PraxisRules: An Approach for Detecting and Handling Process Deviations

2.1 Motivation

Software development companies continually aim at improving software quality and development productivity. Software processes help them in fulfilling these objectives by describing well-known best practices in how the internal development activities should be performed [Humphrey 89]. This is accomplished by means of so-called Software Process Models (SPM). These models represent software development processes as the set of partially ordered activities that have to be executed by process agents [Lonchamp 93]. In software development activities, process agents play different roles, such as developer, designer, modeler, tester etc.

Since the execution of a SPM is a human-centered task, it is essentially an error-prone activity. As a consequence, companies look for means to automate the execution of SPMs. With the help of PSEEs (the Process-centered Soft-ware Engineering Environments), software companies can make sure that the actions executed by process agents are consistent with the adopted software process model [Fuggetta 00]. When these actions are not consistent with the SPM, they are called deviations, and they represent a risk to the objectives of the process. Part of the role of a PSEE consists of keeping track of these kinds of risks. In turn, it is expected that the software quality and development speed improvements can be achieved.

Ideally, avoiding deviations would be an easy objective to accomplish: one would need to observe every action performed by the agents during process enactment and then compare them with the process model. Actions that were not explicitly allowed by the SPM should therefore be strictly forbidden. Nonetheless, this scenario is unrealistic and the difficulties in accomplishing it fit into three categories: Observational Incompleteness, Process Incompleteness and Temporal Incompleteness.

PSEEs are not able to observe every action performed by process agents. Even if they were, they would need to compare them with software process models that are sometimes incomplete and that do not necessarily reflect the way the process is executed. Deciding on when deviations should be detected and when agents should be forced to respond to them depends on future actions, which are most of the time unpredictable.

Observational Incompleteness refers to the fact that observations made by a PSEE over the process execution are inherently incomplete. The reasons for that are twofold: (1) process models are usually expressed at a level of abstraction that is often too far from the actions that are observable during process enactment and, (2) even for processes that are completely detailed, their effective execution can still be unobservable to the PSEE. In other words, process models do not contain enough information to be observable; on the other hand, they contain pieces of information that are not accessible to PSEEs.

Regarding the first reason, as discussed by Curtis et al in [Curtis 92], process models are usually defined by means of natural language constructs. These constructs are useful for human understanding but are hard to be directly usable by a PSEE. The solution to this problem is to add details to the process specification, which usually leads to process models that are too complex to understand and to maintain and to PSEEs that are too intrusive. In practice, the negative side effect of this complexity is the reduced adoptability of these PSEEs [Fuggetta 00]. Regarding the second
reason, parts of the process may be unobservable if they are executed in environments that are not under the control of the PSEE. For example, it is not uncommon for a software process model to contain activities that are executed in specialized (and sometimes proprietary) tools. PSEEs need to take into consideration that they always have a partial view of the actions executed by the agents.

Even if we consider that one can have a total view over the actions executed by the agents, forcing them to strictly follow the process model is an approach that is doomed to failure. This would happen because software processes are usually incomplete, and sometimes incorrect. That is what we call Process Incompleteness. This difficulty happens when one supposes that the process model is a perfect description of how the activity should be performed, and that it is expressed in a way that is adapted to every process agent. These hypotheses have been contradicted by past research that showed that process models are inherently incomplete descriptions that need to be updated over time [Huo 06]. Works like the one of Simidchieva et al [Simidchieva 09], which also shows that different kinds of process models may be more suitable for different kinds of agents.

The net effect of the inherent imperfection of process models is that agents tend to deviate from them during process execution. Empirical evidence for that can be found in the study conducted by Lanubile and Vissagio [Lanubile 00]. They showed that most agents do not follow an assigned process 100% of time. This comes from the fact that developers may be confronted with unanticipated situations that are not represented in the process model. It can also be due to late and unexpected project constraints or to the developers’ tendency to accomplish activities according to their experience and intuitions. A deeper investigation on the consequences of deviations on the speed and quality of the work of developers was conducted by Visser et al in [Visser 90]. They observed expert engineers that defined a process model and then compared their actual process execution with the model they defined. The authors concluded that experts usually deviate from the process model and act opportunistically, using it as a guide in defining what should be done and how. Moreover, they concluded that this behavior does not necessarily reduce the quality of their work.

Finally, determining when deviations should be detected and presented to the process agent and when they should be forced to respond to them is also a problem. This problem is what we call Temporal Incompleteness and it is twofold. If deviations are detected and presented to the agent too late, a lot of time may have been spent working in the wrong direction, implying a loss of time and efforts. On the other hand, the obligation to handle deviations too early may be frustrating to process agents in a situation in which their deviating actions would be tolerable and harmless to the process objectives. This leads to the questions of when deviations should be detected and when should PSEEs force agents to deal with them.

The state of the art PSEEs handles the problem of detecting deviations in two ways. The first way is by looking for deviations in fixed milestones during process execution. For example, the PSEE might verify the correctness of the manipulated artifacts at the end of each activity. The weakness of this approach is that deviations may be detected too late with respect to the moment when they were committed, which may cause loss of time and money. The second way is by representing a model with every possible single atomic change that can be accepted over the artifacts. This avoids a large time delay between deviation and handling, but requires more detailed process models, which may reduce the applicability of the PSEE. Finally, regarding the problem of dealing with deviations, existing PSEEs usually force agents to handle deviations as soon as they have been detected which also reduces their flexibility.

Existing strategies for handling deviations in PSEEs, can be categorized into two groups. The first one is based on canceling or undoing the actions that are not compatible with the SPM and the second one is based on ignoring the actions that cause deviations and continuing the process enactment. The main disadvantage of the first strategy is that it can become very frustrating for the agents especially if they have no choice but deviating. A common occurrence of this problem happens when process agents face a situation that has not been foreseen in the SPM. If the PSEE forbids deviations, agents have no choice but to perform their tasks outside the control and the guidance of the PSEE. The second strategy, ignoring deviations, may also be frustrating to novice agents. This happens because these agents are not experienced enough to anticipate the impacts of their deviations to the outcomes of the process execution. If the PSEE ignores deviations, the agents may end up producing artifacts that do not correspond to what the process model expects [Almeida 10b].
2.2 Research Objectives

The objective of our work to face the above-cited issues was to add flexibility with the way PSEEs deal with deviations. To be precise about what is meant by flexibility, the main underlying hypothesis of this work was that deviations could be seen as inconsistencies in the process enactment. As defined by Spanoudakis and Zisman in [Spanoudakis 02], the management of inconsistencies can be organized into a set of activities, among which two are paramount in this work: the detection of inconsistencies and the handling of inconsistencies. In the context of the enactment of a process, detecting inconsistencies means identifying each deviation and its impact on process goals. To handle inconsistencies means proposing possible actions to deal with them.

Our first intended objective was called *Early Deviation Detection*. This means that we want to inspire ourselves in the artifact-based PSEEs [Cugola 98] to be able to detect deviations as soon as possible during process enactment. We aim to do that by monitoring all modifications to the artifacts used in process enactment and spotting deviations based on them. Nonetheless, detecting deviations as early as possible is not enough. The PSEE should also be able to classify the impact of each deviation on the process objectives. This would ideally allow the agents to focus their attention on handling the most risky deviations without being bothered by the less risky ones. This is what we call *Risk Assessment*, and this was our second research objective.

