in Φ.
– $P(\Phi)$ encompasses all σ, s. t. *all but a finite number* of prefixes of σ are in Φ.

With these operators at hand one can define the six languages which build, for finite languages Φ and Ψ, the hierarchy depicted in Fig. 3.

1. *Safety language*: $\Pi = A(\Phi)$.
2. *Guarantee language*: $\Pi = E(\Phi)$.
3. *Obligation language*: $\Pi = \cap_{i=1}^{m}(A(\Phi_i) \cup E(\Psi_i))$.
4. *Response language*: $\Pi = R(\Phi)$.
5. *Persistence language*: $\Pi = P(\Phi)$.
6. *Reactivity language*: $\Pi = \cap_{i=1}^{m}(R(\Phi_i) \cup P(\Psi_i))$.

In Fig. 3, each language encompasses those beneath it. This follows directly from the definition of the languages. For example, the language *Obligation* is built by the conjunction of the languages *Safety* and *Guarantee*. Further, all the *Safety* languages can be expressed in terms of the *Obligation* language, wheras the corresponding *Guarantee* part is empty (i.e. $E(\Psi_i) = \emptyset$)

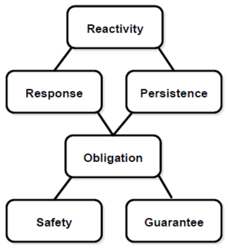


Fig. 3. Safety-response.

Obligation and Guarantee. Due to the existence operator $E(\Phi)$ the *Obligation* and *Guarantee* languages are relevant for the formalization of security and compliance requirements. For example, classical access control could be expressed in a way that the required prefix – denoting the process state – is available already by during the access decision. In this case, it can be decided that the property holds and could be no longer violated.

In another case, the required prefix is not at the decision time and must be built during the further continuation of the process instance. Here, one cannot decide whether the obligation will be fulfilled in a later point in time. The enforcement of such property with classical access control is not possible. That is because it cannot be decided whether the obligation will be fulfilled or not. If at a given timepoint t one is to evaluate where a process instance is in the *Obligation* language O_1, then four different states are possible [20]: (1) *true* it can be shown that the instance is in O_1; (2) *possibly true* at t the instance is in O_1, but there is the possibility that the whole instance will not be in O_1; (3) *possibly false* at t the instance is *not* in O_1, but it is possible that will eventually be in the language; finally, (4) *false*, it can be shown that the instance is not in O_1.

The rewriting approach presented in this paper allows for the automated enforcement of cases that evaluate to *possibly false*. The other cases can be tackled with existing access control mechanisms, in that they on the one hand prevent the transition from *possibly true* to *false* or *possibly false*, or if they do not allow the execution of a *false* instance.

2.2 Safety-Progress Hierarchy

This hierarchy is defined upon three well-known classes of preventive enforcement mechanisms, namely: (1) *static verification* before the execution; (2) *runtime monitoring* during the execution; and (3) *rewriting* before or during the execution. In particular, each of these mechanisms can recognize and therefore circumscribe a particular class of properties. The following formally defines the hierarchy, which is depicted in Fig 4.

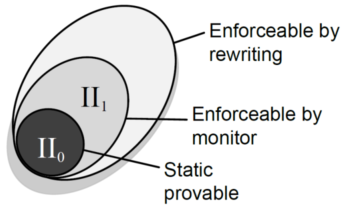


Fig. 4. Safety-liveness hierarchy.

Class Π_0: Statically enforceable properties. A policy $\mathcal{P}$ is statically enforceable if there is a Turing machine $M_{\mathcal{P}}$ for $\mathcal{P}$ that terminates in finite time if $\mathcal{P}$ holds in the process. It can be shown that the class of enforceable security properties corresponds to the class of recursively decidable properties of programs, which in turn corresponds to the arithmetic class Π_0. (See [16] for details.) Several properties can be statically verified before the execution of a process [17]. The proof of a property stating that, e.g., a (stable) process calls exactly 30 services is trivial: enumerate and count the service calls.

Class Π_1: Runtime properties. The basis for the formalization of properties of programs which can be enforced during the execution (so-called "execution monitoring") is an execution machine that generates traces (sequences of events). Given a trace, for properties that can be enforced with an execution monitor there must be a detector that identifies their violation during the runtime. Such a detector must exhibit the following properties [28]: (1) the property detected by the detector is irremediably violated; (2) the detector must identify the violation in finite time; and (3) the detector mus be recursively decidable.

