Support for Eviation Detections in the Context of Multi-Viewpoint-Based Development Processes

Reda Bendraou[1], Marcos Aurélio Almeida da Silva[1,2], Marie-Pierre Gervais[1,2] and Xavier Blanc[3]

[1] LIP6, UPMC Paris Universitas, France
[2] UPO, Université Paris Ouest, France
{reda.bendraou, marcos.almeida, marie-pierre.gervais}@lip6.fr
[3] Labri, Université de Bordeaux 1{xavier.blanc@labri.fr}

Abstract. One recurrent issue in software development processes are developer's deviations from the process model. This problem is amplified in the context of multi-viewpoint-based development of complex systems where the system's specification comes in form of different and intertwined viewpoints. Without a methodological support, these deviations become inevitable. They can be of different kinds: 1) behavioral deviations related to inappropriate actions performed by the developer when realizing process's activities or 2) structural deviations due to inconsistencies in deliverables, which can be in conflict with other viewpoint's outcomes. This paper proposes an approach to overcome these issues. To demonstrate the approach, a prototype was developed and the RM-ODP standard and a viewpoint-based development process were used.

1 Introduction

Recently, multi-viewpoint modeling appeared to be a promising approach for dealing with system's complexity. The system is described through the composition of different viewpoints, each one focusing on a precise concern such as security, persistency, GUI, and so on. The main idea is to focus the developer's attention on a specific aspect of the system thus, abstracting away all the irrelevant details. Different modeling languages can be used for the specification of the system's viewpoints and they can be at different levels of abstraction. This inevitably raises problems related to the heterogeneity of the viewpoints, the overlapping of the concepts between them, and the fact that consistency should be constantly maintained between them. Large-scale systems span multiple and intertwined viewpoints; involve long and multidisciplinary design activities which are not supposed to be known by all project's developers.

In such context, it becomes essential to provide a methodological support during the development process. Indeed, when a developer is performing modeling actions inside a process's activity on a given viewpoint, he usually does not have a global view of the whole system design and is far away from assessing instantly the effects of his actions on the other system's viewpoints

Many approaches provide methodological support with the help of process execution and monitoring engines, also called PSEE (Process-centered Software Engineer-

ing Environment) [2][3]. They mainly ensure that the process is correctly applied by the developers by verifying that the activities are executed in the appropriate order and timing, and that the roles are correctly assigned to the right developers. However, if the developer is performing inappropriate actions inside a given activity, this will not be reported by the PSEE. Moreover, if these actions are in contradiction with the process guidelines, they will straightforwardly impact the other viewpoints. As a consequence, the effects are discovered very late, and hence will require costly maintenance actions. Detecting such deviations as soon as they occur can improve the process organization and prevent from risks of project failure and unnecessary delays.

In this work we propose an approach for providing a process support in the context of multi-viewpoint development processes. This support is manifold. First it ensures that developers perform the process in the specified order. Second, it makes sure that in each viewpoint, the developers are following the process guidelines and that they are not violating the project's methodological constraints. It also enforces that developer's actions in a viewpoint do not impact negatively the other system's viewpoints. Finally, it is able to detect developer's deviations from the process model as soon as they occur and to warn the developer of the deviation's cause.

To illustrate our approach, we use RM-ODP (Reference Model of Open Distributed Processing) [5,7], a multi-view-based standard for the specification of distributed systems. The next section categorizes the kinds of process deviations that may occur in such a context and their effects on the process execution. Section 3 presents in details our approach. It is based on Praxis Rules, our language for expressing process and methodological constraints in the context of multi-viewpoint design. The approach is then evaluated in Section 4 through a case study using RM-ODP. Section 5 concludes this paper and sketches some perspectives of this contribution.

2 Categorization of process deviations

Developer's deviations that may occur during a development process can be categorized into four kinds. The first one is what we call **organizational deviations**. They occur when an activity's deadline is not respected, when a role is not fulfilled or assigned to inappropriate developer. The second kind is called **micro behavioral deviations.** These deviations occur when a developer is performing inappropriate actions inside a modeling activity and thus, violating methodological guidelines or business constraints (e.g. a developer is applying a design pattern in a wrong way). This kind of deviation may be the consequences of developer's misunderstanding of the work to accomplish or his willingness to perform the activity by following his intuitions and experience. In the context of multi-viewpoint modeling, if developers are performing modeling actions that are in complete contradiction with what was specified in other related viewpoints, they won't be notified with the eventual conflicts until the end of the activity i.e., until they submit their deliverables to the PSEE for a structural check. We believe that the early detection of micro behavioral deviations can avoid rework actions which represents a considerable gain in terms of time and efforts. Our proposition aims at detecting them as soon as they occur in order to prevent project managers from process failures.