Regarding handling deviations, our first intended objective was called *Late Deviation Handling*. Given that deviations are detected early and that they are sorted according to the risk they imply, our next objective was to allow process agents to delay the effective handling of the less risky deviations as long as possible. This was done by contributing to the state of the art with a formal framework that helps PSEEs in deciding to which extent deviations can be tolerated. Finally, our last objective was called *Correction Guidance*. The net effect of being able to tolerate deviations for a longer time is that agents would need to deal with many of them at the same time. As demonstrated by a recent work on the inconsistency management field, this is not a simple task [Egyed 08]. Our last research objective was then to be able to provide so-called *correction plans* to guide the process agents into handling deviations.

2.3 Deviations

**Definition 1** (Deviation). *Deviations are actions that violate the constraints present on the process model over the sequence of actions or over the artifact states these actions produce.*

Deviations may happen either because the process agent changed the structure of the manipulated artifacts in a forbidden way, or because he executed actions in moments in which they were not acceptable.

To make this clearer we can compare deviations with two other relatively similar concepts used in the literature: the concepts of exceptions and inconsistencies. As explained by Cugola et al. in [Cugola 96], deviations are related to actions performed by the process modelers during process enactment, while inconsistencies are related to the states of the process execution engine during the enactment. The concept of inconsistency refers to the concept of inconsistent state, i.e. states reached during process enactment that are not consistent with the process model. Deviations are the actions performed by the process agent that lead to inconsistent states.

Deviations are usually undesired events that should be eliminated from process enactment. Conversely, exceptions, as argued by Strong et al in [Strong 95], are intrinsic to organizational processes and cannot be considered as errors to be eliminated. That is why exceptions are included as part of the process model, while deviations are not.

In summary, there is an intersection between the concepts of inconsistency and deviation and of inconsistency and exception but not between deviation and exception. Deviations represent violations to the process model while the exceptions represent exceptional situations included in the process model; therefore there is no intersection between these concepts. However, nothing stops deviations from happening during an exception handling, in the case an exception handler is not executed as it has been specified by the process model. Finally, inconsistencies may be considered as exceptions and therefore be included in the process model or deviations otherwise.

The focus of this work is in the detection and handling of deviations, instead of exceptions. Our objective is to provide PSEEs with means of handling these unexpected and hence non-modeled situations and to reduce the risk implied by them. The next two subsections will explore in more details with the detection and handling of these deviations.
Categorizing the different kinds of deviations

In order to perform its role, the PSEE keeps track of four pieces of information: (1) the actions performed by the process agents, (2) the current activity being executed, and (3) the current structure of the manipulated artifacts. Actions modify either the current activity (by starting or finishing it) or the state of the artifacts (by creating or modifying them). Two important features of actions are that they are performed by process agents and that they are ordered in time. The process model constrains the order in which actions can be performed, which role is allowed to execute each action, and the modifications they can perform over the current activity and the structure of the manipulated artifacts.

Using the process model depicted in Figure 2.1 as a motivating example we can categorize deviations into four groups. This classification is important in understanding the objectives of our work and our contributions, presented further in this chapter.

**Figure 2.1. Example Software Development Process Model**

**Micro Behavioral Deviations:** These deviations concern every violated constraint over the actions performed by a process agent and the structure of the artifacts during the execution of a single activity. A possible negative consequence of a micro behavioral deviation is that the developer may need to redo her work, which may represent a big waste of time.

In our example, consider that the process model states that during the code activity developers have only the right of changing the sourceCode artifact. A micro behavioral deviation would happen if, during the execution of the code activity, a developer tries to modify the designModel. That means that the developer is not working on the code and therefore her actions may delay the process enactment. The detection of such deviations avoids this delay.

**Macro Behavioral Deviations:** These deviations concern every constraint violation over the execution order of process’s activities. This happens, for example, when activities are not executed in the specified sequence, or when some of them are omitted or postponed. This can be due to unexpected project constraints (e.g., the need for an artifact in priority) or by omitting the execution of an activity due to the lack of time.

The main risk of such deviations is that they may lead the process execution into a blocking state. For instance, the process execution may reach a state in which an activity can’t start because one of its inputs is not available due to the fact that the activity that was supposed to deliver it was omitted or postponed. In order to illustrate this kind of situation in our sample process, suppose that a developer wants to execute the code activity before the design activity. Since the design activity has not been executed before, the code activity needs to be performed without a designModel. The quality of the produced sourceCode may be negatively impacted by the absence of the designModel, if the process model expects, for example, to obtain a sourceCode that corresponds to the designModel.

**Structural Deviations:** These deviations concern every constraint violation over the structure of artifacts. In our example, this would happen if no designModel was produced after the design activity; or if the produced model does not conform to a given metamodel (e.g., if it does not conform to the UML 2 Class Diagram well-formedness constraints) or to some proprietary business rules.

Notice that there is a correlation between structural and micro behavioral deviations. This is the case when the process model defines constraints over the expected structure of an artifact during a given process activity. Making the difference between both categories of constraints is important.
because some deviation detection approaches only support detecting micro behavioral deviations that are not structural and others only detect structural micro behavioral deviations.

The effect of a structural deviation is that the activities having an inconsistent model as an input will not really start until the inconsistency is resolved. Indeed, the developer assigned to such activities will spend a part of his allowed activity time trying to resolve another developer’s errors, which can be a tedious task; or he can ask for a rework of the previous activity. In both cases, this can disrupt the project’s organization thus inducing a waste of time and resources. Detecting structural deviations can be of great help to developers since it represents an additional means to make sure that what they produced in an activity is reusable by the following process’s activities.

Organizational Deviations: These deviations concern every constraint violation over the actions a single process agent that is authorized to perform. Such deviations may have a negative impact on the outcomes of the process if, for example, agents without the necessary skills perform activities they were not capable of or allowed to. In our sample process model, suppose that the process modeler states that agents with the designer role are in charge of the design activity and that agents with the developer role are in charge of performing the code activity. An organization deviation would happen if someone without the designer role performs the design activity and produces the designModel artifact. This may result in a lower quality designModel if this person doesn’t have the necessary skills to execute this activity properly.

In terms of deviation detection, the objective of our work was to propose a PSEE that automatically detects deviations that are part of each of the four above-cited categories.

2.4 Approach Overview

In this section we present our research objectives, and for each of them, the challenges that were faced along with the approaches we proposed. There are four objectives; two concerning the detection of deviations, namely Early Deviation Detection and Risk Assessment; and two concerning the handling of deviations, namely Late Deviation Handling and the Correction Guidance.

Deviation Detection

First Research Objective: Early Deviation Detection

Most current PSEEs delay the detection of deviations to some fixed milestones during process enactment. Two common milestones are at the beginning and the end of each process activity. At these milestones the PSEEs can detect if input and output artifacts correspond to what is required in the process model. Our first research objective consisted in allowing PSEEs to detect deviations not only at these milestones but also throughout the execution of process activities.