The following criteria thus formally define the detector:

$$\forall \Pi \in \Psi(\Pi) : \mathcal{P}(\Pi) \Rightarrow \forall \sigma \in \Pi : \hat{\mathcal{P}}(\sigma) \tag{1}$$

$$\forall \tau' \in \Psi : \neg\hat{\mathcal{P}}(\tau') \Rightarrow \forall \sigma \in \Psi : \neg\hat{\mathcal{P}}(\tau'\sigma) \tag{2}$$

$$\forall \sigma \in \Psi : \neg\hat{\mathcal{P}}(\sigma) \Rightarrow \exists i : \neg\hat{\mathcal{P}}(\sigma[..i]) \tag{3}$$

$$\sigma \notin \Gamma \Rightarrow \exists i : (\forall \tau \in \Psi : \sigma[..i]\tau \notin \Gamma) \tag{4}$$

Based upon this, it can be shown that the class of security properties enforced by the monitor correspond to the class of co-recursively countable properties of programs, thereby circumscribing the class Π_1 of the arithmetic hierarchy.

Rewriting properties. The third class of properties in this hierarchy are those which can be enforced with the modification of the program. The definition of rewriting requires an adequate notion of equivalence. (See Sect. 4 for details.) For a given equivalence relation $\approx$, a rewriting mechanism R is considered "adequate" for enforcement if it exhibits the following properties:

$$P(R(M)) \tag{5}$$

$$P(M) \Rightarrow M \approx R(M) \tag{6}$$

These properties express that: Eq. (5) the modified program must guarantee the compliance with a policy; and Eq. (6) if a program complies with a policy before the rewriting, then it also complies with if after a modification. Hamlen et al. [16] show that there are properties which can solely be enforced with rewriting. However, in contrast to the previous classes, the rewriting mechanism does not corresponds to a class in the arithmetic hierarchy. That is, there is no known definition for a set of formal languages whose words denote properties, which can be enforced with rewriting. Conversely, it can be shown that the classes of properties Π_0 (static verification) and Π_1 (runtime monitoring) can be enforced with rewriting mechanisms.

3 Building Blocks: Processes Models and Policies

The previous section demonstrates that there is a class of properties which demand process rewriting for their enforcement. This section describes the technical building blocks that serve as input to ReWrite, namely process specifications and policies, as depicted in Fig. 1.

3.1 μBPMN Syntax

Our rewriting approach considers μBPMN as the modeling language for business processes. μBPMN is a Turing complete subset of BPMN containing its main constructs a formal semantics. Turing completeness is insofar relevant as it guarantees that all the computable processes can be modeled with and reasoned about in μBPMN. The formal semantics is essential to allow the rewriting and prove its correctness with regard to the congruence relations.

The μBPMN syntax possesses both a graphical and an algebraic notation. The graphic notation is the same as the corresponding of BPMN construct, whereas only the subset of the elements depicted in Fig. 5 is considered. The algebraic formalization is required to give semantics to the language, which is in this case based upon π-calculus [27]. (The latter is omitted in this paper.)

The graphical notation of μBPMN depicted in Fig. 5 consists of activities (sub-processes, loops and sub-process loops) and the gateways AND, OR and XOR, which allow for the complete representation of logical constructs in the control flow. Further, activities are linked with sequence arrows. Start and end markers complete the subset of BPMN used in this paper. Based upon [22], we have defined translations of μBPMN to BPEL – for execution – and workflow nets – for reasoning, e.g. soundness.

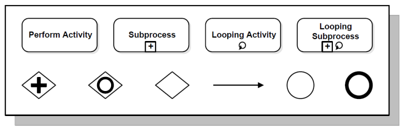


Fig. 5. μBPMN language.

The consideration of subset of BPMN does not restrict the applicability of the approach. First, because more complex modeling elements are seldom employed. According to [32], process models in industry usually consist of a few activities with complex branches. Second, because the language can be extended using the existing building blocks, should that be required.

Process modeled using the μBPMN are mapped a subset of the π-calculus. Processes are thus described by a set of activities that are triggered one after the other, provided the preconditions for their firing hold.

Relevant bits of μBPMN semantics. μBPMN has an operational semantics defined via an abstract process execution engine – mimicking the operation of existing BPMN engines – whose function is to determine the state transitions (choice of the next activity) and the execution environment for the process execution (provision of runtime parameters).