Structural deviations are triggered when a model delivered by an activity has some inconsistencies. In the context of multi-viewpoint modeling the fact that different modeling languages can be used in each viewpoint adds more complexity in maintaining the structural consistency between the different deliverables. The challenge is then to provide an independent-modeling language approach to ensure such consistency. In section 3 we present our proposition to this problem.

Finally, **macro behavioral deviations** occur when a developer decides to execute process's activities in a different order than the one prescribed by the process model. This can be due to an expected project's constraints. In all cases, it is primordial that deviations, whatever their kind, have to be captured by the PSEE and reported instantly to the project manager in order to assist him in taking the appropriate decisions.

3 Praxis and Praxis Rules

Praxis and Praxis Rules are the building blocks of our approach. Due to the lack of space, in the following we focus on demonstrating their use for detecting only structural and micro behavioral deviations. The same principle applies for the other kinds of deviations.

3.1 Praxis

Praxis is used to represent each elementary modeling action performed by a developer within a modeling tool [1]. This representation has already been used in the context of artifacts described in different languages like EMF, UML, XML and Java[1]. It consists of six classes of atomic actions inspired from the MOF reflexive API [4]. The *create(me, mc, t)* and *delete(me; t)* actions respectively create and delete a model element *me*, that is an instance of the meta-class *mc* at the timestamp *t*. The *addProperty(me, p, value, t)* and *remProperty(me, p, value, t)* add or remove the value *value* to or from the property *p* of the model element *me* at timestamp *t*. Similarly, the actions *addReference(me, r, target, t)* and *remReference(me, r, target, t)* add or remove the model element *target* to or from the reference *r* of model element *me* at timestamp *t*.

In the present work, we extend every Praxis action with an extra parameter *v* to represent the *viewpoint* in which it has been performed. For example, the action *create(me, mc, t, v)* represents the creation of the model element *me*, an instance of the metaclass *mc*, in the timestamp *t* in the viewpoint *v*. The same applies for the other kinds of actions. With this extension, we can then represent the different actions performed by developers over different kinds of models. Figures 2 and 3 exemplify two different viewpoints in a given system. The former represents its structural viewpoint with the package *Azureus* and the *Client* and *Server* classes. The second figure represents the behavioral viewpoint by means of a sequence diagram.

Now suppose that a modeler renames server role in the behavioral viewpoint from *Role2* to *MainServer* and that, later, another modeler deletes an operation from the structural viewpoint. These low level actions can be represented by the sequence of Praxis actions below. Notice that Praxis is able to represent changes in different viewpoints in a way that is independent of the meta-models used to represent them.

[1] http://code.google.com/p/harmony.

```
remProperty(role2, name, 'Role2', 12, 'behavioral').
addProperty(role2, name, 'MainServer', 13, 'behavioral').
delete(op1, operation, 14, 'structural').
```

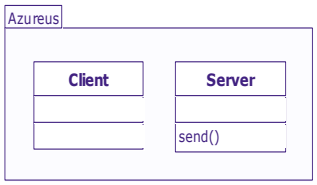

Fig. 2. Structural viewpoint: a UML package with its content

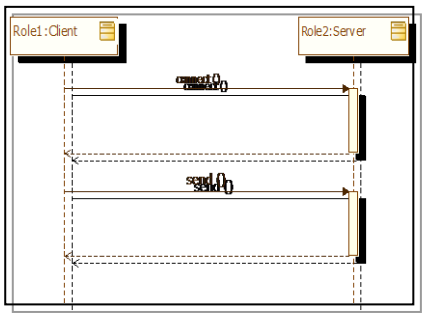

Fig. 3. Behavioral viewpoint : Sequence Diagram

3.2 Praxis Rules

PraxisRules is the rule based language that is used by the PSEE to detect both structural and behavioral deviations during process execution. This language has already been used to detect structural deviations in industrial multi-view models [6] and in structural and behavioral deviations in single viewpoint PSEEs [9]. The PSEE detects deviations by comparing each rule with a Praxis trace captured from the process execution. There are two kinds of rule in PraxisRules, 1) activity post-condition rules that define structural constraints over a sequence of praxis actions and, 2) activity invariant rules that define behavioral constraints over a sequence of praxis actions.