Challenges: What happens during a process activity can be described by means of the states assumed by the artifacts during its execution (structure) and by the actions performed by process agents during the process’s activity (behavior). The first challenge lays in the fact that process modeling languages are usually limited to defining the control flow of process’s activities, the roles, and artifacts that are associated with each activity. Therefore SPMLs do not allow process modelers to formally define how process agents are supposed to execute each activity. This is usually the role of textual guides that are associated to process activities and describe in natural language how process activities should be executed. The second challenge is derived from the first one. Since process modeling languages do not allow process modelers to define formally how activities are going to be executed, PSEEs are not able to verify the actions performed by process agents within each activity.

Contributions: In order to face both challenges, we need to: 1) provide means to process modelers to define how process’s activities should be executed and 2) provide means to the PSEE to verify what is actually being executed by the process agents. We transformed the problem of defining how activities are supposed to be executed to the definition of which actions are allowed (or not) during each process’s activity and which constraints should be enforced over it.

As a fixed vocabulary for the actions allowed during a process activity, we used Praxis [Blanc 08]. Praxis is a formalism developed by our team for representing artifacts by means of the sequence of atomic actions that have been performed during its construction. It represents every action that can be performed over an artifact by means of six fixed atomic actions which has already been applied to different kinds of artifacts, such as models [Blanc 08, Almeida 10c] and
source code [Falleri 14]. The choice of Praxis was based on the assumption that process activities consist mostly on the manipulation of artifacts, either in the construction of output artifacts or in the recording of results of non-artifact manipulation actions (e.g. recording the results of a testing activity). Praxis actions also offer the advantage of having a simple and well-defined semantics.

In order to describe constraints over the sequence of actions and over the structure imposed by them we developed a logic rule-based language called PraxisRules which was initially published in [Almeida 10a]. The definition of this language has been motivated by the lack of a language for specifying the relationship between process activities and actions directly performed by process agents during their execution. PraxisRules is therefore a Domain Specific Language (DSL) dedicated to the problem of specifying which actions are acceptable or not during the execution of each process activity in a process model. It can also be used to define structural constraints, and therefore to detect structural deviations. This rule-based language has been developed as part of this work and has been used in the context of the MOVIDA ANR [Movida 08] project by engineers at THALES Group to verify structural constraints over industrial models. The results of this experimentation have been published in [Le Noir 11]. It has also been included in the context of two other research projects, namely Monoge [Monoge 13] and MeRGE [MEERGE 12].

Second Research Objective: Risk Assessment

Most PSEEs do not provide an assessment of the risks implied by the deviations they detect. Our second research objective is to provide this feature, during process enactment to prevent possible issues in the process realization, which could lead to a blocking situation.

Challenges: The main challenge in accomplishing this objective lies in the fact that this is a very subjective measure that may vary from one process modeler to another. This measure also depends highly on the context of the specific project in which the process is being enacted.

Contributions: In order to cope with the subjective aspect of risk measurement we provide an approach that does not depend on specific risk measures, rather, two simple requirements are utilized: (1) different risk levels should be defined during process modeling and (2) there should be at least a partial order between them. This choice has been motivated by the fact that this research objective aims, at providing a means for process agents to compare multiple detected deviations according to their risk level and therefore to choose which deviations should be handled first based on this information.

In order to allow the risk assessment to be depending on the context of process enactment, we first need to allow the process modeler to define what is the risk implied by each deviation. The main difficulty in doing that is with enumerating all possible deviations from a process model. In order to cope with this problem we proposed an approach for mapping SPMs into a set of rules, called Process Rules [Almeida 11b].

Each process rule is defined in such a way that each possible deviation to the process model corresponds to a single process rule. The process agent is then able to associate a risk level to each process rule, and the PSEE is then able to compare deviations according to their risk by comparing the risks associated to the rules that detect them. A different set of rule-risk associations can be proposed by the process modeler and fine-tuned by the process manager during the process enactment. This will allow for different risks in different contexts. The set of detected deviations along with their associated risk levels is condensed into a single report, called Deviation Report.

Summary of Deviation Detection Approach

A summary of our approach for accomplishing our objectives related to deviation detection is displayed in Figure 2.2. Two actors are involved in this approach: the process modeler and the PSEE. The proposed approach happens in five phases, two of them executed by the process modeler, and therefore manual, and two of them executed by the PSEE, and therefore automated.

The first step (1) consists of having the process modeler define a process model which is annotated with PraxisRules constraints. The second step (2) consists of taking a process model and extracting the set of process rules from it. This process is automated and extracts a set of rules. The process modeler then needs to manually associate a risk level to each process rule (3). During process enactment, the PSEE monitors (4) the actions (represented in the form of Praxis actions) performed by process agents and uses process rules to compute the so-called Deviation Report (5), which simply lists every detected deviation along with the risk level associated with the process rule that detects each of them.
Deviation Handling

First Research Objective: Late Deviation Handling

Our first objective in terms of deviation handling consists of providing an active support from the PSEE in allowing process agents to deviate from the process specification and, as long as they do not represent a threat to the process continuation, to repair these deviations later. It can be reformulated into deciding when the process agent should be forced to handle a deviation.

Challenges: PSEEs introduced in the literature do not allow process agents to delay the handling of deviations. This means that deviations need to be handled as soon as they have been detected or they are just ignored by the PSEE. That happens because PSEEs do not keep track of previously detected deviations and therefore, are of no help to process agents that in the future want to correct these deviations as well as control the risks implied by the deviations that have not yet been corrected. The first challenge is to be able to keep track of these previously detected deviations and of the risks they represent to process execution.

Nonetheless this does not solve the whole problem. PSEEs should also be able to decide which deviations can be tolerated or not and to which extent they are authorized to do so. This is necessary because, as stated previously, not all deviations are equal: some of them are more risky than others. While some deviations only reflect accidental differences in the way a process should be executed, others reflect important behaviors that should be avoided. Our objective is therefore to give the process modeler the possibility of defining which deviations are the most tolerable.

Contributions: The challenge of capturing every deviation and using this trace to decide which extent they are tolerable depends on the concept of Tolerance. So, motivated by the lack of clear definition for this concept, we formalized it. This was our first step. We extended our formalization with the concept of tolerance, so we could define clearly what it means to handle deviations and what it means to tolerate them before they have been handled. We materialized this new concept in a report called Tolerance Report which indicates, for each deviation if it is tolerable or not.

In order to deal with the challenge of deciding which deviations can be postponed and which deviations cannot, we defined an approach based on the so-called Tolerance Levels. Tolerance levels are pieces of information provided by the process modeler to allow the PSEE to decide to which point a deviation needs to be handled. The definition of levels has been motivated by the idea that some deviations are more tolerable than others. The notion of tolerance to deviations to a process rule therefore varies from “any deviation is forbidden” to “any deviation is allowed”.

