Transforming SPEM 2.0-compatible Process Models into Models Checkable for Compliance

Julieth Patricia Castellanos Ardila, Barbara Gallina, and Faiz UL Muram

IDT, Mälardalen University
Box 883, 721 23 Västerås, Sweden
`{julieth.castellanos,barbara.gallina,faiz.ul.muram}@mdh.se`

Abstract. Manual compliance with process-based standards is time-consuming and prone-to-error. No ready-to-use solution is currently available for increasing efficiency and confidence. In our previous work, we have presented our automated compliance checking vision to support the process engineers work. This vision includes the creation of a process model, given by using a SPEM 2.0 (Systems & Software Process Engineering Metamodel)-reference implementation, to be checked by Regorous, a compliance checker used in the business context. In this paper, we move a step further for the concretization of our vision by defining the transformation, necessary to automatically generate the models required by Regorous. Then, we apply our transformation to a small portion of the design phase recommended in the rail sector. Finally, we discuss our findings, and present conclusions and future work.

Keywords: Software process, Compliance checking, Regorous, SPEM 2.0

1 Introduction

Claiming compliance with process-based standards requires that companies show, via the provision of a justification which is expected to be scrutinized by an auditor, the fulfillment of its requirements [1]. The manual production of this justification is time-consuming and prone-to-error since it requires that the process engineer checks hundreds of requirements [2]. A process-based requirement is checkable for compliance if there is information in the process that corroborate that the requirement is fulfilled [3]. This checking can be facilitated by using FCL (Formal Contract Logic) [4], a rule-based language that can be used to generate automatic support to reason from requirements and the description of the process they regulate. In our previous work [5], we have presented our automatic compliance checking vision (See Fig. 1). It consists of the combination of process modeling capabilities via SPEM 2.0 [6]-reference implementation, specifically by using EPF (Eclipse Process Framework) Composer [7], and compliance checking capabilities via Regorous [8], an FCL-based reasoning methodology, and tool.

In our vision, EPF Composer contributes with the appropriate (minimal set of) SPEM 2.0-compatible elements required by Regorous, which, in turn, produces a report that can be used to analyze and improve compliance.

In this paper, we define the transformation necessary (dotted line region shown in Fig. 1) to automatically generate the models required by Regorous, i.e., the FCL rule set, the structural representation of the process and the compliance effects annotations (cumulative interactions between process tasks that produce the desired global properties mandated by the standards [9]). Then, we apply our transformation to a small portion of the design phase recommended in the rail sector and discuss our findings.

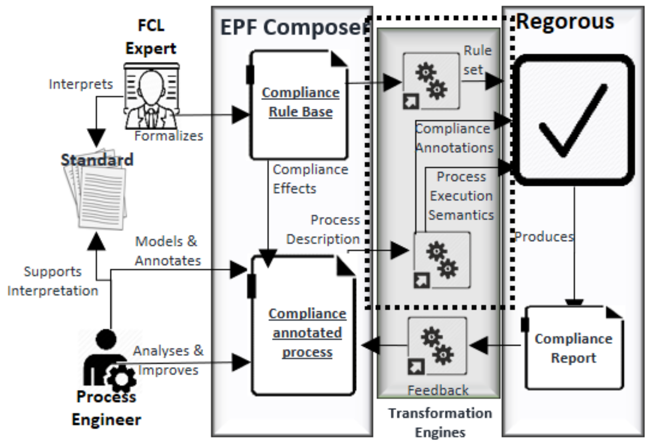


Fig. 1: Automated Compliance Checking Vision [5].

The rest of the paper is organized as follows. In Section 2, we recall essential background information. In Section 3, we present the transformations specification for generating Regorous inputs. In Section 4, we illustrate the transformation with a small example from the rail sector. In Section 5, we discuss our findings. In Section 6, we discuss related work. Finally, in Section 7, we derive conclusions and future work.

2 Background

In this section, we provide basic information on which we base our work.

2.1 EPF Composer

EPF Composer [7] is an open-source tool aiming at supporting the modeling of customizable software processes. We recall two open source standards used by EPF Composer and also required in this paper. **UMA** (Unified Method Architecture) Metamodel [10], a subset of SPEM 2.0 [6], is used to model and manage reusable method content and processes. Method Content defines the core elements used in a process, i.e., *tasks*, *work products* and *roles*. Managed

Content defines textual descriptions, such as *Concept* and *Reusable Asset. Custom Category* defines a hierarchical indexing to manage method content. A *delivery process* describes a complete and integrated approach for performing a specific project and it contains a *Breakdown Structure*, which allows nesting of tasks. **UML 2.0 Diagram Interchange Specification** [11] supports diagram interchange among modeling tools by providing an UML activity diagram representation. An *Activity* corresponds to a process, while a *Node* represents a point in the process, and an *Edge* is used to connects points. Nodes can be of different types. An *Activity Parameter Node* represents a task. *Initial and Final Nodes* represent the start and the end of the process. *Fork and Join Nodes* represent the parallel flows and *Decision and Merge Nodes* represent conditional behavior.

