IMPROVING THE CONSISTENCY OF SPEM-BASED SOFTWARE PROCESSES

Eliana B. Pereira, Ricardo M. Bastos, Michael da C. Móra
Faculty of Informatics, Pontifical University Catholic of Rio Grande do Sul, Porto Alegre, Brazil

Toacy C. Oliveira
COPPE Department, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil

Keywords: Software process metamodel, Process checking, SPEM, Well-formedness rule.

Abstract: The main purpose of this paper is to improve the consistency of Spem-Based Software Processes through a set of well-formedness rules that check for errors in a software process. The well-formedness rules are based on the SPEM 2.0 metamodel and described using the *Unified Modeling Language* - UML multiplicity and *First-Order Predicate Logic* - FOLP. In this paper, the use of the well-formedness rules is exemplified using a part of the *OpenUP* process and the evaluation of the one of the proposed rules is shown.

1 INTRODUCTION

Software development is ultimately a procedure to convert informal specifications, typically gathered from real world scenarios, into formal pieces of code that can be executed by machines. Such a procedure is mainly enacted by developers that follow an orchestrated path from analysis, through coding and testing. Orchestration emerges from a software process specification that details how process elements such as roles, tasks and work products, are interconnected in an organized manner (Jacobson *et al.*, 2001). Although developers can find off-the-shelf software process specifications such as *Rational Unified Process* - RUP (Kruchten, 2000) and *Object-Oriented Process, Environment and Notation* - OPEN (Open, 2006), there is no "one size fits all" process, which means a process must be defined to meet each project's needs (Xu and Ramesh, 2003).

To define a software process it is necessary to consider project's constraints such as team, resources, technology and time-to-market, to create the fabric of interconnected process elements that will guide software development (Jacobson *et al.*, 2001). Typically, software process engineers combine elements from "off-the-shelf" processes, since they represent best practices in the software engineering discipline. Software process engineers are also assisted by *Situational Method Engineering* - SME. SME recommends creating a set of method fragments or method chunks (pieces of processes) where each one of these fragments or chunks describes one part of the overall method (in this paper called software process). Each software project starts with a process definition phase where the method fragments or chunks are selected and organized to attend the specific needs related to the project (Henderson-Sellers *et al.*, 2008).

Regardless the strategy used to define a software process specification, it is important to understand the associated complexity of interconnecting the process elements that will be used to maximize the outcome of a software project. Typically a process specification interconnects dozens, sometimes hundreds, of process elements and any inconsistency in the process will negatively impact on how developers perform. Inconsistent processes have several forms. For example, inconsistency may appear when a task requires information that is not produced by any other task; when two or more work products duplicate information; or even when tasks are sequenced in cycles. These problems are hard to identify if no automated approach is adopted.

According to (Hug *et al.*, 2009), as software processes are based on process models, which are directed by concepts, rules and relationships, a metamodel becomes necessary for instantiating these

Pereira E., Bastos R., da C. Móra M. and Oliveira T..
IMPROVING THE CONSISTENCY OF SPEM-BASED SOFTWARE PROCESSES .
DOI: 10.5220/0003501100760086
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 76-86
ISBN: 978-989-8425-55-3

process models. Meta-modeling is a practice in software engineering where a general model (metamodel) organizes a set of concepts that will be later instantiated and preserved by specific models (instances). In this scenario, a software process metamodel could represent basic interconnection constraints that should hold after the metamodel is instantiated (Henderson-Sellers and Gonzalez-Perez, 2007), thus minimizing inconsistencies. An evidence of the importance of metamodels for software processes is the existence of metamodels such as *Software & Systems Process Engineering Meta-Model Specification* - SPEM 1.1 (OMG, 2002), *OPEN Process Framework* - OPF (Open, 2006), among others. Recently the *Object Managements Group* – OMG issued a new version of its standard for Process Modeling, namely SPEM 2.0, which offers the minimal elements necessary to define any software process (OMG, 2007).

Although the SPEM 2.0 metamodel represents a great advance in software process specification and consistency, its use is not straightforward. SPEM 2.0 defines several concepts using the UML class diagram notation and represents several constraints with natural language. For example, SPEM 2.0 allows the specification of a *Task* that does not consume, produce and/or modify any *Work Product*. This is clearly an inconsistency once a *Task* has a purpose, expressed in terms of creating or updating *Artifacts* (*Work Products*) (Kruchten, 2000).

In order to improve the consistency of the software processes instantiated from SPEM 2.0 this paper proposes a set of well-formedness rules to check for the software processes consistency. The focus of this paper is only the consistency of the roles, work products, tasks and their relationships. Each well-formedness rule expresses a condition that must be true in all software process instances. To create the well-formedness rules we have started our work by redefining some relationships in the SPEM 2.0. For those more elaborated well-formedness rules we have used FOLP.

The paper is organized as follows: Section 2 presents the related works. Section 3 describes the SPEM 2.0. Section 4 presents some packages of SPEM 2.0. In Section 5, the consistency well-formedness rules are shown. Section 6 evaluates some well-formedness followed by the conclusions.

2 RELATED WORK

Several papers have focused on defining software process from a process metamodel. Some approaches (Puviani, 2009), (Habli and Kelly, 2008), (Serour and Henderson-Sellers, 2004), (Bendraou *et al.*, 2007) propose solutions using well known metamodels such as OPF or SPEM, while others define their own process metamodels (Wistrand and Karlsson, 2004), (Gnatz *et al.*, 2003), (Ralyte *et al.*, 2006).