The lowest tolerance level would represent the first situation (in which only what is defined in the process model is allowed) and the highest level would represent the second one (in which any execution is allowed). We defined two extra tolerance levels: the level in which a deviation is allowed until some precise point in the process enactment, and the level in which a deviation is allowed provided that it is handled before the end of process enactment. The number of tolerance levels and their precise semantics are not an essential part of the present approach. The only thing we consider essential is to be able to propose a mapping rule that translates process rules derived from a process model into relaxed rules, which indicate when deviations rule are tolerable to each. We consider it essential because it is the basis of our definition of tolerance.
Second Research Objective: Correction Guidance

The second objective consists of providing automatically derived correction plans for helping the process agents into correcting these deviations. It can be reformulated into searching for sequences of actions that correct deviations.

Challenges: Late Deviation Handling implies that the PSEE needs to be able to manage enactment traces containing multiple deviations. It turns out that this problem is much more complex than the problem of proposing corrections for a single deviation. On the one hand, if we suppose that there is only one deviation to be corrected, we can suppose that the process state differs from the expected state by the effects of a single deviation. On the other hand, if we need to consider multiple deviations, we need to deal with a process state that can be arbitrarily different from the expected state. Hence, complex plans may be necessary in order to bring the process agent back to a state that is closer to an expected state.

The second challenge in proposing correction plans is that the number of possible plans that could possibly correct detected deviations is infinite. This creates the problem of carefully limiting the number of generated plans, in order to not overwhelm the process agents. The problem may also be understood as the problem of correctly deciding which subset of possible plans would be the most useful to process agents.

Contributions: The first problem we decided to handle was the need to compute plans for correcting multiple deviations. We looked for planning algorithms that are commonly used in such complex situations. Two extra requirements for this algorithm were being able to support heuristics to limit the number of considered plans and to compare plans in terms of usefulness. Based on these requirements, we proposed a planning algorithm that extends the so-called Iterative Deepening Depth First Search Strategy (IDDFSS) [Russel 03]. This algorithm comes from the artificial intelligence applications and is able to fulfill our requirements.

A particularly important feature of this algorithm is the ability of fine-tuning the search for correction plans with heuristics for the selection of which deviations have more priority in handling and which solutions should be executed in priority. In order to define such strategies we inspired ourselves firstly on the concept of Generator Functions already present in the inconsistency management domain [Egyed 08, Almeida 10c] in order to decide on which correction plans are the most suitable for correcting detected deviations and on output of our risk assessment objective to decide upon which corrections are more desirable than others. A correction plan is most desirable if it leads to a situation in which the most risky deviations in the current state have been handled.

An important feature of Generator Functions is that they are independent of the process model being enacted, and therefore can be reused in other process models. They are also defined by means of rules that are similar to well-known situation-response rules, yet more powerful. They can therefore be fine-tuned by process modelers to meet their needs.

Summary of Approach

A summary of our approach for deviation handling is presented in Figure 2.3. Similarly to our approach for detection of deviations, two actors are involved in this process: the process modeler that manually defines the tolerance levels and generator functions; and the PSEE that monitors the process execution and proposes correction plans.

This approach is organized into five steps. The first step consists of defining the generator functions for the context of the process model (1), i.e. the process modeler needs to define the possible actions performed by process agents and what are the possible correction plans in case of these actions cause deviations. The second step consists of defining the tolerance levels associated with each possible deviation to the process (2). These levels would be associated to each process rule, similarly to the risk level defined in the previous section. Finally, the PSEE would allow process agents to enact the process model, observe the agents’ actions (in the form of Praxis actions) (3) and use the process rules to detect deviations and to produce the deviation and tolerance reports (4) containing the deviations that were detected, corrected and the deviations that are still tolerable or not. The PSEE would then use the risk assessment associated to each detected deviation and the generator functions to propose correction plans prioritizing riskier deviations. This would be done by means of our proposed planning algorithm (5) that is based on the use of the IDDFSS algorithm and of the Generator Functions.
2.5 Overall PSEE Architecture

This section presents the architecture of our PSEE that takes into consideration our contributions for detecting and handling deviations presented earlier in this chapter. This architecture is composed of 5 components and is summarized in Figure 2.4 and described below:

**Process Execution Engine (PEE):** This component is responsible for taking a process model and providing the process agents to an environment in which they are able to execute the development activities in the process model. More specifically, this environment consists of a user interface that indicates the correct order of execution of the process activities, as well as the artifacts required and produced by each process activity. Additionally, this environment should start the external tools that should be used by process agents to execute the development activity. For example, it should start a modeling tool in modeling activities, or a development environment in case of development activities. As an extra task, this component also should keep track of the current activity being executed, and then adapt to the provided user interface accordingly.

**Action Listener (AL):** This component monitors the actions executed by the process agents in the environment provided by the PEE and produces the corresponding enactment trace. Since process agents use different tools to produce different artifacts, a different AL needs to be provided to each tool used by agents.

**Rule Extractor (RE):** This component takes input to the process model and generates the equivalent set of process rules.

**Deviation Detection Engine (DDE):** This component takes as input the process rules generated by the RE, the PraxisRules and the risk and tolerance levels associated with each process rule by the process modeler. It uses all this information to produce the tolerance and deviation reports that should be used by the process agents to manage the deviations that have already been detected, corrected and tolerated.

**Plan Generator (PG):** This component takes as input the output of the DDE and the set of generator functions provided by the process modelers. These inputs are used to derive correction plans that, when performed by the process agents, reduce the overall risk represented by the currently detected deviations.
2.6 Results, Discussion and Perspectives

Regarding the validation aspects, a lot of efforts were made in that sense. First the approach was implemented in a prototype written in Java and Prolog. This prototype was integrated with the Eclipse development environment, the SVN versioning environment and with different Bug Trackers. This prototype implementation allowed testing (i) the expressiveness of PraxisRules by implementing four different industrial software processes in our tool: the Eclipse OpenUP process and three different processes used by the open source projects ArgoUML, Openbravo and Joomla. The Eclipse OpenUP process in particular was used in a case study in which we intended to stress test our four research objectives with a deeper evaluation [Almeida 12]. Also, the work of Marcos presenting the different aspects introduced earlier in this document (and detailed in section 2.7 for the interested reader) was published in two journal papers, one being a top ranked journal SPE (Am[Falleri 13][Almeida 13], and 8 conferences, among them ASE, CAISE, MoDELS, EDOC, and ECMFA.

(ii) The usefulness of our approach was evaluated in two empirical evaluations. The general purpose of the first empirical study was to measure whether or not process and artifact guidance do provide efficiency and quality benefits for developers. At this aim, we gave our tools to novice modelers and compared the time and quality results of a group using our tool and a group without it. Even though our results were not statistically representative (small population), in our experiment, people in the group using our tool produced better quality work and required less time to finish their work than the control group (without our tool)[Almeida 10b]. In a second empirical evaluation, we exploited our integration with Bug Tracking tools to show that our tool can be useful in detecting risky deviations in historic bug tracking repositories. We showed a positive correlation between the number of developer deviations detected by our tool and the existence of risky situations in these repositories (the more developer deviate from the bug tracking and handling process, the more the bug takes longer to resolve or is reopened). We investigate 29,000 process execution traces in order to evaluate if our PSEE is able to detect deviations that are risky to process objectives [Almeida 13]

For the reader who is interested to have a deep dive into the “How?” we achieved our research objectives, the next section is aimed for that purpose.