2.2 Regorous

Regorous [8] is a tool-supported methodology for compliance checking in which the compliance status of a process is provided with the causes of existing violations. To check compliance, Regorous requires a **rule set**, which is the formal representation of the standard's requirements in Formal Contract Logic (FCL) [4]. An FCL rule has the form $r : a_1, ..., a_n \Rightarrow c$, where r is the unique identifier, $a_1, ..., a_n$, are the conditions of the applicability of a norm and c is the normative effect. The different kind of normative effects can be found in [4]. A rule set is represented in the schema called *Combined Rule Set* from which we recall some elements. *Vocabulary* contains an element called *term*, which attribute *atom* is used to describe rule statements. The second element, called *Rule*, is used to define every rule of the logic. A rule is specified with the unique identifier called *label*, the description of the rule called *control objective*, and the actual rule called *formal representation*. Regorous current implemented tool uses the **Canonical Process Format (CPF)** [12], a modeling language agnostic representation that only describes the structural characteristics of the process. A *Canonical Process* is the container of a set of *Nets* which represent graphs made up of *Nodes* and *Edges*. Nodes types can be *(OR, XOR, AND) Splits/Joint*, which capture elements that have more than one incoming/outgoing edge. Nodes can also represent *Tasks* and *Events*, which are nodes that have at most one incoming/outgoing edge. The **compliance effect annotations**, which represents the fulfillment of a rule on a process element, are captured in Regorous by using a schema called *Compliance Check Annotations*. A *ruleSetList* contains the *ruleSets uri* which is the identification of the rule set. The *conditions* and the *taskEffects* represent the process sequence flow and the tasks respectively and have an associated *effects name* which corresponds to its actual compliance effects annotation.

2.3 Automatic Compliance Checking Vision: The Modeling Part

In this section, we recall the methodology used for modeling the SPEM 2.0-compatible models in EPF Composer required by Regorous. The methodology is explained with an example from ISO 26262 presented in [5]. The modeled requirement is obtained from part 6 clause 8, number 8.1, which states: *"Specify software units in accordance with the architectural design and the associated safety requirements"*. The formal representation of this requirement is presented in Equation 1.

$$\begin{aligned}r2.1: addressSwUnitDesignProcess \Rightarrow [OANPNP] - performSpecifySwUnit\\ r2.2: performProvideSwArchitecturalDesign, performProvideSwSafetyRequirements\\ \Rightarrow [P] performSpecifySwUnit\\ r2.2 > r2.1\end{aligned} \tag{1}$$

The modeling in EFP Composer required the creation of three plugins. Initially, we create a plugin for capturing standard's requirements (See Fig. 2), which contains not only their description in natural language e.g., *R2*, but also its atomization e.g., *r2.1* and *r2.2*. The requirement atomization is used to assign the rule representation (See Equation 1). A second plugin is used to capture process elements, as depicted in Fig. 3.

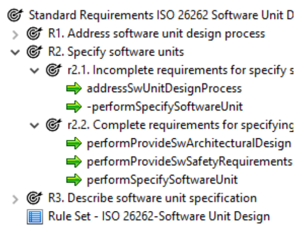

Standard Requirements ISO 26262 Software Unit D
R1. Address software unit design process
R2. Specify software units
r2.1. Incomplete requirements for specify s
addressSwUnitDesignProcess
-performSpecifySoftwareUnit
r2.2. Complete requirements for specifying
performProvideSwArchitecturalDesign
performProvideSwSafetyRequirements
performSpecifySoftwareUnit
R3. Describe software unit specification
Rule Set - ISO 26262-Software Unit Design

Fig. 2: Requirements Plugin.

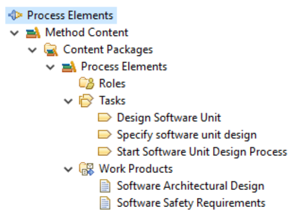

Process Elements
Method Content
Content Packages
Process Elements
Roles
Tasks
Design Software Unit
Specify software unit design
Start Software Unit Design Process
Work Products
Software Architectural Design
Software Safety Requirements

Fig. 3: Process Elements Plugin.