For the purposes of this paper, it is enough to consider a trace-based semantics for μBPMN based upon this calculus. Let Π be a process, a path χ of Π is defined as a sequence of input/output operations of Π upon the input $\omega \in \Sigma$. The set of all possible input parameters of a process Π is denoted by Γ^{ω}; $\Pi(\omega) \rightarrow \chi_{\omega}$ denotes the path triggered by inputting ω into Π, which eventually produces the path denoted by χ^{Π}_{ω}. Finally, Ξ_{Π} denotes the set of all paths in a process generated by $\Pi(\Gamma^{\omega})$. The congruence relations defined in Sect. 4 build upon reductions of generated paths in order to describe the changes (rewriting) in the process structure before the runtime. These changes are denoted as follows: $n \rightsquigarrow \chi_{\omega}$ when activity names are deleted in one path; $\mathcal{N} \rightsquigarrow \chi_{\omega}$ if activity names are deleted in all the possible paths of Π.

3.2 Compliance Rules

Figure 6 depicts the overall structure of a compliance rule in ReWrite. Each rule consists of a scope, a modality and a control. The *scope* defines the process and, therein, the control flow to which the rule applies. The *modality* describes how

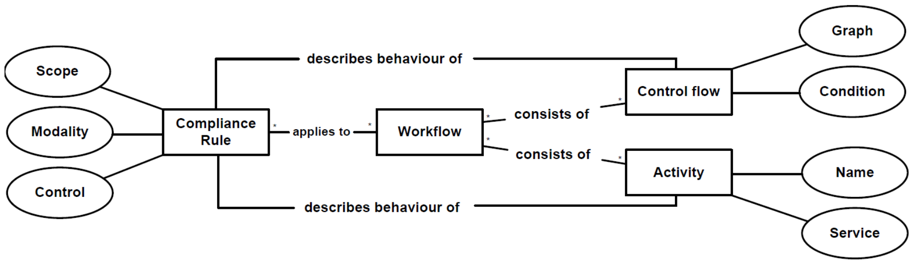


Fig. 6. The structure of compliance rules.

and where to apply the rule in a particular scope. The *control* defines which set of activities are to be triggered, as well as their order, so to ensure rule compliance.

This structure is expressive enough to capture the so-called "usage control" policies and, hence, the majority of regulatory compliance requirements on automated processes [24]. Generally, usage control policies refer to control flow constraints based upon the patterns in Dwyer et al. [13]. That is, they regard, for example, the absence, presence and cardinality of events, as well as their interdependencies (e.g. one event as a response of the other and mutual exclusion of event pairs). The evaluation envisaged for ReWrite considers these patterns.

More specifically, the class diagram depicted in Fig. 7 shows how the compliance rules are implemented in ReWrite. The scope consists of a list of processes to which the rule applies and one or more XPath expressions that describe the integration of (the controls specified in) this rule into the set of processes. The modality defines how the set of activities defined in the scope are treated. These activities can be deleted, replaced or complement with other activities. The latter (i.e. appending) may have different modes. For example, one can distinguish between appending before, after or during an activity (in one execution branch of a parallel execution). Similarly, one can add time constraints (e.g. "within three hours") and cardinality constraints (e.g. "exactly three times"). The controls with which the modalities are added to the process (process rewriting) are described as fragments of the BPEL language.

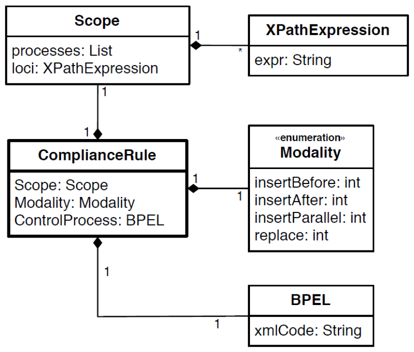
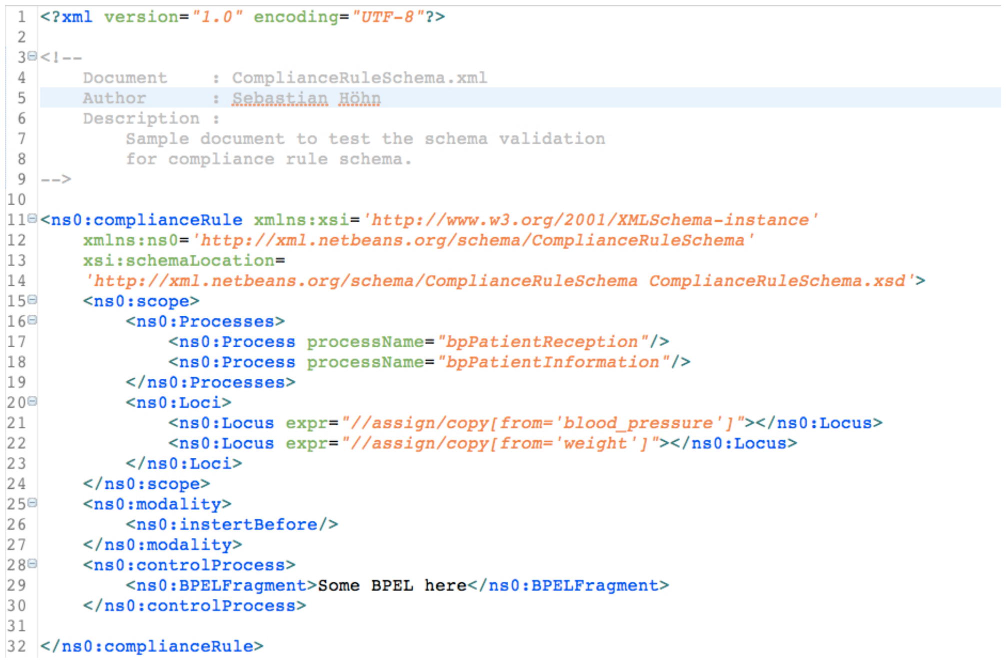