2.7 A brief yet deep dive in the approach

In the following, I will go through the objectives set in the beginning of this chapter and give you a brief yet precise description of the work we accomplished to achieve them. The description will follow the structure given earlier in the section “Research Objectives”, i.e., presenting our contributions on Deviation Detection, following by Deviation Handling. But first, I will introduce the building block of the approach, our DSL, PraxisRules.
PraxisRules

PraxisRules is a rule-based logic language we defined in order to allow process modelers to specify the allowed actions during an activity in a process model. The process modelers may also use this language to specify structural constraints over the manipulated artifacts that should be enforced during the execution of a given process activity.

In Figures 2.5 and 2.6, we present the metamodel of PraxisRules. Figure 2.5 presents the part of the language dedicated to detecting structural deviations and Figure 2.6 presents the part of the language dedicated to detecting micro behavioral deviations. Since it is a rule-based language its most general concept is the one of RuleSet which aggregates rules. These rules are said to be in the same context, i.e., they refer to metaclasses from a set of imported metamodels (represented by the association importMetaModel) and can refer to rules defined in other rule sets (represented by the association importRuleSet).

There are two kinds of rules in this language: the Structural rules, represented by the Rule metaclass, and the Behavioral Rules, represented by the TemporizedRule metaclass. The first one analyzes the structure of an artifact in a given moment (e.g. before, during or after the execution of an activity) in order to detect structural deviations. The second kind of rules analyzes every event detected during the execution of an activity in order to spot micro behavioral deviations. Notice that the later are subclasses of the former because behavioral rules can impose behavioral constraints that depend on structural considerations, while structural rules are limited to structural constraints.

The semantics of PraxisRules supposes that the PSEE collects a sequence of actions performed by the process agents, called. Process enactment Trace. Actions in these sequences are described as Praxis atomic formulae. There are six kinds of atomic formulae in Praxis: create(e,c) and delNode(e,c) represent the creation and destruction of an element e of metaclass c; addReference(e,r,f) and remReference(e,r,f) represent the addition and removal of a reference between e and f called r and addProverty(e,a,v) and remProverty(e,a,v) represent the addition and removal of a value v to the property a in e. Furthermore, for each Praxis action a, the atomic formula last a represents the last action of the kind a in the current process enactment trace. This simple vocabulary of actions is sufficient for representing any modifications over an artifact (set of artifacts) during the execution of an activity in a process model [Falleri 14].

In PraxisRules, each action is also called an event. By looking at the metamodel we notice that, structurally, the only difference between the two kinds of rules lies in the temporizer attribute. In fact, while structural rules are always evaluated taking into consideration the state of the artifacts, the behavioral rules are always evaluated by taking into consideration a specific event in the process enactment trace. The attribute temporizer names this specific action so that it can be mentioned inside the definition of the rule.

Each rule is defined as a Formula that defines a query over the Process Enactment Trace, captured by the PSEE. While structural rules are limited to querying the Praxis actions (represented by the PraxisQuery metaclass) found in this trace in order to understand the structure implied by them, the behavioral rules involve temporal formulae, which manipulate Event’s in the Process Enactment Trace. In a rule, Event’s with the same name as the temporizer of the rule match this action.

As mentioned earlier Structural Rules are the rules used to verify structural deviations. Such rules represent formulae that identify patterns in the sequence of Praxis actions. Syntactically, structural rules follow the general syntactical pattern ruleName(A0,...,An)<=>formula, where ruleName represents the name of the rule, A0...An represent a set of variables called head variables and formula is a query to the process enactment trace. ruleName(V0,...,Vn) is said to hold if there is an answer to the query formula by replacing the variables in the head of the rule by the values V0,...,Vn.

Formulae may be atomic or composite. CompositeFormula represent logical connectives such as conjunction (represented as and(F1, ..., Fn)), disjunction (represented as or(F1, ..., Fn)) and negations (represented as not(F1)) where F1, ..., Fn are formulae. Atomic formulae can be divided into three groups: the PraxisQuery, the LibraryFormula and the RuleCall. The first group represents the atomic Praxis events in the process enactment trace,
following the same syntax adopted by Praxis. The second group represents built-in events and formulae whose semantics is provided by the PSEE. For example, the formula `sum(A, B, C)` would represent the atomic formula that computes the sum of A and B and puts the result in the variable C.

Finally, the `RuleCall` represents calls to other rules.

Let us now illustrate these concepts in a more concrete example.

**Example 1 (Structural Rule).** Suppose that our process model contains at least two activities. In the first one the process agent needs to create three packages in a UML model called Model, View and Controller. In the second one he needs to create the model classes in the Model package. A structural constraint in this process could state that the View and Controller packages should remain empty during the second activity.

Let us call the second activity `fillModelPackage`. If we want to verify if, during the execution of the second activity, the packages View and Controller exist and are empty we would define it by means of the following PraxisRule:

```
fillModelPackageStructuralInv() <= and
create(View, #package),
addProperty(View, #name, "View"),
addProperty(Controller, #name, "Controller"), not{
addReference(View, #element, ElementView)
not{addReference(Controller, #element, ElementView)}
}
```

The name of this rule is `fillModelPackageStructuralInv`. It holds if the packages called “View” and “Controller” exist and are empty. In order to verify this rule, the PSEE needs, every time a new action is detected, verify if this rule still holds. If it doesn’t hold, a deviation (whose cause is the last observed action) is detected. Syntactically this rule is described as a query that is a composite formula which is a conjunction of four atomic formulae and two `CompositeFormula`. The `PraxisQuery create` and `addProperty` verify if the packages exist and the two negated `PraxisQuery addReference` make sure they do not contain any other element.
In structural rules, the semantics of a PraxisQuery is always equivalent to its last counterpart in Praxis i.e. it always looks for the last action in the trace that matches the query but whose effects have not been canceled by later actions. For example, the atomic formula create(View, #package) searches for the last creation of a package in the model sequence and matches the identifier of the package being created with the variable View. The atomic formula addProperty(View, #name, "View") searches for the last addition of a value to the name property of the element View. This operation should add the value “View” to that property. Both formulae are respectively equivalent to lastCreate(View,#package) and lastAddProperty(View,#name,"View") in Praxis.

The Behavioral Rules are logical rules that verify the behavior of the process agent during the execution of an activity. As a complement to the purely structural rules we define the metaclass TemporizedRule (c.f. Figure 4.2) which adds an attribute, called temporizer and the possibility of using temporal operators and formulae in its query. A behavioral rule validates events in the enactment trace. Syntactically, it follows the pattern ruleName(A0,...,An) @ temporizer <=> formula, where, similarly to structural rules, ruleName represents the name of the rule, A0,...,An represent a set of variables called head variables and formula is a query to the process enactment trace.

The query of a temporal formula may refer to events, which are atomic formulae marked with a temporal variable indicated by the use of the@ signal. For example, the atomic formula creation@ create(C, #class) refers to the creation of a class C as an event called creation. Temporal operators are then used to define the allowed order of events in the enactment trace. The following operators have been defined:

- before{E1,E2}: the event E1 happens before E2 in the enactment trace;
- after(E1,E2): the event E1 happens after the event E2 in the enactment trace;
- first(E): the variables in formula E match to the first time such event happens in the enactment trace;
- last(E): the variables in formula E match to the last time such event happens in the enactment trace;
- between{E1,E,E2}: the event E happens after the event E1 and before the event E2 in the enactment trace.