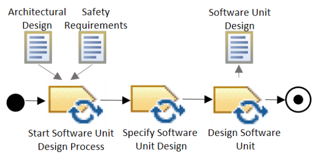


Fig. 4: Process Activity Diagram.

Finally, a third plugin is used to capture the compliance annotated tasks, in which we also create the delivery process and its corresponding activity diagram (See Fig. 4). To annotate the tasks, the concept that represents the compliance effects is added to the task. The reader can discover more details about the previous modeling in [5].

2.4 CENELEC EN 50128

CENELEC EN 50128 [13] is a standard that prescribes requirements for the development, deployment, and maintenance of safety-related software for railway control and protection application. The software component design phase is part of the lifecycle required by the software quality assurance, which states that the quality concerning the lifecycle shall address activities and tasks consistent with the plans (e.g., the safety plan). We recall some requirements corresponding to the software component design phase in Table 1.

Table 1: Requirements from the Rail Standard.

ID	Description
R1	Initiate component design phase.
R2	Input documents: Software Design Specification.
R3	A software component design shall be written under the responsibility of the designer.

3 Generating Regorous Inputs

In this section, we present the two steps required to generate Regorous inputs, namely the **mapping** between the elements provided by EPF Composer and required by Regorous, and their **algorithmic solution**. We start with the mapping of the elements required for creating the **rule set**. As presented in Table 2, the information related to the rules is obtained from the *Delivery Process* provided by EPF Composer (described with UMA elements), and should conform to the Regorous schema called *Combined Rule Set.* Then, in Table 3, we present the mapping required for the process structure, which is provided in a UML activity diagram and required to be transformed to the canonical process (CPF). Finally, the *compliance effects annotations* require a structure that complies to the Regorous schema called *Compliance Check Annotations.* This information can be retrieved from EPF Composer taking into account that the process elements can be extracted from the process structure (described with UML elements) and the compliance effects annotations can be extracted from the delivery process (described with UMA elements). The mapping is presented in Table 4.

Table 2: Mapping Elements from UMA to the Rule Set

UMA	Rule Set	Mapping Description
Reusable Asset	Rule Set	Reusable Asset is used in EPF composer to storage the information related to the rule set. Therefore, its information is transformed into the rule set required by Regorus. The attributes transferred are *name*, *presentationName* and *briefDescription.*
Concept	Term	Concept is used in EPF Composer to storage the information related to the vocabulary used in the creation of the rules. Therefore its content is transformed into the vocabulary required in the rule set, specifically, each Concept is a Term. The attribute transferred is *name.*
Content Category	Rule	Content categories contain rules. Therefore, their content is transformed into the body of the rule. The attributes transferred are *name*, *presentationName*, and *briefDescription.*

The algorithmic solution for obtaining the rule set, which mapping is described in Table 2, is presented in Algorithm 1. The algorithm initiates with the description of its required input (DeliveryProcess), and the expected output (RuleSet). Then, the input is parsed with the function *getElemementsByTagName*, which searches the elements to be mapped, with the function *Map* to the output. The first element searched is the *uma:ReusableAsset*, which attribute

Table 3: Mapping Elements from UML Diagram to the Canonical Process

UML	**CPF**	**Mapping description**
Activity	Canonical Process	The UML activity diagram is used in EPF Composer to describe the dynamics of the software process. Therefore, its information is transformed into a canonical process in CPF. The attribute transferred is *id*.
Initial Node	Start Event	The initial node of the activity diagram becomes a node with type start event in CPF.
Parameter Node	Task Type	Each parameter node in the activity diagram becomes a task type in the CPF. Attributes transferred are *id* and *name*.
Control Flow	Edge	Each control flow in the activity diagram becomes an edge in CPF. Attributes transferred are *id*, *name*, *source* and *target*.
Final Node	End Event	The final node in the activity diagram becomes an end event type in CPF.
Decision /Merge Node	XOR Split /Join	The decision/merge nodes in the activity diagram becomes an XOR-Split/XORJoin Type in CPF.
Fork/Join Node	AND Split/Join	The fork/join nodes in the activity diagram becomes an ANDSplit/AND-Join Type in CPF.