In (Puviani, 2009), (Serour and Henderson-Sellers, 2004), (Wistrand and Karlsson, 2004) and (Ralyte *et al.*, 2006) the authors consider metamodels to define method fragments, method chunks or method components. Although they differ in terminology, fragments, chunks or components, represent small elements of a software process. This approach is known as *Situational Method Engineering* - SME, which is a subset of the *Method Engineering* - ME discipline. According to (Henderson-Sellers *et al.*, 2008), SME provides a solid basis for creating software process. Chunks, fragments or components are typically gleaned from best practice, theory and/or abstracted from other processes. Once identified and documented, they are stored in a repository, usually called method base (Henderson-Sellers and Gonzalez-Perez, 2007).

In (Bendraou *et al.*, 2007) the authors propose an extension to SPEM 2.0 to address the lack of the "executability" of this metamodel. The objective of the extended metamodel is to include a set of concepts and behavioural semantics. In (Habli and Kelly, 2008) the authors present a process metamodel that embodies attributes to facilitate the automated analysis of the process, revealing possible failures and associated risks. The metamodel allows associating risks to the activities and mitigates them before they are propagated into software product. Gnatz *et al.* (2003) also propose a metamodel to define software process. The authors are mainly interested in performing process improvement together with static and dynamic tailoring (adjustment) of process models.

Though process metamodels are used by many research groups, the software process consistency issue is not widely explored. Most works lack rules to check the consistency of the created software processes. Specifically related to the software process consistency some few works might be found in the literature. Bajec *et al.* (2007), which describe an approach to process configuration, present some constraint rules in their work to constrain some aspects of the software process construction. The authors decompose their rules in four subgroups: *process flow rules*, *structure rules*, *completeness rules* and *consistency rules*. The *completeness rules* and *consistency rules* are related to this work since these rules are derived from a process metamodel. According to (Bajec *et al.*, 2007), the completeness rules help to check whether a software process includes all required components. To the authors these rules can be specified in a simple manner using attributes in the metalink class, which is equivalent to multiplicities in the association relation in UML. An example of the completeness rule in (Bajec *et al.*, 2007) is that *each activity must be linked with exactly one role.* The consistency rules are considered by the authors similar to completeness rules. Their goal is to assure that the selection of the elements to a process is consistent. While completeness rules only apply to elements that are linked together, consistency rules deal with interdependency between any two elements. An example of the consistency rule is *each artifact depends on at least one production activity*.

Hsueh *et al.* (2008) propose an UML-based approach to define, verify and validate software processes. The authors consider UML as the modeling language to define the processes and work with class diagram to model the process static structure, the state diagram to model the process element's behavior and the activity diagram to model the process sequence. For the process structure they describe a process metamodel based on UML 2.0 and present some rules in *Object Constraint Language* - OCL. Conceptually, that work is related to this one as it considers a process metamodel and some formalized rules to help model verification. However, there are some important differences. In (Hsueh *et al.*, 2008), the correctness, completeness and consistency of a process are verified by only checking the class multiplicities. All their OCL rules are CMMI-related rules and are used to verify if the software process meet the requirements of CMMI.

Atkinson *et al.* (2007) propose using an existing *Process Modeling Language* - PML to define process. Although the authors do not consider a metamodel they present a set of rules related to the process consistency. They also present a tool, *pmlcheck*, used to check a process before performing it. Basically, the consistency rules implemented in *pmlcheck* are related to the actions (the tasks of SPEM 2.0) and resources (the work products of SPEM 2.0). Rules to check errors related to action requirements are implemented. These types of rules check four errors: actions consuming and producing no resources, actions only consuming resources, actions only producing resources and actions modifying a resource that they were not consuming. There are also rules to trace dependencies through a process. These rules are: checking if resources required by an action are produced in an earlier action and checking if produced resources are consumed by at least one action.

Besides the studies above, we consider our work similar to the works about UML model consistency. Although, usually, these works are interested in consistency issues between the various diagrams of an UML specification they also consider the UML language and the consistency aspect. Additionally, in their majority, they describe formal approach (Lucas *et al.*, 2009), what we have also been done.

3 SPEM 2.0

The SPEM 2.0 metamodel is structured into seven packages. The structure divides the model into logical units. Each unit extends the units it depends upon, providing additional structures and capabilities to the elements defined below. The first package is Core that introduces classes and abstractions that build the foundation for all others metamodel packages. The second package, the Process Structure, defines the base for all process models. Its core data structure is a breakdown or decomposition of nested Activities that maintain lists of references to perform *Role* classes as well as input and output *Work Product* classes for each *Activity*. The Managed Content package introduces concepts for managing the textual content of a software process. The Process Behaviour package allows extending the structures defined in the Process Structure package with behavioural models. However, SPEM 2.0 does not define its own behaviour modelling approach. The Method Content package provides the concepts to build up a development knowledge base that is independent of any specific processes. The Process with Methods package specifies the needed concepts to integrate the Process Structure package and Method Content package. Finally, the Method Plugin package allows managing libraries and processes.

SPEM 2.0 is expressed using *MetaObject Facility* - MOF 2.0 meta-modeling language. Figure 1 shows the use of MOF 2.0 and UML 2.0 for modelling and defining SPEM 2.0. The Figure shows different instantiation layers of the formalism used for the SPEM 2.0 specification. MOF is the universal language that can be used on any layer, but in our case MOF is instantiated from the M3 layer by SPEM 2.0 on the M2 layer. The UML 2 meta-model itself, as depicted on the right-hand side of the M2 layer, instantiates MOF defined on M3 layer in the same way. Finally, process models can be instantiated using the M1 layer. In Figure 1, "Method Library" is shown as an example of a concrete instance of SPEM 2.0. In that sense, SPEM 2.0 defines process elements such as *Tasks* and *WorkProducts* as well as relationships among them whereas Method Library provides the concrete instance to these elements.