Fig. 7. Class diagram for rules.

We define a XML schema in order to express policies rules in a machine readable format. Figure 8 depicts a policy specified in this XML schema. This rule applies to two processes (`bpPatientReception` and `bpPatientInformation`), in particular their parts (`loci`-tag) `blood_pressure` and `weight` (here we replace the corresponding XPath expressions for simplicity). The `modality`-tag `insertBefore` conveys that the control process must be inserted before these `loci`. The control process, specified in the tag `BPELFragment`, can require in both



```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Document    : ComplianceRuleSchema.xml
    Author      : Sebastian Höhn
    Description :
        Sample document to test the schema validation
        for compliance rule schema.
-->

<ns0:complianceRule xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xmlns:ns0='http://xml.netbeans.org/schema/ComplianceRuleSchema'
    xsi:schemaLocation=
    'http://xml.netbeans.org/schema/ComplianceRuleSchema ComplianceRuleSchema.xsd'>
    <ns0:scope>
        <ns0:Processes>
            <ns0:Process processName="bpPatientReception"/>
            <ns0:Process processName="bpPatientInformation"/>
        </ns0:Processes>
        <ns0:Loci>
            <ns0:Locus expr="//assign/copy[from='blood_pressure']"></ns0:Locus>
            <ns0:Locus expr="//assign/copy[from='weight']"></ns0:Locus>
        </ns0:Loci>
    </ns0:scope>
    <ns0:modality>
        <ns0:instertBefore/>
    </ns0:modality>
    <ns0:controlProcess>
        <ns0:BPELFragment>Some BPEL here</ns0:BPELFragment>
    </ns0:controlProcess>

</ns0:complianceRule>
```

Fig. 8. Example of compliance rule.

cases the authentication of users using, for instance, their employee information. The concrete service invocation is omitted here.

4 Congruence Relations and Rewriting Operators

This section sketches the rewriting operators necessary to ensure compliance. While modifying processes models one must ensure that the functional characteristics of processes are maintained. Therefore, before defining the rewriting operators, this section sketches the congruence relations operators must fulfill.

4.1 Congruence Relations

Determining the semantic congruence of two arbitrary processes is a not decidable problem [18]. We define atomic operators and a stepwise approach to changing processes which turns out to be decidable and preserve the congruence relations. This section presents four congruence relations which are defined in terms of process paths. Taking two processes p_1 and p_2, the relations are:

– *Full semantic congruence:* all the paths generated by the process p_1 can also be generated by the process p_2. Formally: $\Xi_{p_1} = \Xi_{p_2}$. This relation is unsuitable for rewriting, as its own definition prohibits modifications.

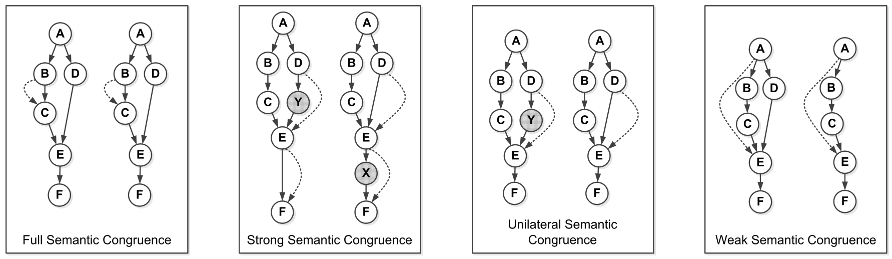


Fig. 9. Schematic illustration of the congruence relations.