Table 4: Mapping from UMA/UML Metamodel to the Compliance Annotations

UML /UMA	**Compliance Annotations**	**Mapping Description**
Reusable Asset	ruleSet	A reusable asset becomes a ruleSetList. The attribute transferred is the *name*.
edge	conditions	Each edge becomes a special element in the compliance annotations file called condition. The attribute transferred is the id.
node	Task Effects	Each node becomes a Task Effect. The attribute transferred is the id. Then, the id is also used to search for the concepts that should be converted into the compliance effects in the delivery process file.
Concept	Effect	Every concepts associated to the task is transferred to the Effect. The attribute transferred is the name.

name is mapped to the rules URI. Then, the algorithm searches for the elements *uma:ContentCategory*, which provides the attributes *id*, *controlObjective* and *formalRepresentation* of each rule. Algorithm 2, which maps the elements described in Table 3, takes as input the UML Activity Diagram and provide the Canonical Format. The function *getElementsByTagName* searches for every elements that describes process structure and maps it to their counterpart in CPF. The mapping of the process structural elements requires a unique identifier that is generated internally each time the function *Map* is used. Algorithm 3 describes the solution for mapping the elements presented in Table 4. The required inputs are the UML Activity Diagram and the DeliveryProcess. The expected output is the ComplianceEffectsAnnotations. The algorithm searches in the delivery process the element tagged as *uma:ReusableAsset* and mapped it to the rule set. Similarly, the algorithm searches for the elements tagged as *uml:edge* and *uml:node* in the UML Activity Diagram and mapped them to the conditions and taskEffects respectively. The node id is used to search for the elements tagged as *uma:concept* in the DeliveryProcess, which is mapped to the effects.

```
input  : DeliveryProcess
output: RulseSet
LoadFile (DeliveryProcess);
NodeReusableAsset←getElementsByTagName (uma:ReusableAsset);
Map (ruleSet←ReusableAsset);
conceptsList←getElementsByTagName (uma:Concept);
for $i \leftarrow 0$to getLength (ConceptsList) do
    Map (Term.atom ←Concept.name)
end
contentCategoryList ← getElementsByTagName (uma:ContentCategory);
for $j \leftarrow 0$to getLength (contentCategoryList) do
    ruleControlObjective←getAttribute (briefDescription);
    if ruleControlObjective is not empty then
        Map (rule←contentCategory)
    end
end
```

Algorithm 1: Algorithm for Obtaining the Rule Set.

```
input  : UMLActivityDiagram
output: CanonicalFormat
LoadFile (UMLActivityDiagram);
NodeActivity←getElementsByTagName (uml:Activity) ;
Map (CanonicalProcess← NodeActivity);
nodesList←getElementsByTagName (uml:node);
for $i \leftarrow 0$ to getLength (nodesList) do
    if nodeType=uml:ActivityParameterNode then
        Map (TaskType←node)
    end
    if nodeType=uml:InitialNode then
        Map (StatEvent←node)
    end
    if nodeType=uml:ActivityFinalNode then
        Map (EndEvent←node)
    end
    if nodeType=uml:ForkNode then
        Map (ANDSplitType←node)
    end
    if nodeType=uml:JoinNode then
        Map (ANDJoinType←node)
    end
    if nodeType=uml:DecisionNode then
        Map (XORSplitType←node)
    end
    if nodeType=uml:MergeNode then
        Map (XORJoinType←node)
    end
end
edgesList←getElementsByTagName (uml:edge);
for $j \leftarrow 0$to getLength (edgeList) do
    Map (Edge←edge)
end
```

Algorithm 2: Algorithm for Obtaining the Process Structure.

```
input  : UMLActivityDiagram,DeliveryProcess
output: ComplianceEffectsAnnotations
LoadFile (UMLActivityDiagram, DeliveryProcess) ;
NodeReusableAsset (from DeliveryProcess)←getElementsByTagName (uma:ReusableAsset) ;
Map ((ruleSet←ReusableAsset) ;
edgesList(from UMLProcess)← getElementsByTagName (uml:edge);
for $i$ ← 0 to getLength (edgeList) do
    Map (conditions←edge)
end
nodeList(from UMLProcess)← getElementsByTagName (uml:node);
for $j$ ← 0 to getLength (nodeList) do
    Map (taskEffects←node) TaskId←ObtainUMAValue(nodeList);
    ContentElementList(from DeliveryProcess)←
      getElementsByTagName(ContentElement);
    for $k$ ← 0 to getLength (ContentElementList) do
        if ContentElementList.id = TaskId then
            ConceptsList(from DeliveryProcess)← getElementsByTagName(Concept);
            for $l$ ← 0 to getLength (ConceptsList) do
                Map (effects←Concept);
            end
        end
    end
end
```

Algorithm 3: Algorithm for Obtaining the Compliance Effects Annotations.