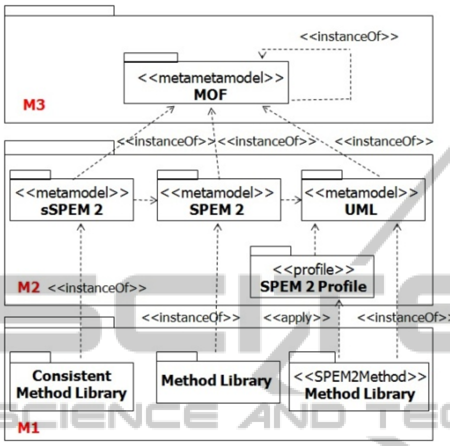

Figure 1: Specification Levels.

The consistency well-formedness rules proposed were defined in the M2 layer. They are based on the elements and relationships of the Process Structure and Process with Methods packages. In Figure 1 we have also represented how our proposal is located in the instantiation layers. In the left-hand side of the M2 layer, the sSPEM 2.0, which stands for *conSistent SPEM 2.0*, has all content of SPEM 2.0 more our consistency well-formedness rules. The sSPEM 2.0 is also an instance of MOF and it may be instantiated using the M1 layer. In Figure 1 the "Consistent Method Library" is shown as an instance of the sSPEM 2.0. It means that the "Consistent Method Library" has concrete instances of the elements and relationships of the SPEM 2.0 which were checked using the consistency well-formedness rules of the sSPEM 2.0.

4 PROCESS DEFINITION

This section explores the main SPEM 2.0 packages and introduces our proposal for process checking.

4.1 Process Structure in the SPEM 2.0

In SPEM 2.0 the main structural elements for defining software processes are in the Process Structure package. In this package, processes are represented with a breakdown structure mechanism that defines a breakdown of *Activities*, which are comprised of other *Activities* or leaf Breakdown Elements such as *WorkProductUses* or *RoleUses*. Figure 2 presents the Process Structure metamodel.

The *ProcessPerformer*, *ProcessParameter*. *ProcessResponsabilityAssignment* and *WorkProductUseRelationship* classes are used to express relationships among the elements in a software process. The *WorkSequence* class also represents a relationship class. It is used to represents a relationship between two *WorkBreakdownElements* in which one *WorkBreakdownElement* depends on the start or finish of another *WorkBreakdownElement* in order to begin or end. Another important process element which is not defined in the Process Structure package is the *Task*. This element is defined in the Process with Methods package which merges the Process Structure package. A task describes an assignable unit of work. In the *Process with Methods* package the class that represents the task element is the *TaskUse* class which is a subclass of the *WorkBreakdownElement* class of the Process Structure package. Figure 3 shows the relationships for the *TaskUse* class which are defined in the Process with Methods package.

Basically, the *TaskUse* class has relationships with the same elements as the *Activity* class. Figure 3 also shows that both the *TaskUse* class as well the *RoleUse* and *WorkProductUse* classes have, respectively, relationships with *TaskDefinition*, *RoleDefinition* and *WorkProductDefinition* classes. These classes are defined in the Method Content package and are used in the Process with Method Package by the merge mechanism.

All software process may use the concepts defined in the Method Content by creating a subclass of *Method Content Use* class and reference it with a subclass of *Method Content Element* class. The *Method Content Element* and *Method Content Use* classes are defined, respectively, in the Method Content package and Process with Methods package. All software process may use the concepts defined in the Method Content by creating a subclass of *Method Content Use* class and reference it with a subclass of *Method Content Element* class. *RoleUse, WorkProductUse* and *TaskUse* are subclasses to the *Method Content Use* class and *RoleDefinition*, *WorkProductDefinition* and *TaskDefinition* are subclasses to the *Method Content Element* class.

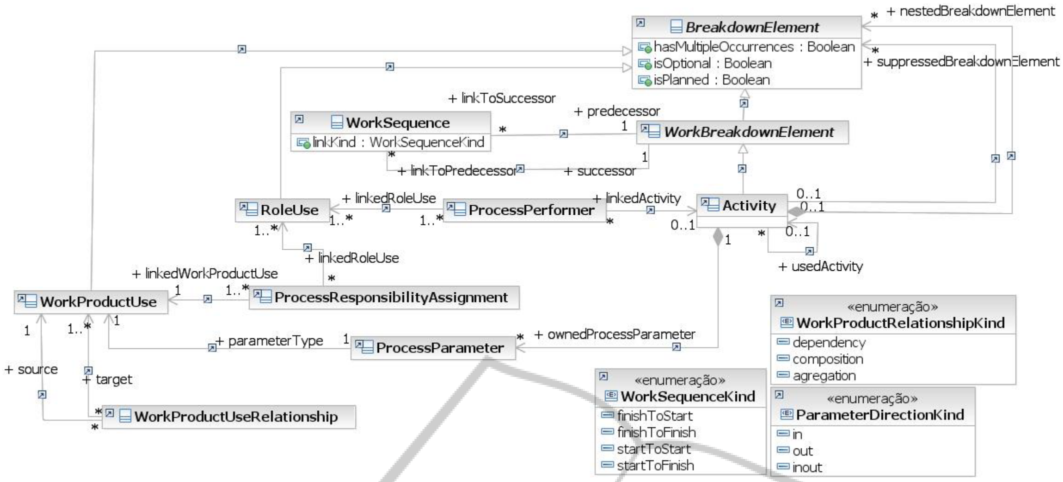

Figure 2: Process Structure Metamodel.