- *Strong semantic congruence:* all the paths of the process p_1 can also be generated by the process p_2 after the reduction to common activity names. Formally: $\mathcal{N} = \Xi_{p_1} \cap \Xi_{p_2}$.
- *Unilateral semantic congruence:* the set of all paths generated by the process p_2 is a subset of the set of paths generated by p_1. Formally: $\Xi_{p_2} \subset \Xi_{p_1}$.
- *Weak semantic congruence:* At least one path generated the process p_1 is also a path of the process p_2. Formally: $\Xi_{p_1} \cap \Xi_{p_2} \neq \emptyset$.

Fig. 9 illustrates these relations. For simplicity, the processes are drawn as a simple state transition diagram (instead of μBPMN), with the dotted lines showing a possible move in the "congruence game". The shaded circles denote activities that have been added to the process flow. The pictures denote examples of a process flow that preserve the relations.

Lack of space prevents us from providing the formal definition of all these relations. Below we provide the formal definition for the strong semantic congruence, then jumping to the rewriting operators.

Definition 1. *Two processes p_1 and p_2 are strongly semantic congruent if the set of generated process paths after the reduction to the common names $\mathcal{N_C} = \Xi_{p_1} \cap \Xi_{p_2} \neq \emptyset$ is identical: $(\mathcal{N_C} \rightsquigarrow \Xi_{p_1}) = (\mathcal{N_C} \rightsquigarrow \Xi_{p_2})$.* $\dashv$

It is easy to show that this relation is commutative. The strong semantic congruent relation allows changes in the process paths, as the activities names are reduced to a set of names common to both processes. In doing so, rewriting operators can be defined using this definition.

4.2 Rewriting Operators

The relations introduced in Sect. 4.2 are undecidable for two arbitrary processes. This follows from Jancar [18], who has proved that for Petri nets. His proof can be carry over to μBPMN: for each Petri net used in [18] we can build a corresponding μBPMN process, a fact that follows from the Turing completeness of both languages.

Still, for rewriting to work it is necessary to guarantee that the original and the rewritten processes are semantically congruent and that the latter indeed

	Append	Delete	Replace
Full	no	no	no
Strong	yes	yes	yes
Unilateral	partial	partial	no
Weak	partial	yes	partial

Table 1. Overview of the congruence guarantees for rewriting operators.

executes the controls required by the compliance policies. To achieve this in a decidable manner, we define atomic operators for appending, deleting and replacing activities in the process which, whenever applied in isolation or sequentially (one after the other), guarantee that the semantic congruence holds. Due to the lack of space, the following focuses on the "append" operator. An overview of the congruence guarantees for all the rewriting operators is given in Table 1.

The "append" operator. The "append" operator adds (missing or required) activities to the process model. This operator can be applied without disturbing the semantic congruence of processes. However, the following restrictions have to be made for the case of the unilateral and weak semantic congruence:

– If an activity has already been appended to one of the process models, then it is impossible to append further activities to the other model without disturbing the unilateral semantic congruence relation.
– If the two processes possess solely one common path, then appending one activity may disturb the congruence relations.

We have proved these properties for the corresponding congruence relations. For the strong semantic congruence, the following can be shown:

Theorem 1. *The append operator preserves the strong semantic congruence.* ⊣

Proof (Sketch). By appending an additional activity A to a process, then either Ξ_{P_1} or Ξ_{P_2} are extended with further traces. The computation of $\mathcal{N}_{\mathcal{C}} = \Xi_{P_1} \cap \Xi_{P_2}$ according to Def. 1 will, however, remove this extension (by renaming the activities), so that the processes still fulfill the strong semantic congruence. □

An analogous procedure can be used to demonstrate that the append operator preserves the unilateral and the weak semantic congruence relations, as shown in Table 1. Note that the "replace" operator is defined on the grounds of the primitive operators "append" and "delete".

Adequacy of semantic congruence. Establishing a relationship between the rewriting operators, congruence relations and the original business goals of a process is not trivial. Here, one can distinguish between the adequacy of different relations. Considering the strong semantic congruence and the "delete" operators, for example. It is possible to remove nearly all activities of a process and still fulfill the strong semantic congruence. Our experience shows that the unilateral semantic congruence is the most promising relation within the ReWrite framework. It still suffers from the problem of the “delete” operator, but not to the same degree as the strong semantic congruence.