4 Models Checkable for Compliance from the Rail Sector

The purpose of this section is to provide evidence that the models provided by EPF Composer, and transformed with our algorithm, are checkable for compliance with Regorous. The software process model to be checked for compliance is the one modeled in Fig. 4 (originally created for compliance with an automotive standard). In this evaluation, three steps are required. Initially, we generate the compliance annotated software process in EPF Composer, following the methodology described in Section 2.3. Second, we apply the transformation described in Section 3. Finally, we verify that the models generated have enough information to be processed by Regorous. This verification is done manually, namely, we highlight the mapping of the elements required for checking compliance. We also check compliance with Regorous and describe the type of analysis that can be carried out after compliance checking.

We start by annotation a small portion of the design phase (modeled in Fig. 4) with the recommended requirements provided in the rail sector (see CENELEC requirements in Section 2.4). First, we formalize the standard's requirements applying the definitions for creating the rules presented in Section 2.2. As the formula 2) shows, the rule r1.1, which is the formalization of the requirement R1, defines an obligation of addressing the phase. Rules r.2.1 and r.2.2 are related to the requirement R2 in the following way: r.2.1 prohibits the specification of the design, but r.2.2 permits the specification of software units if the software design specification is obtained. Similarly to r.3.1 and r.3.2, which are related to requirement R3. Rule r.3.1 prohibits the production of software units, but r.3.2 permits them if not only the specification is performed but also is a designer has been assigned. In the previous rules, priority relations are defined to give higher priority to the permits over the obligations.

$$\begin{aligned}
&r1.1 : [OM]addressComponentDesignPhase\\
&r2.1 : addressComponentDesignPhase \Rightarrow [OANPNP] - performSpecifyComponentDesign\\
&r2.2 : obtainSoftwareDesignSpecification \Rightarrow [P]performSpecifyComponentDesign\\
&r3.1 : performSpecifyComponentDesign \Rightarrow [OANPNP] - produceSoftwareComponentDesign\\
&r3.2 : performSpecifyComponentDesign,\\
&assignDesigner \Rightarrow [P]produceSoftwareComponentDesign\\
&r2.2{>}r2.1, r3.2{>}r3.1
\end{aligned} \quad (2)$$

Standards requirements and the respective rules are modeled in EPF Composer in a plugin as depicted in Fig. 5.

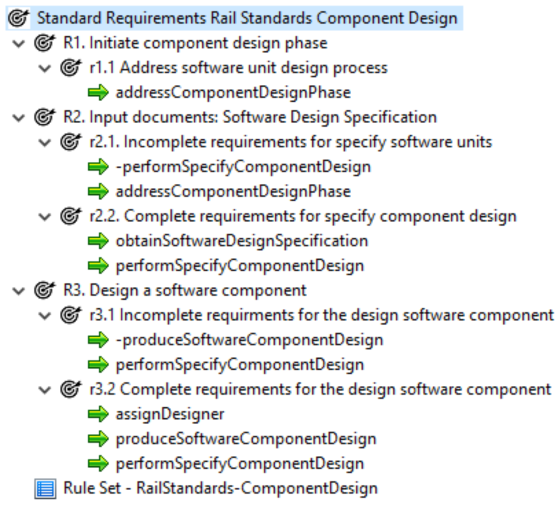

Standard Requirements Rail Standards Component Design
- R1. Initiate component design phase
  - r1.1 Address software unit design process
    - addressComponentDesignPhase
- R2. Input documents: Software Design Specification
  - r2.1. Incomplete requirements for specify software units
    - -performSpecifyComponentDesign
    - addressComponentDesignPhase
  - r2.2. Complete requirements for specify component design
    - obtainSoftwareDesignSpecification
    - performSpecifyComponentDesign
- R3. Design a software component
  - r3.1 Incomplete requirments for the design software component
    - -produceSoftwareComponentDesign
    - performSpecifyComponentDesign
  - r3.2 Complete requirements for the design software component
    - assignDesigner
    - produceSoftwareComponentDesign
    - performSpecifyComponentDesign
- Rule Set - RailStandards-ComponentDesign

Fig. 5: Requirements Plugin.