It is important to consider that both models presented in Figure 2 and Figure 3 had some multiplicities modified from the SPEM original metamodel. This is so because these models already represent models of sSPEM 2.0 and include some well-formedness rules proposed in this paper (which will be explained in Section 5).

4.2 Errors in a Software Process

We consider that errors in a process are motivated mainly by the following two reasons: (1) process metamodels are typically specified with UML class diagrams, which are only capable of representing simple multiplicity constraints. As a result they need an external language such OCL or Natural Language to represent complex restrictions. As with SPEM 2.0, most constraints are represented in Natural Language, which can lead to interpretation errors; and (2) software process metamodels are usually composed by several elements as they must represent activity workflows, information flows and role allocations. As a result, using a process metamodel can be cumbersome as the user must deal with several concepts to represent a process.

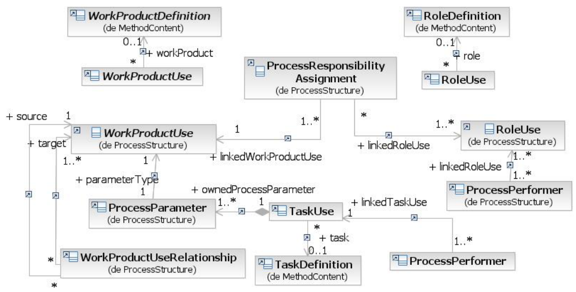

Figure 3: Relationships of the TaskUse Class.

According to (Atkinson *et al.*, 2007), the errors in a software process are most often introduced by a modeller and related to syntax or typographical mistakes that affect the process consistency. A modeller might, for example, make a simple error by connecting a work product that still was not produced in the software process as an input in a task. It would break a dependency because the task was expecting an unavailable work product.

To avoid errors in a process we propose checking it before enactment. Process checking is the activity of verifying the correctness and the consistency of a process. In this paper, process checking is made from a set of well-formedness rules specified from the SPEM 2.0 metamodel. The well-formedness rules are associated with the metamodel classes and relationships which represent the process elements and their relations. Every instance of process elements and relationships that have one or more associated well-formedness rules is checked. If violated, error messages appear. In the next section, we explain our well-formedness rules. Some rules are expressed using UML multiplicity and others, which involve more elements and/or express more elaborated rules, are described in FOLP.

5 PROCESS CHECKING

In this section we describe a set of well-formedness rules related to software process correctness and consistency. We propose using these rules for process checking. The well-formedness rules from this research were defined considering the concepts defined in the Process Structure and Process with Methods packages of SPEM 2.0 metamodel.

Although the Method Content package has also important concepts for software process it only defines reusable content which is used through the classes of the Process with Methods package.

5.1 Well-formedness Rules

As the SPEM metamodel is represented by UML class diagrams we consider that many constraints already exist in this metamodel through the multiplicity used between the classes. The following rule is one that is already defined in the SPEM 2.0 metamodel and constraints process multiplicity: a *Process Performer* must be associated to exactly one *TaskUse*. There is a "*linkedTaskUse*" relationship between *TaskUse* and *Process Performer* classes. The multiplicity is constrained to have only one relationship.

Considering all multiplicities defined between the classes of the Process Structure and Process with Methods packages we have noted that inconsistencies may be introduced into a software process. For example, it is possible create tasks that are not performed by anybody because a *TaskUse* can be associated to 0..* *ProcessPerformers*. This type of error could be introduced by an oversight that may hinder enactment since every task must be performed by at least one agent (human or automated agent).

To solve the problem above and others similar to it, we have started our work by redefining some relationships in the SPEM 2.0 metamodel. The modified relationships define the rules shown in Table 1. In this Table, each rule contains a numeration to ease its identification.

Table 1: Relationships modified in SPEM 2.0.

A *TaskUse* must be associated to at least one *ProcessPerformer*. ***(Rule #1)***
A *ProcessParameter* must be associated to exactly one *WorkProductUse*. ***(Rule #2)***
A *RoleUse* must be associated to at least one *ProcessPerformer*. ***(Rule #3)***
A *WorkProductUse* must be associated to at least one *ProcessResponsabilityAssignment*. ***(Rule #4)***
A *TaskUse* must have at least one *ProcessParameter*. ***(Rule #5***)

The classes and relationships that represent the rules above are depicted in Figure 2 and Figure 3. Basically, the rules presented define: **1)** Work products need to have roles assigned to it in a software process. (*Rule #4*); **2)** Tasks must have input and/or outputs in terms of work products and must be performed by roles. (*Rules #1, #2 and #5*); and **3)** Roles need perform tasks. (*Rule #3*).