Activity post-condition rules have the form "*ruleName(Variables) <=>expression."* where *ruleName* is the name of the rule, *Variables* is a list of variables in the rule and *expression* is a logical expression. The meaning of such rule is that *ruleName(Variables)* is *true* in the sequence if and only if *expression* holds. In the expression, a combination of logic predicates such as *and{…}* for conjunctions, *or{…}* for disjunctions and *not{…}* for negations and references to Praxis actions (such as *create(ME,MC,T,VP))* is allowed.

Activity invariant rules have the form "*ruleName(Variables) @ action <=> expression."*, where *ruleName* represents the name of the rule and *Variables* list of variables in the rule. *@action* is an action variable that refers to a particular action in the sequence. The expression *expression* is then used to validate the presence of *@action* in the sequence or to define the allowed order of other actions taking it as reference. The order of actions is defined by temporal operators like *@before{…}* and *@after{…}*. For example, the expression *@before{ action1 @ addReference(A, B, C), action2 @ remReference(A, B, C) }* means that the action matching *addReference(A, B, C)*, represented by the action variable *@action1*, should appear before the action matching *remReference(A, B, C),*represented by the action variable *@action2*, in the Praxis sequence. Notice that variables are represented by words starting in uppercase letters and in lower case letters for action variables; that timestamps may be omitted from Praxis actions; and that the syntax *[@action] call ruleName(Parameters)* is used

to call the rule *ruleName* with the parameters *Parameters* and with the action *@action* for activity invariant rules.

In order to check a constraint that may span multiple models i.e. inter-model constraints, in this paper we allow the viewpoint information in the Praxis representation of actions to be used in PraxisRules. Indeed, contrarily to languages like OCL [8], PraxisRules does **not** impose unique context constraints and can be used to express constraints over a sequence of editing actions among a set of viewpoints. Thus, every time a developer performs a modeling action in a specific viewpoint, the latter is captured by Praxis and annotated with the timestamp and the viewpoint information. Inter-model constraints represented in the form of a PraxisRule are then checked over the combination of the entire viewpoints' sequences of actions. If the rule does not hold, a deviation is triggered.

For instance, let us consider the structural viewpoint given in Figure 2 and the behavioral viewpoint given in Figure 3. In this example, one inter-model behavioral constraint could be that during an activity called *renameMessages*, the developer would be asked to rename the messages in the behavioral viewpoint, but that the names he provides need to correspond to names of operations in the structural viewpoint. Using Praxis Rule this is how such a constraint is expressed:

```
renameMessagesInv(M) @ action <=> or {
   action @  remProperty(E, name, Name, 'behavioral'),
   and {
      action @  addProperty(E, name, Name, 'behavioral'),
       call existsElementInViewpointByName(Name, operation, 'structural')
   }}
existsElementInViewpointByName(Name, MC, V) {
    create(E, MC, V),
    addProperty(E,name, Name)
}
```

This rule called *renameMessagesInv* states that an action *@action* should be either a *remProperty(E, name, Name, 'behavioral')*, meaning a removal of a name of some element in the behavioral viewpoint, or an *addProperty(E, name, Name, 'behavioral')*, meaning the addition of a name for some element in the behavioral viewpoint. The *addProperty* action is further constrained by the existence of an operation *OP* in the structural viewpoint having the same *Name* as the element *E* renamed by the developer. This extra constraint is enforced by the rule called *existsElementInViewpointByName*. This rule is verified by the PSEE by replacing the action variable *@action* with every action executed during the activity. A micro behavioral deviation is then raised if the developer executes any action that does not conform to the constraint. That is the case when he renames a message with the name of inexistent operation or when other actions like creating new elements are performed.

4 Evaluation

In order to evaluate the feasibility of our approach we developed a prototype and tested it in the context of a development process. We used RM-ODP, a multi-viewpoint-based standard for the specification of distributed systems [7]. As a process example, we borrowed the one presented in the UML4ODP profile specification [5]. It describes the design process of the "Templeman Library system" using RM-ODP.

In the case study presented in Section 4.2, Praxis Rules are used 1) to specify RM-ODP consistency rules and to illustrate the occurrence of a structural deviation (inter-viewpoints) and its detection in case of one of these rules is violated; 2) to define the set of allowed actions during the modeling of a given viewpoint of the Library system. The process part consisting in defining the Enterprise viewpoint was taken as an example to illustrate the occurrence of micro behavioral deviations and how our prototype detects them. The RM-ODP defines 5 viewpoints, namely the Enterprise, the Information, the Computational, the Engineering and the Technology viewpoints. For the interested reader, more details can be found on RM-ODP in [7]. In the following, we present our prototype.