Then, we import the plugin that contains the process elements (See Fig. 3). Finally, we create the plugin for annotating the process tasks. In this plugin, we copy the tasks from the plugin that contains the process elements and make them *contribute* to the original ones, which allows to extend them in an additional way. The tasks are annotated according to the compliance effects they represent. For this, we check the process model depicted in Fig. 4. As we see, the task *Start Software Unit Design Process* represents the initiation of the software component design and therefore it produces the compliance annotation *addressComponentDesignPhase*. This task also responds to the compliance effect *obtainSoftwareDesignSpecification* since it has a work product with a similar name. Task *Specify Software Units* responds to the compliance effect *performSpecifyComponentDesign*. Finally, the task *Design Software Unit* has a work product *Software Unit Design*, which makes the task respond to the compliance effect *produceSoftwareComponentDesign*. Once the tasks are annotated, we create the delivery process and the activity diagram, export the plugins and apply the transformations to obtain the Regorous inputs to check compliance.

In what follows, we provide essential code snippets, in which we highlight the mapping of the elements required for checking compliance. We start showing the generated *Rule Set*. As presented in Listing 1.1, the generated *Rule Set* has the elements *Vocabulary*, which contains the rules, described in EPF Composer with an *uma:concept*. It also contains the *rules*, which were described in the content category elements that correspond to the rules.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RuleSet xmlns="http://www.nicta.com.au/bpc/CombinedRuleSetDefinition
    /0.1" uri="RuleSetRailStandards" >
   <Vocabulary>
      <Term atom="addressComponentDesignPhase"/>
      ...<!--other Term atoms -->
   </Vocabulary>
   <Rules>
      <Rule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:type="DflRuleType" ruleLabel="r1.1">
         <ControlObjective>r1.1 Address software unit design process</
             ControlObjective>
         <FormalRepresentation>=&gt;[OANPP] addressComponentDesignPhase</
             FormalRepresentation>
      </Rule>
      ...<!--other rules -->
   <SuperiorityRelations>
      ...
   </SuperiorityRelations>
</RuleSet>
```

Listing 1.1: Rule set generated

In Listing 1.2, we present the generated process structure. We highlighted one *Node* that represents the start point of the process and one node that represents a task Type. An *Edge* represents a connection between the nodes.

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<ns4:CanonicalProcess name="Software Unit Design Process" ...>
   <Net id="1529072497607">
      <Node id="1529072497608" xsi:type="ns4:EventType" xmlns:xsi="http:
          //www.w3.org/2001/XMLSchema-instance">
         <name>Start</name>
         <attribute value="startevent1" typeRef="Id"/>
      </Node>
      <Node id="1529072497609" xsi:type="ns4:TaskType" xmlns:xsi="http:
          //www.w3.org/2001/XMLSchema-instance">
         <name>Start Software Design Process</name>
         <attribute value="StartSoftwareDesignProcessID" typeRef="Id"/>
      </Node>
      ...<!--other nodes -->
      <Edge id="1529072497612" targetId="1529072497609" sourceId="
          1529072497608" default="false">
      ...<!--other edges -->
   </Net>
</ns4:CanonicalProcess>
```

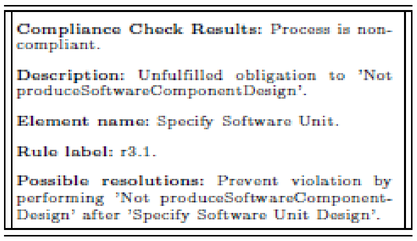
Listing 1.2: Process structure generated

In Listing 1.3, we present the compliance annotations. For example, the *rule set uri* is the rule set identification, *conditions element id* represent control flows identification, and the *taskEffects* represent the tasks, which *effects name* corresponds to the effects.

```
<?xml version="1.0" encoding="ASCII"?>
<cca:ComplianceAnnotations xmi:version="2.0" xmlns:xmi="http://www.omg.
     org/XMI" xmlns:cca="http://www.nicta.com.au/bpc/eclipse/
     ComplianceCheckAnnotations">
<ruleSetList>
   <ruleSets uri="RuleSetRailStandards"/>
</ruleSetList>
   <conditions elementId="_jNj1AExVEeiW4M4duzOA6Q"/>
   <conditions elementId="_jukQUExVEeiW4M4duzOA6Q"/>
   ...<!--other conditions-->
   <taskEffects elementId="_hCKUcExVEeiW4M4duzOA6Q">
      <effects name="addressComponentDesignPhase" negation="false">
      <effects name="obtainSoftwareDesignSpecification" negation="false">
   ...<!--other taskEffects -->
   <localVocabulary/>
</cca:ComplianceAnnotations>
```

Listing 1.3: Compliance annotations generated

Then, we checked compliance with Regorous. The report results (See Fig. 6) not only shows that the *process in non-compliant*, but also the description of the uncompliant situation, the element that may be the source of the violation, the rule that has been violated and the possible resolution. With this information, it may be easier for the process engineer to make a focused analysis to improve the compliance status. In the example, the rule 3.1 (highlighted in Fig. 5), refers to *Incomplete requirements for the design of software Components*, which means that we do not have the requirements in place to address the task called *Specify Software Unit Design*. To solve the uncompliant situation, we refer to the counterpart rule, which is the one marked as *r.3.2*, in which the compliance effects *assign designer* and *produceSoftwareComponentDesign* and *performSpecifyComponentDesign* are included. To be able to complete the assignment of these effects, we need to include a role called *designer* to the task *Specify Software Unit Design* as presented in Fig. 7. The improved process is again checked, resulting in a report with no violations of the rules.