Since not all well-formedness rules could be expressed through UML diagrammatic notation we introduced first-order predicate logic (FOLP). To write the rules, we first translate the classes, relationships and attributes of SPEM 2.0 metamodel into predicates and logical axioms. Due to space constraints, the translation is not detailed here. We assume that each class and attribute of the metamodel represents a predicate. For example, the *ProcessPerformer* class and its attributes *linkedRoleUse* and *linkedTaskUse* are expressed using the following predicates:

processPerformer(x) where *x* is a instance of a *ProcessPerformer*. **(P1)**

linkedRoleUse(x, y) where *x* is a instance of a *ProcessPerformer* and *y* is a instance *of RoleUse*. **(P2)**

linkedTaskUse(x, y) where *x* is a instance of a *ProcessPerformer* and *y* is a instance *of TaskUse*. **(P3)**

The composition relationship which is a special type of UML association used to model a "whole to its parts" relationship is represented in FOLP with the predicate *part-of(x,y)*. In this predicate, x is an instance of *part* and y represents its *whole*. Considering the properties defined in UML for this type of association the following logic axioms are defined:

$$\forall x\ \neg part\text{-}of(x,x) \quad \textbf{(A1)}$$

$$\forall x,y\ (part\text{-}of(x,y) \rightarrow \neg part\text{-}of(y,x)) \quad \textbf{(A2)}$$

$$\forall x,y,z\ (part\text{-}of(x,y) \wedge part\text{-}of(y,z) \rightarrow part\text{-}of(x,z)) \quad \textbf{(A3)}$$

$$\forall x,y,z\ (part\text{-}of(x,y) \rightarrow \neg part\text{-}of(x,z)) \quad \textbf{(A4)}$$

Some additional predicates that express usual relations in a software process were also created. Such predicates are needed as they are reused for many different well-formedness rules. For example, the following predicates represent, respectively, a work product that is produced by a task and the dependency relationship between two work products. Dependency relationships are used to express that one work product depends on another work product to be produced in a software process.

$$\forall x,y,z((taskUse(x) \wedge workProductUse(z) \wedge processParameter(y) \wedge direction(y, \text{'out'}) \wedge parameterType(y,z) \wedge part\text{-}of(y,x)) \rightarrow taskProduce(x, z)) \quad \textbf{(P4)}$$

$$\forall z,x,y((workProductUse(x) \wedge workProductUse(y) \wedge (workProductUseRelationship(z) \wedge kind(z, \text{'dependency'}) \wedge source(z, x) \wedge target(z, y))) \rightarrow dependency(x, y))) \quad \textbf{(P5)}$$

Similar predicates also exist for the modification and consumption relations of the work products by the tasks in a software process. Such relations are obtained just replacing the value of the constant *'out'* of the *direction* predicate by *'in'* or *'inout'*. When the *'in'* value is used we have the predicate *taskConsume(x, z)* **(P6)** and when the *'inout'* value is used we have the predicate *taskModify(x, z)* **(P7)**. Activities have the same relations of input and output (production, consumption and modification) with work products, so we have considered similar predicates to these elements (**P8**, **P9** and **P10**).

Work products also may assume other types of relationships, in addition to the dependency relationship. In the SPEM 2.0 metamodel these types of relationships are '*composition*' and '*aggregation*'. Both relationships express that a work product instance is part of another work product instance. However, in the composition relationship the parts lifecycle (child work products) are dependent on the parent lifecycle (parent work product). The composition and aggregation predicates just replace the value of the constant '*dependency*' of the *kind* predicate by '*composition*' or '*aggregation*' (**P11**, **P12** and **P13**).

The *composition*, *aggregation* and *dependency* relationships between work products are transitive relations. The logical axioms bellow formalizing this property:

$$\forall x,y,z(composition(x,y) \wedge composition(y,z) \rightarrow composition(x,z)) \quad \textbf{(A5)}$$

$$\forall x,y,z(aggretation(x,y) \wedge aggretation(y,z) \rightarrow aggretation(x,z)) \quad \textbf{(A6)}$$

$$\forall x,y,z(dependency(x,y) \wedge dependency(y,z) \rightarrow dependency(x,z)) \quad \textbf{(A7)}$$

Considering the predicate and logical axioms above the first consistency well-formedness rules to *WorkProductUse* were expressed in FOLP. They are presented in the Table 2 and define: **1)** A work product may not be the *whole* in a relationship (composition, aggregation or dependency) if one of its *parts* represent its *whole* in another relationship or represent its *whole* by the relation transitivity. (*Rule #6, #7 and #8*); **2)** A work product may not represent the *whole* and the *part* in the same relationship (composition, aggregation or dependency). (*Rules #9, #10 and #11*); and **3)** A work product that represents the *part* in a composition relationship may not represent *part* in another relationship of this type. (*Rule #12*)

Note that the well-formedness rules above define the same properties that logical axioms of the *part-of* predicate. However, the well-formedness rules are necessary once the relationships between the work products are not expressed using the UML association represented by the *part-of* predicate. These relationships are expressed using UML classes and attributes and consequently, need to be represented by other predicates and constrained by new rules.

Table 2: First Well-Formedness Rules to WorkProducts.

$\forall x,y\ (composition(x,y) \rightarrow \neg\, composition(y,x))$	***(Rule # 6)***
$\forall x,y\ (aggretation(x,y) \rightarrow \neg\, aggretation(y,x))$	***(Rule # 7)***
$\forall x,y\ (dependency(x,y) \rightarrow \neg\, dependency(y,x))$	***(Rule # 8)***
$\forall x\ \neg\, composition(x,x)$	***(Rule # 9)***
$\forall x\ \neg\, aggretation(x,x)$	***(Rule # 10)***
$\forall x\ \neg\, dependency(x,x)$	***(Rule # 11)***
$\forall x,y,z\ (composition(x,y) \rightarrow \neg\, composition(x,z))$	***(Rule # 12)***

A second important group of consistency well-formedness rules to the *WorkProductUse* written in FOLP are shown in Table 3.

Table 3: Second Group of Well-Formedness Rules to WorkProducts.