4.1 Prototype

In our prototype, we adopted MagicDraw[2] as a modeling tool. Our choice was influenced by the fact that MagicDraw provides a UML Profile for RM-ODP called UML4ODP. Each viewpoint is specified as a stereotyped package (e.g., <<Enterprise_spec>>). Of course, any modeling tool can be used in place of MagicDraw.

The following picture displays a screenshot of our prototype (c.f., Figure 4). It shows a scenario of a process enactment. The parts (1), (2) and (3) represent an extension to MagicDraw in order to display the RM-ODP viewpoint that the developer is working on. Part (4) is also an extension that shows the activity being enacted by the developer. In this sample scenario, the developer executed an action that was not allowed by the process model. That is why a dialog box (5) is prompted to indicate that a deviation occurred due to the execution of an action that violates the process modeling rules. The same kind of dialog box is used to display guidelines during process enactment.

4.2 Case Study

The case study consisted in running a multi-viewpoint-based process on top of our prototype. During the process enactment, we deliberately caused different kinds of deviations i.e. structural and micro behavioral and we controlled if the tool succeeded in detecting all of them instantly. In the following, we present the process modeling rules represented using PraxisRules.

The process model and modeling rules.

As an example of a development process using RM-ODP, we used the one initially described in natural language in the UML4ODP specification, page 68 of [5]. For the sake of clarity, we focus on the part of the process activities required for the specification of the Enterprise viewpoint of the library application. A first step was to map each activity of the process to Eclipse cheat sheet steps. The second step consisted in representing the consistency rules between the different viewpoints in form of Praxis Rules (i.e., Activity post-condition or invariant rules).

Let us take the first activity of the process as an example i.e., "Identify the communities, with which the system is involved, and their objectives". For this activity, a process modeling rule was defined using Praxis Rule (see Figure 5). This rule comes

[2] Site, http://www.magic-draw.com

in the form of an activity invariant rule that states that in the Enterprise viewpoint, one can only create elements that relate to that viewpoint. Additionally, it states that only three kinds of elements can be created: *communities*, *objectives* and *"objective of" associations*, which link communities to objectives. These elements are represented in UML respectively by components tagged with the *EV_Community* stereotype; classes tagged with the *EV_Objective* stereotype and associations tagged with the *EV_ObjectiveOf* stereotype. During the process execution, if the developer performs an action not allowed by this rule, a micro behavioral deviation is triggered instantly and its cause is displayed to the developer.

For brevity reasons, since we need one activity post-condition and an activity invariant rule per activity, which would account for 16 rules for the selected process, we are not able to present all the praxis rules in this paper. However, the complete source code of our prototype, along with the process model that was used to detect structural and micro/macro behavioral deviations is available at our WebSite[3]

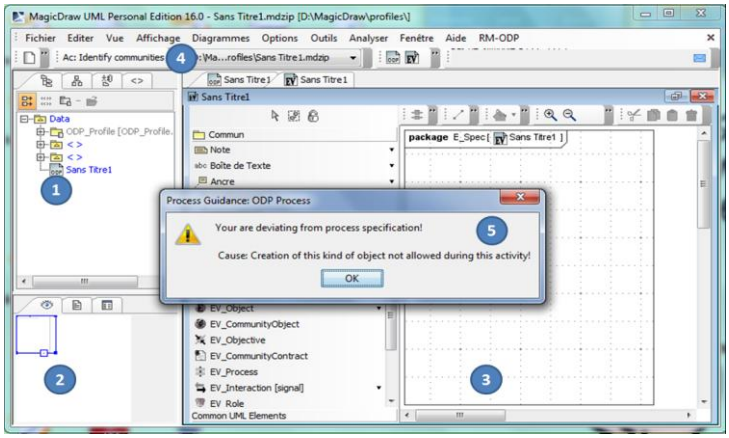

Fig. 4. Screenshot of a process sample executed in our prototype

```
public IdentifyCommunities() @ a <=> and {
   a @ call inViewpoint("enterprise"),
   not {  t @ create(C,MC),
   not {  or {   and { addProperty(C, stereotype, S),
                              call goodCombination(MC,S)  },
                MC = property  }}
  }
}.
goodCombination(MC, S) <=> or { and { MC = "component", S = "ev_community" }, … }
inViewpoint(V) @ action <=> or { action @ create(E,MC,V), … }
```

Fig. 5. Sample of process modeling rule

[3] (*http://lip6.fr/Marcos.Almeida/publications.html*).