Compliance Check Results: Process is non-compliant.

Description: Unfulfilled obligation to 'Not produceSoftwareComponentDesign'.

Element name: Specify Software Unit.

Rule label: r3.1.

Possible resolutions: Prevent violation by performing 'Not produceSoftwareComponent-Design' after 'Specify Software Unit Design'.

Fig. 6: Compliance Report.

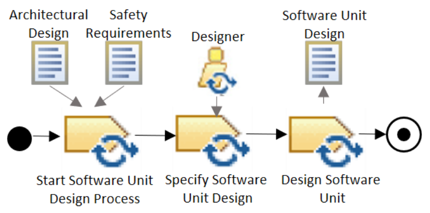


Fig. 7: Activity Diagram.

5 Discussion

Automated compliance checking of software processes with Regorous generates a compliance report that not only communicate the compliance status of the software process, i.e., whether the process is compliant or not, but also the sources of violations, i.e, the rules that have being violated and the target of the uncompliant situations (specific tasks), and possible resolutions. This information may increase **efficiency** in the process compliance since it permits the process engineer to focus on specific process elements and the reparation policies they may require. In the example presented in Section 6, it was clear, from the compliance report, that the task affected was *Specify Software Unit* and we focus on it to understand the missing process elements. If rules are correctly formalized, and their formalization covers the standards requirements entirely, also **confidence** can be increased since uncompliant situations, in all the levels, would be spotted. Since we have modeled in detail the requirements provided in Table 1, we can consider the checking of the small process reliable. A software process can be checked for compliance with different standards. This specific aspect could potentially be beneficial since it promotes **process reusability**, i.e., a process engineer can take processes designed in previous projects, check their compliance status with the normative requirements of the new project and improve it, based on the violations reported. In the example, we saw that the software process model created for automotive could be used as a base for model a small portion of the design phase recommended in the rail sector.

As we see in Fig. 1, the adopted methodological approach for our automatic compliance checking vision, is tool supported. While the **maturity** of the methodology is high, its tool support still requires additional work. EPF Composer and Regorous have been tested separately and the bridge between them, namely, the transformations between the EPF Composer and Regorous, have been designed and implemented. The transformations, applied to the portion of the design phase recommended in the rail sector (See Section 4), are *correct* since they have generated a complete set of inputs that are compatible with Regorous schemas, making possible to check compliance. The transformation implementation, which is still in a prototyping stage, could be improved if techniques, such as Model Driven Engineering (MDE) are applied. We consider essential to further exploit the process modeling language agnosticism underlying Regorous methodology to be able to perform a future seamless integration of the tools required for our compliance checking vision.

6 Related Work

Automatic compliance checking of processes is one of the mechanisms that can provide benefits, as we have discussed above, to compliance management. In particular, researchers in the business and legal compliance context have explored potential formalisms to create compliance checking frameworks, such as the ones presented in [14] and in [15]. However, they are based on temporal logics, in which the modeling of normative requirements is still considered difficult.

To model the rules more naturally, we have chosen Regorous, which underlying formalism called FCL, permits the modeling of deontic notions (i.e., obligations, prohibitions and permissions) which are the actual notions that describe normative requirements. Automatic compliance checking of safety-critical software processes has not been as explored as in business management. However, in [16], the authors presented initial steps of an approach for process reasoning and verification, which is based on the combination of Composition Tree Notations (CTN), a high-level modeling notation used for modeling process structure, and Description Logics (DL). DL is used to reason about the compliance of the process structure. Instead, our approach includes the accumulation of compliance effects that trigger new effects, focusing on the process behavior. Another difference we have included in our approach is the use of SPEM 2.0-compatible software process models, which may be preferred over other process modeling languages since it allows the creation of process method contents that can be reused in different kind of processes. SPEM 2.0-related community, to the best of our knowledge, has not addressed compliance checking. However, based on SPEM 2.0, some solutions for compliance management exists. In [3], compliance tables are generated. Compliance tables require the modeling of the standard's requirements, which should be mapped to the process elements that fulfill them. The modeling of compliance elements is also exploited in [17], in which the modeling of standards requirements is required to detect whether the process model contains sufficient evidence for supporting the requirements. The approach provides feedback to the safety engineers regarding detected fallacies and recommendations to solve them. In our case, we have also exploited not only the modeling of standard requirements, but also we have provided a mechanism to include rules within the standard's requirements, which facilitate the resolution of uncompliant situations after the automatic compliance checking is performed.