$\forall x\ (workProductUse(x) \rightarrow \exists y\ (processParameter(y) \wedge direction(y, \text{'out'}) \wedge parameterType(y, x)))$	***(Rule #13)***
$\forall x,y(taskProduce(x,y) \rightarrow \exists r,w,z(roleUse(r) \wedge (processPerfomer(z) \wedge linkedTaskUse(z, x) \wedge linkedRoleUse(z,r)) \wedge (processResponsabilityAssignment(w) \wedge linkedRoleUse(w,r)) \wedge linkedWorkProductUse(w,y))))$	***(Rule #14)***
$\forall x,y,t\ (workProductUse(x) \wedge dependency(x,y) \wedge taskProduce(t,x) \rightarrow taskConsume(t,y))$	***(Rule #15)***

The well-formedness rules above establish: **1)** Work products must be produced by at least one task in a software process. (*Rule #13*); **2)** At least one responsible role by the work product must be associated in its production tasks. (*Rule #14*); and **3)** If a work product has dependencies in terms of other work products these dependencies must be input in its production tasks. (*Rule #15*)

The last group of well-formedness rules are related to *TaskUses* sequencing. To establish the tasks sequence from SPEM 2.0 metamodel the *WorkSequence* class and its *linkKind* attribute are used. It is possible using the following values in sequencing between *TaskUses*: *finishToStart*, *finishToFinish*, *startToStart* and *startToFinish*.

Some predicates and logical axioms related to precedence between the tasks were created. Initially, to capture the concept of successor and predecessor task we have defined the predicates *pre-task*(t_1, t_2)

and *pos-task*(t_2, t_1), where t_1 and t_2 are *TaskUse* instances and indicate, respectively, t_1 as predecessor task of t_2, or, inversely, t_2 as successor task of t_1. The predicates *pre* and *pos-task* are transitive and asymmetric relations. The following logical axioms establish these properties to these relations:

$$\forall(t_1, t_2)\ (pre\text{-}task(t_1, t_2) \leftrightarrow pos\text{-}task(t_2, t_1)) \qquad \textbf{(A8)}$$

$$\forall(t_1, t_2, t_3)\ (pre\text{-}task(t_1, t_2) \wedge pre\text{-}task(t_2, t_3) \rightarrow pre\text{-}task(t_1, t_3)) \qquad \textbf{(A9)}$$

$$\forall(t_1, t_2)\ (pre\text{-}task(t_1, t_2) \rightarrow \neg pre\text{-}task(t_2, t_1)) \qquad \textbf{(A10)}$$

$$\forall t_1\ \neg pre\text{-}task(t_1, t_1) \qquad \textbf{(A11)}$$

Based on the predicates and logical axioms related to precedence between tasks we have defined new consistency well-formedness rules. These rules, shown in Table 4, define: 1) The tasks sequencing must not have duplicated sequences. (*Rule #16*) **2)** Work Products must be produced before they are consumed. (*Rule #17*) and **3)** The dependencies of a work product must be produced before it in a software process. (*Rule #18*)

The well-formedness rule *#16* shown in the Table 4 is only to *startToFinish* transition. Consider the same rule to the following transitions: *startToStart*, *finishToFinish* and *startToFinish*.

Table 4: Well-Formedness Rules to Process Sequence.

$\forall x,x1,x2((taskUse(x1) \wedge taskUse(x2) \wedge (workSequence(x) \wedge predecessor(x, x1) \wedge sucessor(x, x2) \wedge linkKind(x, \text{'startToFinish'}))) \rightarrow \neg\exists y(workSequence(y) \wedge predecessor(x,x1) \wedge sucessor(x,x2) \wedge linkKind(x, \text{'startToFinish'})))$	***(Rule #16)***
$\forall x, y\ (taskConsume(x, y) \rightarrow \exists x2\ (taskProduce(x2, y) \wedge pre\text{-}task(x2, x)))$	***(Rule #17)***
$\forall x,y\ (dependency(x,y) \rightarrow \exists t1, t2\ (taskProduce(t1, x) \wedge taskProduce(t2, y) \wedge pre\text{-}task(t2,t1)))$	***(Rule #18)***

6 EVALUATION OF THE WELL-FORMEDNESS RULES

This section presents a process checking example using a part of the OpenUP process. The section also evaluates one of the well-formedness rules proposed in this paper. The main goal is demonstrate that the predicates and logical axioms used in the well-formedness rules really express the intended meaning.

6.1 Process Checking Example

To present a process checking example we have considered the *Inception Iteration* of the OpenUP process, which is shown in Figure 4. In this Figure, above the dash line, the activities and tasks of the iteration are represented. Additionally, some information about activities sequence is also shown. Below the dash line, the tasks of the *Initiate Project* activity are detailed in terms of roles and work products (inputs and outputs). All information shown in the Figure 4 is based on the OpenUP process except the *Rule Test* which was introduced by us only for this evaluation. Originally, in OpenUP, the *Analyst* is also responsible for the *Vision* work product.

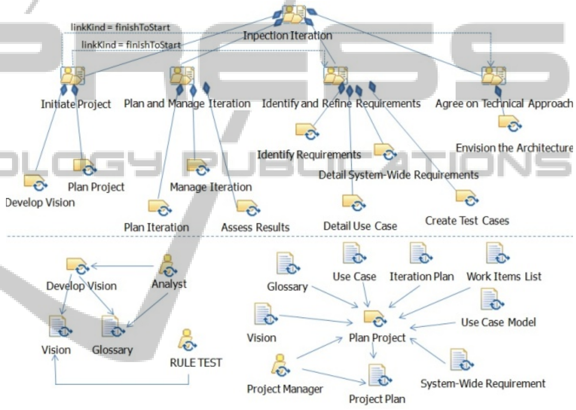

Figure 4: Inception Iteration of the OpenUp.