Let us now illustrate these concepts in two more concrete examples.

Example 2 (Behavioral Rule – Verifying Order). Let us consider an activity in a process model during which the process agent is supposed to create a Model package and a set of classes within this package. The following rule, called fillModelPackageBehavioralInv, shows an example of how to specify that the classes in the model package should be created before the creation of the model package. A more concrete example could be made in the context of Test Driven Development where tests should be specified before the pieces of code supposed to successfully pass these tests:

```
fillModelPackageBehavioralInv() @ packageCreation <=>
```
implies { packageCreation @ create(ModelPackage, #package),
and {
    addProperty(ModelPackage, #name, "Model"),
    addReference(ModelPackage, #element, SomeClass), before {
        classCreation @ create(SomeClass, #class), packageCreation @
        create(ModelPackage, #package)
    }}
}

This rule validates the action of creating the model package represented by the event named packageCreation. It states that if the package represented by the variable ModelPackage is called Model and contains the class SomeClass, the creation of the class SomeClass should appear before the creation of ModelPackage in the process enactment trace.

In order to evaluate such rule, the PSEE matches every action performed by the process agent during the fillModelPackage process activity to the temporizer and evaluating if the formula is verified. If not, a deviation is spotted.

Example 3 (Behavioral Rule – Verifying Actions). This second example shows a rule, called fillModelPackageOtherBehavioralInv, that is used to specify which actions are allowed during the execution of the activity described in the preceding example.

fillModelPackageOtherBehavioralInv() @ operation <=>
or { operation @ create(SomePackage, #package),
    and {
        create(SomePackage, #package), or {
            operation @ addProperty(SomePackage, Prop, Val), operation @ remProperty(SomePackage, Prop, Val), operation @
            addReference(SomePackage, Ref, Elem),
            operation @ remReference(SomePackage, Ref, Elem), operation @ addReference(Elem, Ref, SomePackage), operation @ remReference(Elem, Ref, SomePackage)
    }}
}

This rule, called fillModelPackageOtherBehavioralInv defined a behavioral rule associated to an activity in which the agent should create and modify packages. Differently from the fillModelPackageBehavioralInv rule presented before, this rule does not define the order in which actions are expected to happen but only which actions are allowed to be executed.

The @operation part acts as a temporal variable for the operation executed by the agent and corresponds to the temporizer attribute in the metamodel. According to the rule, the operation executed by the agent (named operation) is either the creation of a package called SomePackage (represented by the action create(SomePackage, #package) or one of the following six operations involving SomePackage (addProperty(SomePackage, Prop, Val)or remProperty(SomePackage, Prop, Val), etc.)

As a final remark over the expressiveness of this language, it is important to note that the level of granularity of these rules may be fine-tuned to give either a more descriptive or a more prescriptive flavor to the process.

For example, we could have specified this activity invariant in different levels of abstraction:

- Only an action create(SomePackage, #package) immediately followed by an action addProperty(SomePackage, #name, "Model") is allowed;
- Only actions creating and modifying packages are allowed;
- Only actions creating new elements and modifying elements during this activity are allowed.

This means that even if the language is powerful enough to prescribe the order in which the
actions may be executed in a very low level of abstraction, this level may be adjusted according to
the objectives of the process modeler. In fact, this is the main advantage of PraxisRules when
compared to UML2 Activity based languages: in UML2 one could express the expected behavior
during a process activity by means of an activity diagram based on Praxis-like actions, but this new
diagram would be limited to a very prescriptive and low level description of the required behavior,
while the level of granularity of a PraxisRules specification can vary according to the needs of the
process modeler.

It is also worth noticing that PraxisRules can be extended in order to capture more details about
developers’ actions performed during process execution. This is what we demonstrated by extending
PraxisRules in the context of multi-view based development processes involving different developers
working on different viewpoints at a time [RM-ODP 95]. The action trace was extended with the
notion of “timestamp” and “viewpoint” so we could capture at what time a given viewpoint was
modified and to check if this would trigger any inconsistency with what other developers are doing in
the other viewpoints. The RM-ODP (Reference Model of Open Distributed Processing), a multi-
view-based standard for the specification of distributed systems was taken as an example for
validation purposes since the development process using RM-ODP involves five complementary
viewpoints and a set of consistency rules to be enforced between the different viewpoints during the
development process. Thus, the action trace create(me, mc, t, v) would refer to the creation
of a model element me, that is an instance of the metaclass mc at the timestamp t and in the viewpoint
v. One can also decide to include the role that performed the action.

Early Deviation Detection

This section presents our approach for deviation management during process execution. As said
before, the objective of this approach is to allow process managers to keep track of the deviations that
eventually occur during process enactment and of their consequences, namely, the actions that should
be performed by agents to cope with these deviations. In order to do that, we defined the notions of
tolerance and mitigation. A deviation is said to be tolerated while it can be temporarily ignored by
the project manager and mitigated when its effects have been canceled by actions executed by the
process agent.

Our approach is based on the use of two reports called deviation and tolerance reports that are
computed by the PSEE and provided to the process agent and managers during process execution.
The first one lists every deviation detected so far while the second one lists the deviations that are
still tolerable and the ones that are not. The set of tolerable deviations that have been mitigated
consists of the set of deviations that appear in the deviation report but not in the tolerance report. A
risk level can be also affected to the different deviations to help the project manager classifying them
and assessing their impacts.

To define both reports, we decided to use a formal basis. This allowed us to define the involved
concepts unambiguously and to derive our implementation for them directly from our formal
framework. We decided to base our formal definitions on the Linear Temporal Logic (LTL) [Pnueli
77]. This decision has been based on its recent use on reasoning about software and business
processes [Knuplesch 10], [Yang 07], [Smith 02]. This language provides the constructs needed to
represent the log of process states and process agent actions observed during process execution.
Moreover, it provides the temporal operators required to define the temporal constraints implied by a
software process model that need to be verified by the PSEE at runtime.

In our work we introduced a set of formal definitions for all the concepts used in the detection and
handling of process deviation. The first one is the one of Software Process Model as given below

**Definition 2.** (Software Process Model). A SPM is a special kind of directed graph \((N, E, C, Pre, Post, Inv, Role)\) where \(N\) is the set of nodes; \(E\) is the set of edges; \(C\) is a set of
constraints; \(Pre, Post\) and \(Inv\) are functions that map nodes to sets of constraints \((N \rightarrow Set(C))\),
representing their respective pre, post-conditions and invariants; and \(Role\) is function that maps
a node into a set of constraints which determine which roles are allowed to execute it \((N \rightarrow Set(C))\).

In \(N\), there are three kinds of node: object, control and executable nodes as defined in UML.
Furthermore, in \(E\), there are two kinds of edges: object and control flow edges. Object flow edges
connect object and executable nodes and flow edges link control and executable nodes. Finally, Pre, Post, Inv and Role functions range only over executable nodes.