7 Conclusions and future work

In this paper, we defined the transformation necessary to automatically generate the models checkable for compliance in Regorous from SPEM 2.0-compatible process models. We also applied our transformation to a small portion of the software component design phase recommended in the rail sector and discussed aspects related to our findings.

To increase the maturity of the results shown in this paper, a proper plugin is going to be implemented to enable the push-button solution for the entire generation of the inputs required by Regorous. Also, as presented in [5], we need to further validate our approach and complete some tasks, i.e., the addition of the rule editor to facilitate the modeling of FCL rules, which currently is done manually, and the mechanism to back-propagate compliance results into EPF Composer. This work is expected to be partly delivered within the final release of the AMASS platform [18].

Acknowledgments. This work is supported by the EU and VINNOVA via the ECSEL JU project AMASS (No. 692474) [19]. We thank Guido Governatori for his guidance during the execution of this project.

References

1. Gallina, B., Ul Muram, F., Castellanos Ardila, J.: Compliance of Agilized (Software) Development Processes with Safety Standards: a Vision. In: 4th international workshop on Agile Development of Safety-Critical Software. (2018)
2. Castellanos Ardila, J., Gallina, B.: Towards Increased Efficiency and Confidence in Process Compliance. In: 24th European Conference EuroSPI. (2017) 162–174
3. McIsaac, B.: IBM Rational Method Composer: Standards Mapping. Technical report, IBM Developer Works (2015)
4. Governatori, G.: Representing business contracts in RuleML. International Journal of Cooperative Information Systems. (2005) 181–216
5. Castellanos Ardila, J.P., Gallina, B., Ul Muram, F.: Enabling Compliance Checking against Safety Standards from SPEM 2.0 Process Models. In: Euromicro Conference on Software Engineering and Advanced Applications. (2018)
6. Object Management Group Inc.: Software & Systems Process Engineering Meta-Model Specification. Version 2.0. OMG Std., Rev (2008) 236
7. The Eclipse Foundation.: Eclipse Process Framework (EPF) Composer 1.0 Architecture Overview. http://www.eclipse.org/epf/composer_architecture/ (2013)
8. Governatori, G.: The Regorous approach to process compliance. In: IEEE 19th International Enterprise Distributed Object Computing Workshop. (2015) 33–40
9. Koliadis, G., Ghose, A.: Verifying Semantic Business Process Models in Verifying Semantic Business Process Models in Inter-operation. In: IEEE International Conference on Service-Oriented Computing. (2007) 731–738
10. IBM Corporation: Key Capabilities of the Unified Method Architecture (UMA)
11. Object Management Group: UML 2 . 0 Diagram Interchange Specification. (2003)
12. La Rosa, M., Reijers, H., van der Aalst, W., Dijkman, R., Mendling, J., Dumas, M., García-bañuelos, L.: APROMORE: An advanced process model repository. Expert Systems With Applications (2011) 7029–7040
13. EN50128 BS: Railway applications Communication, signalling and processing systems Software for railway control and protection systems (2011)
14. Elgammal, A., Turetken, O., van den Heuvel, W., Papazoglou, M.: Formalizing and applying compliance patterns for business process compliance. Software and Systems Modeling. (2016) 119–146
15. El Kharbili, M.: Business Process Regulatory Compliance Management Solution Frameworks: A Comparative Evaluation. 8TH Asia-Pacific Conference on Conceptual Modelling. (2012) 23–32
16. Kabaale, E., Wen, L., Wang, Z., Rout, T.: Representing Software Process in Description Logics: An Ontology Approach for Software Process Reasoning and Verification. In: Software Process Improvement and Capability Determination. SPICE 2016. Communications in Computer and Information Science. (2016) 362–376
17. Ul Muram, F., Gallina, B., Gomez Rodriguez, L.: Preventing Omission of Key Evidence Fallacy in Process-based Argumentations. In: 11th International Conference on the Quality of Information and Communications Technology. (2018)
18. AMASS Platform: https://www.polarsys.org/opencert/
19. AMASS: Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems. http://www.amass-ecsel.eu/