One of the tasks of Figure 4 (*Develop Vision*) is also represented with a UML object diagram, which is shown in Figure 5. The object diagram show the class instances of the SPEM 2.0 used to create tasks, work products, roles and their relationships in a software process. In Figure 5 letters are used to facilitate its understanding. The *letter A* indicates the *WorkProductUse* classes used to create the objects *Vision* and *Glossary*. The *letter B* represents the objects *01* and *02*, which are instances of the ProcessParameter class. These kinds of objects represent the inputs and outputs to the task objects. In Figure 5, the object that represents a task is represented by the *DV* (*Develop Vision*) identifier. This object is an instance of *TaskUse* class and is indicated in the Figure 5 by the *letter C*. The objects representing instances of the *RoleUse* class are indicated in Figure 5 by the *letter D*. Finally, the *letters E* and *F* represent, respectively, objects of the *ProcessResponsabilityAssignment* (object 01 and 02) and *ProcessPerformer* classes (object 02). The instances of the *ProcessResponsabilityAssignment* are used to define roles as responsible for work products and the instances of the ProcessPerformer are used to link roles as performer to the tasks.

As seen, all process information of this example may be represented using classes and relationships of the SPEM 2.0. It means that the used process is compliance with the SPEM 2.0 metamodel. Another fact that shows the consistency of the used process is the validation result of the object diagram found in the *case* tools like *Rational Software Modeler*. This validation result is error free.

However, as mentioned in Section 4, not all need information in a software process can be expressed using only the UML language. Thus, when we carry out the checking in the same process using our well-formedness rules it presented errors indicating some inconsistencies. The first inconsistency of the software process used in this example is in the task *Develop Vision*. As seen in Figure 4, the task *Develop Vision* produces the work product *Vision* which has as responsible role the role *Rule Test*. This role does not perform the task *Develop Vision* and this fact violates the *Rule #14* which defines that at least on responsible role of a work product must participate of their production tasks. Another problem can be seen in the task *Plan Project*. Note that this task has as mandatory inputs the work products *Use Case, Use Case Model* and *System-Wide Requirements* which are not yet produced in the software process when this task is performed. This inconsistency violates the *Rule #17*.

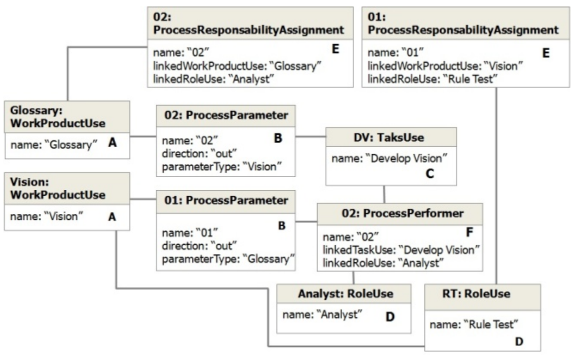

Figure 5: Object Diagram to the *Develop Vision* Task.

6.2 Evaluation of the Well-formedness Rules

We have evaluated our well-formedness rules expressed in FOLP to check their correctness. Since the amount of rules presented in this paper is vast and due the space constraints, we present only the evaluation of rule *Rule #14*.

To start the evaluation we have created some variables and assigned values for them. Each variable represents an object of the object diagrams shown in Figure 5. Table 5 lists the variables and values used to this evaluation.

Table 5: Variables used in the First Evaluation.

x::= 'DV'	x is the *TaskUse* 'Develop Vision'
y::= 'Vision'	y is the *WorkProductUse* 'Vision'
r::= 'Analyst'	r is the *RoleUse* 'Analyst'
t::= '02'	t is the *Process Parameter* '02' with *direction* equal to '*out*' and *parameterType* equal to 'Vision'
z::= '02'	z is the *ProcessPerformer* '02' with *linkedRoleUse* equal to 'Analyst' and *linkedTaskUse* equal to 'Develop Vision'
w::= '01'	w is the *ProcessResponsability Assignment* '01' with *linkedRoleUse* equal to 'Rule Test' and *linkedWorkProductUse* equal to 'Vision'

We have evaluated the task *Develop Vision* which presents an error in the software process. The formalization of the *Rule #14* is the following:

$$\forall x,y(taskProduce(x,\ y)\ \rightarrow\ \exists r,\ w,\ z\ (roleUse(r) \wedge (processPerfomer(z) \wedge linkedTaskUse(z,x) \wedge linkedRoleUse(z,r)) \wedge (processResponsabilityAssignment(w) \wedge linkedRoleUse(w,r) \wedge linkedWorkProductUse(w,y))))$$

This rule uses the *taskProduce(x, y)* that is represented by the following sentence in FOLP:

$$\forall x,y,t((taskUse(x) \wedge workProductUse(y) \wedge (processParameter(t) \wedge direction(t,\ 'out') \wedge parameterType(t,\ y)) \wedge part\text{-}of(t,\ x)) \rightarrow taskProduce(x,\ y))$$

Initially we have evaluated the *taskProduce(x,y)*. Considering the variables of Table 5 we have:

taskUse(DV)::= T
workProductUse(Vision)::= T
ProcessParameter(02)::= T
direction(02, 'out')::= T
parameterType(02, Vision)::= T
part-of(02, DV)::= T
taskProduce(Criar DV, Vision)::= T