4.3 Discussion

The realization of the prototype and the case study was an important step for us and revealed the feasibility of the approach. Most of all, we were able to ensure the detection of both structural and behavioral deviations. These deviations are detected instantly and the developer is informed with the cause of its deviations. Regarding structural deviations, we were able to detect both intra- and inter-viewpoint inconsistencies, which is of prime importance in the context of multi-viewpoint modeling. In our case study, the language used for defining the different viewpoint was the same but thanks to Praxis Rule, having different modeling languages would not change anything to our solution

A step further in the validation process would be to conduct an empirical study to assess the benefits, in terms of time and quality, of offering such support to the developers. In a previous work, we realized such a study but this was done with a process example which did not include the modeling of a system with several viewpoints [10].

We assessed the "effort of adoption" by considering the coding efforts required for extending the MagicDraw case tool to implement our approach. Thanks to MagicDraw Open API, implementing the MAL component has been implemented as a 360 lines of Java (PropertyChangeListener). This component took one day of work for a Java experimented developer. When it comes to the PEE, our approach is mostly independent from it. During this experiment it consisted in a simplistic extension of MagicDraw interface which amounted to less than 200 lines of Java code. The only dependency of the other components to this one is that the DDE needs to know which is the current activity being executed by the developer so that it can verify the chosen behavioral and structural rules. Our approach would then be able to be reused by any existing PEE that is able to provide these pieces of information to the other components of our approach.

5 Conclusion

In the context of multi-viewpoint-based projects, the risks that developers deviate during the development process are amplified. The heterogeneity of the viewpoints, there overlapping and the need to ensure consistency between them inevitably introduce many chances for developer deviations. If not handled on time, these deviations may cause the failure of the project in terms of reliability of the project's outcome, delays and costs. In this paper we proposed an approach that allows capturing developer's deviations during process realization. Whatever their kind i.e., behavioral or structural, these deviations are detected instantly as they occur and their causes are reported to the developer or project manager. They can then take the appropriate decisions and anticipate the risks that may penalize the course of the project.

As a perspective of this work we are currently studying the resolution of the optimal path to reconcile the developer with the process description in case of late deviation detections i.e. the early deviation detection is turned off by the developer. We also plan to put in place a more important empirical study for validating our approach.

References

1. X. Blanc, et al. Detecting model inconsistency through operation-based model construction. In Robby, editor, Proc. Int. Conf. Software engineering (ICSE'08), volume 1, pages 511–520. ACM, 2008.
2. G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. IEEE Trans. Software Eng., 24(11):982–1001, 1998.
3. M. Kabbaj, R. Lbath, and B. Coulette. A deviation management system for handling software process enactment evolution. In Q. Wang, D. Pfahl, and D. M. Raffo, editors, ICSP, volume 5007 of Lecture Notes in Computer Science, pages 186–197. Springer, 2008.
4. OMG: Meta Object Facility (MOF) 2.0 Core Specification (January 2006)
5. Information technology — Open distributed processing — Use of UML for ODP system specifications ITU-T Recommendation X.906, ISO/IEC 19793. At: http://www.lcc.uma.es/~av/download/UML4ODP_IS_V2.pdf
6. J. Le Noir, O. Delande, D. Exertier, M. A. A. da Silva, and X. Blanc. 2011. Operation based model representation: experiences on inconsistency detection. In Proceedings of ECMFA'11, Springer-Verlag, Berlin, Heidelberg, 85-96.
7. ITU-T Recommendation X.902 (1995) | ISO/IEC 10746-2:1996, Information technology – Open Distributed Processing – Reference Model: Foundations.
8. OMG: UML 2.0 OCL Specification, November 2003
9. M. A. A. da Silva, R. Bendraou, X. Blanc, and M.-P. Gervais. Early deviation detection in modeling activities of mde processes. In Petriu et al. [11], pages 303–317.
10. M. A. A. da Silva, A. Mougenot, R. Bendraou, J. Robin, and X. Blanc. Artifact or process guidance, an empirical study. In Petriu et al. [11], pages 318–330.
11. D. C. Petriu, N. Rouquette, and Ø. Haugen, editors. Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II, volume 6395 of LNCS. Springer, 2010.