The example given in Figure 2.7 illustrates a software process model whose objective is producing the source code of an application by the means of an intermediary design model. It is going to be used throughout this section to illustrate our definitions. The idea behind the simplicity of the example is to focus on the issues in managing deviations during process execution rather than on the process itself. The process model consists in two process activities design and code that should be executed in the following order:

- **Design**: during this activity the agent should create a design model of the system, which would consist for example of a set of UML2 models (e.g. use case diagram, class diagram, activity diagram, etc.). This is specified by the object flow edge that goes from design to designModel.

- **Code**: during this activity, the agent should implement the design model constructed in the previous activity in a programming language. The outcome of this activity is the running source code of the application. This is represented by the object flow edges that go respectively from the designModel object node to the code executable node and from the code executable node to the sourceCode object node.

**Figure 2.7. Example Software Development Process Model**

This process model is represented by the tuple \((N, \mathcal{E}, C, \text{Pre}, \text{Post}, \text{Inv}, \text{Role})\) where:

\[
N = \{\text{initialNode}, \text{designModel}, \text{design}, \text{sourceCode}, \text{code}, \text{finalNode}\}
\]

\[
\mathcal{E} = \{(\text{initialNode}, \text{design}), (\text{design}, \text{designModel}), (\text{designModel}, \text{code}), (\text{design}, \text{code}), (\text{code}, \text{sourceCode}), (\text{code}, \text{finalNode})\}
\]

\[
C = \{\text{atLeastOneClassInDesign}, \text{createOnlyClassesOrModels}, \text{designerRole}, \text{developerRole}\}
\]

\[
\text{Pre} = \emptyset
\]

\[
\text{Post} = \{(\text{design}, \{\text{atLeastOneClassInDesign}\})\}
\]

\[
\text{Inv} = \{(\text{design}, \{\text{createOnlyClassesOrModels}\})\}
\]

\[
\text{Role} = \{(\text{design}, \{\text{designerRole}\}), (\text{code}, \{\text{developerRole}\})\}
\]

The example above introduces four constraints. Two of them (called respectively \text{atLeastOneClassInDesign} and \text{createOnlyClassesOrModels}) define a post condition and an invariant of the design node. The other two constraints (called \text{designerRole} and \text{developerRole}) define two roles designer and developer. The design activity should be executed by some agent with the \text{designerRole} while the code activity should be executed by some agent with the \text{developerRole}.

Using this process model as an example, the PSEE assumes two roles during its execution. The first role consists of guiding the process agent through the process. It would present both activities to the process agents and it would indicate that they should be executed in a specific order: design before code, and by process agents with the associated roles: designer and developer. During the execution of each activity, the PSEE would describe what needs to be accomplished by the agents during each of them. For example, during the code activity, it would provide the designModel as input to the user and indicate that the sourceCode needs to be produced and that the post
The condition associated to the design activity is respected at the end of its execution.

The second role of a PSEE is enforcing the execution of the process, i.e. avoiding actions that are not allowed by the process model. For example, during the code activity, it would not allow the agent to modify the designModel.

In order to play both roles, the PSEE needs to be able to observe the actions performed by the agents to provide a precise semantics for the process model and then to compare the observed actions with the semantics of the process model. These concepts are precisely defined in LTL in the following subsections.

A **Process Execution Trace** is defined as a sequence of sets of observations made by the PSEE over the process as executed by process agents. More precisely, during the execution of a software process model, observations are recorded as logical propositions, that represent what has been observed by the PSEE. Hereunder we formalized this notion as follows:

**Definition 3** (Process Execution Trace): For a SPM defined as \((N, E, C, Pre, Post, Inv, Role)\), a Process Execution Trace is a possibly empty finite execution trace \(s=s_1, s_2, \ldots\) whose states \(s_i\) are sets of observations \(s_i=[o_1, \ldots, o_n]\) such that \(o_i \in \mathcal{O}\).

The set \(\mathcal{O}\) contains propositions representing any possible observation. It is defined as the union of four distinct sets \(\mathcal{O} = A \cup M \cup X \cup H\). Each set of propositions is defined as follows:

- **A** is the set containing one proposition for each action that can be performed during the process enactment. It contains:
  - The propositions “begin” and “end” representing the initial and final control nodes in \(N\),
  - one proposition \(begin(n)\) and one proposition \(end(n)\) for each executable node \(n \in N\),
  - and one proposition for each Praxis action over the object nodes in \(N\), represented as \(PraxisActions(N)\);

- **M** is the set containing one proposition \(exists(n)\) for each object node \(n \in N\),

- **X** is the set containing one proposition \(executing(n)\) for each executable node \(n \in N\),

- **H** is the set containing one proposition \(holds(c)\) for each constraint \(c \in C\);

If we consider the process example given above (figure 2.7), the following sets of propositions will be used to describe the execution traces captured by the PSEE:

- \(A = \{begin, begin(design), end(design), begin(code), end(code), end\} \cup PraxisActions(N)\)
- \(M = \{exists(designModel), exists(sourceCode)\}\)
- \(X = \{executing(design), executing(code)\}\)
- \(H = \{holds(atLeastOneClassInDesign), holds(createOnlyClassesOrModels), holds(designerRole), holds(developerRole)\}\)

The following trace could be a possible execution on the process given earlier

\(s = \{begin\}, \{begin(design), holds(designerRole)\}, \{create(designModel, model), executing(design), holds(createOnlyClassesOrModels)\}, \{create(c1, class), exists(designModel), executing(design), holds(createOnlyClassesOrModels)\}, \{addReference(designModel, element, c1), exists(designModel), executing(design), holds(createOnlyClassesOrModels)\}, \{end(design), exists(designModel), holds(atLeastOneClassInDesign)\}, \{begin(code), exists(designModel), holds(developerRole)\}, \ldots\)
The next step consists in providing a precise semantics for the SPM so that a process execution trace can be compared to it in order to detect deviations. We consider that a process in fact defines a set of constraints or rules over the process execution trace. For example, if the process defines that the activity \( a_i \) should produce the artifact \( a_o \) as output; it means that, after the execution of \( a_i \), the artifact \( a_o \) should be observed in the execution trace.

The building block of our approach is what we call a process rule. The semantics of a process model is then given by the set of process rules that can be derived from it.

**Definition 4** (Process Rule): Let \( A, M, X \) and \( H \) be the sets of propositions defined for a process \((N, E, C, Pre, Inv, Role)\) as in **Definition 3**. A process rule is an LTL formula in the form \( G (p \Rightarrow q) \) where \( p \) is a LTL formula over \( A \cup X \) and \( q \) is a LTL formula over \( A \cup M \cup X \cup H \).

We classify the process rules by their form. They all follow the general form \( G (p \Rightarrow q) \), meaning that for every state \( s_i \), if \( p \) holds in \( s_i \), \( q \) should hold in \( s_i \). Four kinds of process rules are emphasized: the pre-condition rules, the post-condition rules, invariant rules and the control flow rules, which are a special case of post-condition rules. In these rules, notice that the temporal operator \( X p \) is used, meaning that if \( s_i \) is the current state, \( p \) should hold in the state \( s_{i+1} \).