Then:

$$\forall x,y,t\ ((T\ \wedge\ T \wedge (T \wedge\ T \wedge\ T) \wedge\ T) \rightarrow T)$$

$$\forall x,y,t\ (T \rightarrow T)::= T$$

Predicate *taskProduce(DV, Vision)* evaluates to *True*. Once the task *Develop Vision* produces the work product *Vision* the expected value was *True*. Considering *Rule #14* we have:

roleUse(Analyst)::= T
processPerformer(02)::=T
linkedRoleUse(02, Analyst)::= T
linkedTaskUse(02, DV)::= T
processResponsabilityAssignment(01)::=T
linkedWorkProductUse(01, Vision)::= T
linkedRoleUse(01, Analyst)::= F

Then:

$\forall x, y\ (T \rightarrow \exists r, w, z\ (T \wedge (T \wedge T \wedge T) \wedge (T \wedge F \wedge T)))$

$\forall x, y\ (T \rightarrow \exists r, w, z\ (F))$

$\forall x, y\ (T \rightarrow F)::= F$

The value to the *Rule #14* is *False*. This value was expected once the values assigned to the variables generate one inconsistency in the software process as already shown in the Subsection 6.1. It suggests that the theory of the *Rule #14* is valid.

Although we have not detailed the evaluation of the *Rule #17*, the value returned to this evaluation is *False*. It also indicates that the theory of this rule is valid.

7 CONCLUSIONS

In this paper, we have proposed well-formedness rules that allow finding errors in a software process before it is enacted. By noting inconsistencies in the process, we believe it is possible for modellers to refine a process model until it is free of inconsistencies.

The proposed well-formedness rules were based on SPEM 2.0 metamodel. To define them we have modified multiplicity constraints and for the more elaborated rules which could not be expressed only with UML, we have used FOLP.

Several research directions, which we are working on, have been left open during this paper, and here we emphasize two of them. First, more well-formedness rules considering others process elements and consistency aspects need to be provided. Related to this, preliminary studies suggest two important facts: (1) other process elements and relationships must be included in the SPEM 2.0 metamodel and (2) the OCL language does not support the definition of all well-formedness rules needed to guarantee consistency. For example, the well-formedness rules to check cycles in a software process, which involve temporary aspects, may not be expressed using OCL. This fact has been the motivation to use FOLP in this paper. Secondly, with regard to automatic support, the prototype of a tool prototype is being developed. This will support the definition and tailoring of SPEM-based software processes. Furthermore, a process checking, which implements the well-formedness rules, will be provided.

ACKNOWLEDGEMENTS

Study financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91.

REFERENCES

Atkinson, D. C., Weeks, D. C. and Noll, J. *Tool Support for Iterative Software Process Modeling*. Information and Software Technology, 493-514, 2007.

Bajec, M., Vavpotic, D. and Krisper, M. *Practice-Driven Approach for Creating Project-Specific Software Development Methods*. Information and Software Technology, 345-365, 2007.

Bendraou, R., Combemale, B., Cregut, X. and Gervais, M. P.*Definition of an Executable SPEM 2.0*.In: 14th Asia-Pacific Software Engineering Conference, 2007.

Gnatz, M., Marschall, F., Popp G., Rausch A. and Schwerin W. *The Living Software Development Process*. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3371, 2003.

Habli, I. and Kelly, T. *A Model-Driven Approach to Assuring Process Reliability*. In: 19th International Symposium on Software Reliability Engineering, 2008.

Henderson-Sellers, B. and Gonzalez-Perez, C. *A Work Product Pool Approach to Methodology Specification and Enactment*. Journal of Systems and Software, 2007.

Henderson-Sellers, B., Gonzalez-Perez, C. and Ralyté, J. *Comparison of Method Chunks and Method Fragments for Situational Method Engineering*.In: 19th Australian Conference on Software Engineering, 2008.

Hsueh, N. L., Shen, W. H., Yang, Z. W and Yang, D. L. *Applying UML and Software Simulation for Process Definition, Verification and Validation*. Information and Software Technology, 897-911, 2008.

Hug, C., Front, A., Rieu, D. and Henderson-Sellers, B. *A Method to Build Information Systems Engineering Process Metamodels*. The Journal of Systems and Software, 1730-1742, 2009.

Jacobson, I., Booch G., Rumbaugh J. *The Unified Software Development Process*, Addison Wesley, 2001.

Kruchten, P. *The Rational Unified Process: An Introduction*. NJ: Addison Wesley, 2000.

Lucas, F. J., Molina, F. and Toval, A. *A Systematic Review of UML Model Consistency Management*. Information and Software Technology, 1631-1645, 2009.

OMG, *Sofware Process Engineering Metamodel - SPEM 1.1*. Available from: http://www.omg.org/, 2002.
OMG, *Sofware Process Engineering Metamodel - SPEM 2.0*. Available from: http://www.omg.org/, 2007.
Open. Available from: http://www.open.org.au, 2006.
Puviani, M., Serugendo, G. D. M., Frei, R. and Cabri G. *Methodologies for Self-organising Systems: a SPEM Approach*. In: International Conference on Web Intelligence and Intelligent Agent Technology, 2009.
Ralyté, J., Backlund, P., Kuhn, H. and Jeusfeld M. A. *Method Chunks for Interoperability*. In: 25th Int. Conference on Conceptual Modelling, 2006.
Serour, M. K. and Henderson-Sellers, B. *Introducing Agility – A Case Study of SME Using the OPEN*. In: 28th Computer Sof. and Applications Conf., 2004.
Wistrand, K. and Karlsson, F. *Method Components Rationale Revealed*. In: Lecture Notes in Computer Science, Vol. 3084/2004, 2004.
Xu, P., Ramesh, B. *A Tool for the Capture and Use of Process Knowledge in Process Tailoring*, In: Proc. of Hawaii Int. Conference on System Sciences, 2003.