To tackle this problem, we add the following restrictions to the framework: (1) only activities that violate a compliance policy are deleted and, hence, should anyway *not* be executed; (2) if the semantic congruence is only violated at the same time that a compliance policy is violated, then the processes are still congruent. It can be shown that these restrictions are necessary and sufficient to tackle the problems arising from the “delete” operator.

5 Realization and Evaluation

The prototypical implementation of the ReWrite framework is based upon open-source technologies for the automation of processes with the standards of μBPMN, BPEL and automated translations from μBPMN into BPEL and the XML tools. This section reports on the realization of ReWrite and its envisaged evaluation.

The ReWrite framework has been designed as a component of the Security Workflow Analysis Toolkit (short, SWAT) [5]. SWAT is an Eclipse-based, extensible toolkit for the automated, well-founded security analysis of business processes models to analyze process models in a multitude of ways. SWAT provides for the following features: *process editing*, with import and export functions; *process simulation* to generate log files and configure policy violations using OpenESB as execution platform; *policy editing* to specify security and compliance policies; and *workflow analysis* to check whether process models comply with properties. See [5] for details.

Figure 10 depicts the architecture of ReWrite in SWAT. The design strategy while integrating ReWrite into SWAT is the “security-as-a-service” approach. Correspondingly, the architecture consists of two modules, namely: (1) *Modeling* for process design and policy specification; and (2) *Execution* for the rewriting of non-compliant process models and the automated execution of processes. Regarding (1), SWAT offers support for process modeling in μBPMN, BPEL and Petri nets, whereas for process execution the specifications are translated into BPEL. Compliance policies are specified using the XML editor “Oxygen”, which is embedded in SWAT. Ongoing work is designing a policy editor and consistency checker for compliance rules. Regarding (2), SWAT employs OpenESB as an execution platform, whereas Glassfish acts an application server. It should be noted, though, that the actual realization

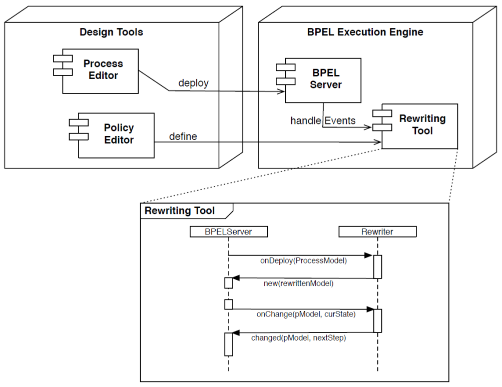


Fig. 10. ReWrite in SWAT.

of ReWrite does not require these technologies. Ongoing work is testing with a realization based upon jBPM.

The core of rewriting is actual modification of processes. The algorithms implementing these operators must guarantee, one the one hand, that the congruence relations are preserved (see Sect. 4.2) and, on the other, that the produced process models are correct with regard to the compliance policy. The latter depends on the compliance policy (i.e. controls defined therein) that are applied to the process. This is, however, out of the scope of this paper, as well as determining inconsistencies among policies.

The strategies for actually rewriting the processes is based on XSLT patterns, which are responsible for the transformation of XML documents (process models) with stylesheets. (Recall that the policy defines the scope of the rule, i.e., where to rewrite the process.) The following shows this using the append operator.

Example 2. To guarantee the integrity of data, one can trigger controls after each data input. Considering a health care setting, for example, one could demand that, after inputting the results of an exam, the patient ID should be entered again to avoid mistakes. The realization of this requirement demands inserting a control after one activity. The corresponding algorithm thus replaces one activity with a pair activity and control. The XSLT template to enforce this control is as follows (we omit the actual call to the service realizing the control):

```
<xsl:template match=''bpel:assign[@actionType='ControlAfterEach']''>
 <bpel:sequence name=''Rule_Verify''>
  <bpel:assign name=''EnterResult'' policy:actionType=''Result''>
    <xls:apply-template/>
  </bpel:assign>
  <bpel:assign name=''VerifyPatientID''>
    Replace by actual control from compliance rule
  </bpel:assign>
 </bpel:sequence>
</xsl:template>
```

This template employs the namespace `policy`, with which action types can be assigned to activities. This facilitates the rewriting in processes where multiple occurrences of the same activity happen and are in the scope of a rule. ⊣

ReWrite envisaged evaluation. We plan to carried out an extensive evaluation of ReWrite, where qualitative and quantitative issues were of interest: firstly, which policies can be rewritten; secondly, what are the performance figures obtained in doing so. This section reports on the former.

To evaluate the expressiveness of ReWrite we plan to employ *workflow patterns* [29] and *compliance policy patterns* [13]. Specifically, we take a representative subset workflow patterns as the minimal process specification. These patterns can be seen as appropriate building blocks for process specifications and, hence, if rewriting succeeds for these patterns, it also succeeds for more complex specifications composed using workflow patterns. The compliance rules build upon the patterns of Dwyer et al. [13]. These patterns characterize a representative set of primitive structural requirements of programs, such as the absence, precedence and bounded existence of activities. However, the patterns can equally well be applied to business processes [3, 25]. More importantly though, the properties characterized by the patterns lie in the class of properties whose enforcement requires rewriting (see Sect. 2).

To actually assess the expressiveness of ReWrite, we need to determine which policy pattern can be added to which workflow pattern and, further, which could be alternatively enforced with an execution monitor (EM) and which demand rewriting (RW). Our goal is to demonstrate that each workflow pattern can be rewritten to comply with the corresponding policy pattern. Ongoing experiments deliver very promising preliminary results that substantiate this conjecture.

6 Related Work

Approaches to assessing business process compliance can be classified as [8, 25]: (1) *forward compliance checking* aims to design and implement processes where conforming behavior is enforced and (2) *backward compliance checking* aims to detect and localize non-conforming behavior. ReWrite is a forward compliance checking approach based on business process models and compliance policies. Related work addresses the annotation of business processes [15, 23], requirements elicitation [11, 10, 26] and formal verification [2–4, 6].

Program rewriting is a mechanism for enforcing security properties [16, 19]. It was initially formalized by Hamlen et al. [16] and he provided an implementation of a certified program-rewriting mechanism. These rewriting mechanisms are used to enforce low level properties such as type safety and do not consider business processes or high level concepts of modern programming languages. Similar approaches based on type systems to enforce certain security properties such as memory safety exist, among others, for Java bytecode [21], the .NET framework [14]. To our knowledge no approaches exist that investigate the transfer of automated rewriting techniques to business processes on a formal level.

The approaches discussed in the previous paragraph do not take into account the achievement of the programs' goals. The definition of some congruence relation is given, e.g. [19], but it is never explicitly specified in a way that it becomes possible to actually evaluate this relation for real world application.

De Backer and Snoeck discuss a concept called semantic compatibility which results in definitions of similar types of congruences [9]. As opposed to our approach, they are based on the languages defined by Petri Nets and they do not cover congruence of the processes themselves. They investigate how two processes that are deployed by different participants who need to achieve common business goals are able to cooperate. This notion of "compatibility" they devise is not applicable to the scenarios investigated in our research, because we discuss the business goals of a single process and not the interplay between two distinct processes in distinct administrative domains.

7 Summary and Further Work

This paper introduces a framework for rewriting business processes, thereby enforce security, compliance and privacy policies. Specifically, it motivates rewriting by showing that existing enforcement mechanisms cannot cope with a relevant class of properties. The paper then defines the syntax and semantics for a graphical process modeling language and that of compliance rules. In order to ensure that the rewritten process still allows for the execution paths of the original process and, thereby, achieves the original business goals, different congruence relations are defined. Process rewriting must then not only correct the process, but also preserve some of these relations (depending on the kind of operation).

Lessons learned. Rewriting is an established field in the theory of programs and logics. This paper carries over some of these concepts into business process compliance management. Although rewriting is in general undecidable for chains of modifications, this paper shows that it is possible to define primitive operators that allow for decidable procedures. The ReWrite framework thus opens up the possibility of automated correction of processes to ensure process compliance "by design" (for modeled or reconstructed processes) or during runtime.

Further work. Besides the ongoing and future work already indicated in the text, future work comprises four directions: *firstly*, on the formal side it is still necessary to investigate the congruence relations. Spefically, we need to flesh out the details of the most sutiable relation to cover all the operators. *Secondly*, we see a relationship between the techniques we employ and process repositories. That in the sense that the same technologies for querying could be employed to support rewriting. Clarifying this interplay is subject of further work. *Thirdly*, the kinds of policy supported by ReWrite can be generalized to also consider security properties and other usage control policies. Future work will tackle the expressive power of policies and corresponding analysis techniques (e.g. inconsistency detection). Fourthly, testing ReWrite for monitoring process instances.

References

1. R. Accorsi. Sicherheit im prozessmanagement. digma *Zeitschrift für Datenrecht und Informationssicherheit*, 2013.
2. R. Accorsi and A. Lehmann. Automatic information flow analysis of business process models. In *BPM*, volume 7481 of *LNCS*, pages 172–187. Springer, 2012.
3. R. Accorsi, L. Lowis, and Y. Sato. Automated certification for compliant cloud-based business processes. *BISE*, 3(3):145–154, 2011.
4. R. Accorsi and C. Wonnemann. Strong non-leak guarantees for workflow models. In *ACM SAC*, pages 308–314. ACM, 2011.
5. R. Accorsi, C. Wonnemann, and S. Dochow. SWAT: A security workflow toolkit for reliably secure process-aware information systems. In *ARES*, pages 692–697. IEEE, 2011.
6. R. Agrawal, C. Johnson, J. Kiernan, and F. Leymann. Taming compliance with sarbanes-oxley internal controls using database technology. In *ICDE*, pages 92–101. IEEE, 2006.
7. B. Alpern and F. Schneider. Defining liveness. *IPL*, 21(4):181–185, October 1985.
8. A. Antón, E. Bertino, N. Li, and T. Yu. A roadmap for comprehensive online privacy policy management. *CACM*, 50(7):109–116, July 2007.
9. M. D. Backer and M. Snoeck. Business process verification: a petri net approach. Open access publications from Katholieke Universiteit Leuven, 2007.

10. T. Breaux and A. Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE TSE*, 34(1):5–20, 2008.
11. T. Breaux, A. Antón, and E. Spafford. A distributed requirements management framework for legal compliance and accountability. *Computers & Security*, 28(1-2):8–17, 2009.
12. E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ICALP*, volume 623 of *LNCS*, pages 474–486. Springer, 1992.
13. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *IEEE CSE*, pages 411–420. ACM, 1999.
14. ECMA. Ecma-335: Common language infrastructure. european association for standardizing information and communication systems. Tech. Rep., ECMA, 2002.
15. A. Ghose and G. Koliadis. Auditing business process compliance. In *ICSOC*, volume 4749 of *LNCS*, pages 169–180. Springer, 2007.
16. K. Hamlen, G. Morrisett, and F. Schneider. Computability classes for enforcement mechanisms. *ACM TOPLAS*, 28(1):175–205, January 2006.
17. M. Hilty, D. Basin, and A. Pretschner. On obligations. In *ESORICS*, volume 3679 of *LNCS*, pages 98–117. Springer, 2005.
18. P. Jancar. Undecidability of bisimilarity for petri nets and some related problems. *Theor. Comput. Sci.*, 148(2):281–301, 1995.
19. R. Khoury and N. Tawbi. Corrective enforcement: A new paradigm of security policy enforcement by monitors. *ACM TISSEC*, 15(2):10, 2012.
20. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Logic and Algebraic Programming*, 78(5):293–303, May/June 2008.
21. T. Lindholm and F. Yelling. *The Java Virtual Machine*. Addison-Wesley, 1999.
22. N. Lohmann, E. Verbeek, and R. Dijkman. Petri net transformations for business processes - A survey. In *TPNOMC*, volume 5460 of *LNCS*, pages 46–63. Springer, 2009.
23. K. Namiri and N. Stojanovic. Using control patterns in business processes compliance. In *WISE*, volume 4832 of *LNCS*, pages 178–190. Springer, 2007.
24. A. Pretschner, F. Massacci, and M. Hilty. Usage control in service-oriented architectures. In *TRUSTBUS*, volume 4657 of *LNCS*, pages 83–93. Springer, 2007.
25. E. Ramezani, D. Fahland, and W. M. P. van der Aalst. Where did i misbehave? diagnostic information in compliance checking. In *BPM*, volume 7481 of *LNCS*, pages 262–278. Springer, 2012.
26. S. Sadiq, G. Governatori, and K. Namiri. Modeling control objectives for business process compliance. In *BPM*, volume 4714 of *LNCS*, pages 149–164. Springer, 2007.
27. D. Sangiorgi and D. Walker. *The pi-Calculus: A Theory of Mobile Processes*. Cambridge Press, 2001.
28. F. Schneider. Enforceable security policies. *ACM TISSEC*, 3(1):30–50, 2000.
29. W. van der Aalst. Workflow patterns. In *Encyclopedia of Database Systems*, pages 3557–3558. Springer, 2009.
30. W. van der Aalst. *Process Mining – Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
31. M. Weske. *Business Process Management: Concepts, Languages and Architectures*. Springer, 2011.
32. C. Wolter, M. Menzel, A. Schaad, P. Miseldine, and C. Meinel. Model-driven business process security requirement specification. *J. Systems Architecture*, 55(4):211–223, 2009.