Once we defined the notion of Process rule, we use that latter to generate the Process LTL Set which defines the semantics of a process model, i.e., it defines the set of process rules that describe every allowed process enactment trace for this process model. It basically contains three precondition rules for each object node, stating respectively, at the moment the process activity execution is initiated, which are the input artifacts that are required, the pre-condition constraints that should hold and the process agent roles that are allowed to execute this activity. It also contains two post condition rules for each object node, stating respectively, at the moment the process activity execution is finalized, which are the output artifacts that are required and which are the post condition constraints that should hold. Additionally, invariants in the process model are represented by means of invariant rules that should hold during the execution of a process activity. Finally, a control flow rule is added for each control flow edge, in order to detect deviations to them during process enactment. The Process LTL set is automatically generated from the SPM.

**Definition 5** (Process LTL Set): Let \((N, E, C, Pre, Post, Inv, Role)\) be a SPM and \( A, M, X \) and \( H \) the process proposition sets defined for it based on **Definition 3**. A process LTL set \( P \) is the set containing the following process rules:

- the process rule \( “G (\text{begin} \Rightarrow X (\forall \text{begin}(a))))” \), for the set of executable nodes \( a_i \in N \) such that \( \text{begin} \rightarrow a_i \in E \);
- the process rule \( “G (\text{end}(a) \Rightarrow X \text{ end})” \), for each object flow edge \( a \rightarrow \text{end} \in E \);
- a precondition rule \( “G (\text{begin}(a) \Rightarrow \exists \text{exists}(o))” \), for each object flow edge \( o \rightarrow a \in E \);
- a precondition rule \( “G (\text{begin}(a) \Rightarrow \exists \text{holds}(c))” \), for each constraint \( c \) such that \( c \in \text{Pre}(a) \);
- a precondition rule \( “G (\text{begin}(a) \Rightarrow \exists \text{Role}(a))” \), for each constraint \( c \) such that \( c \in \text{Role}(a) \);
- a postcondition rule \( “G (\text{end}(a) \Rightarrow \exists \text{exists}(o))” \), for each object flow edge \( a \rightarrow o \in E \);
- a postcondition rule \( “G (\text{end}(a) \Rightarrow \exists \text{holds}(c))” \), for each constraint \( c \) such that \( c \in \text{Post}(a) \);
- an invariant rule \( “G (\text{executing}(a) \Rightarrow \exists \text{holds}(c))” \), for each constraint \( c \) such that \( c \in \text{Inv}(a) \);
- a control flow rule \( “G (\text{end}(a) \Rightarrow X (\forall \text{begin}(a)))” \), for the set of executable nodes \( a_i \in N \) that can appear as the next state of a state in which \( a \) was executed.

As an illustration of this concept, the representation of the process model introduced earlier (cf. figure 2.7) in terms of a process LTL set would be as follows

<table>
<thead>
<tr>
<th>Process Rule</th>
<th>Kind</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin ( \Rightarrow X ) begin( (\text{design}) )</td>
<td>process begin</td>
<td>initialNode</td>
</tr>
<tr>
<td>( G (\text{end(code)} \Rightarrow X \text{ end}) )</td>
<td>process end</td>
<td>finalNode</td>
</tr>
<tr>
<td>( G (\text{begin( (\text{design}) \Rightarrow \exists \text{designerRole}) } )</td>
<td>precondition</td>
<td>design</td>
</tr>
</tbody>
</table>
G (end_design) ⇒ X begin_code
G (end_design) ⇒ exists.designModel)
G (end_design) ⇒ holds(at_least_one_class_in_design)
G (executing_design) ⇒ holds(create_only_classes_or_models)
G (begin_code) ⇒ holds(developer_role)
G (begin_code) ⇒ exists.designModel)
G (end_code) ⇒ exists(source_code))

At this point, the PSEE needs to verify the observed process execution trace against the semantics of the process model in order to detect deviations. The set of deviations found in a process execution trace is called a Diagnostic Set. Each element in a diagnostic set represents a possible deviation according to the process rules in a process LTL set P.

**Definition 6 (Diagnostic Set):** Let $s$ be a finite process execution trace and $P$ be a set of process rules. The diagnostic set of $P$, called $D$, is the set of triples $(r, i, d)$ such that:

- $r \in P$, $r = G \left( a \Rightarrow p \right)$
- $[s \models r] = d$ and $d = \perp$ or $d = ?$
- $i$ is the smallest natural number such that $\exists s' \models a \Rightarrow p$ for some process execution trace $s'$ starting with $s$

However, if and only if $begin$ is not executed at the first state of $s$, the diagnostic set is $(begin, 1, \perp)$.

A triple $(r, i, d)$ is the diagnostic for the $i$-th observation in the sequence $s$. For a process execution trace $s \models r$ is a formula that is true if the LTL formula $r$ holds in $s$ (conversely, $s \not\models r$ holds if $r$ does not hold in $s$). $[s \models r]$ is an expression that may evaluate to one of three possible values: $\top$, if $r$ holds in $s$ independent of the future actions executed over it, $\perp$ if it does not, and $? \perp$ if $r$ holds or not depending on the future observed actions. In case of deviations, there are thus two possibilities of interest. On the one hand, if $d = \perp$, that indicates that the rule $r \in P$ does not hold and will not hold for any possible continuation of $s$. On the other hand, if $d = ?$, that indicates that the rule $r \in P$ may hold or not depending on how $s$ continues.

These possibilities are the only ones of interest here because they indicate that a deviation to $r$ has happened or may happen in the future depending on the next actions. The diagnostic set therefore contains the set of deviations that have already been detected or that may be detected in the future, depending on the next actions performed by the process agents.

Let us consider again the SPM represented by the activity diagram represented in Figure 2.7, and the following set of process rules and execution trace:

$$r = G (begin(code) \Rightarrow exists(sourceModel))$$

$$[begin]\), \{begin(code)\}, \{end(code)\}, \{begin(developer)\}, \{end(developer), exists(sourceModel)\}$$

The rule $r$ specifies that the code activity can only be executed in the presence of the designModel. The process set $P$ defines that $r$ is the only rule to be considered with detecting deviations in this example, and finally, $s$ defines a sequence in which the agent starts the process execution, then executes the code activity without the designModel and then executes the design activity.

The agent is considered to be deviating from the SPM if the diagnostic set contains any element of the form $(r, i, \perp)$. The Diagnostic Set defines the degree of conformity to the SPM, in the sense that every deviation in the execution trace will be reported in it as a diagnostic of the form $(r, i, \perp)$. The diagnostic for $P$ is therefore $D = \{(r, 2, \perp)\}$. This happens because code is executed in the second state which requires, by $P$, the designModel to be detected in the second state, but is not. In the next section, we introduce how to assign a risk level on deviations according to their impact on the process continuity.
**Risk Assessment**

The PSEE needs to classify the detected deviations according to the risk level they represent. In our work we did not intend to present a classification of risk that can be adapted to any possible development project, instead, we just defined a mathematical formalization that describes the minimum requirements for any given classification.

We call \( R \) the set of risk levels adopted in a project. Our approach consists of associating to each deviation one element from \( R \), so that we can compare deviations by comparing their associated risk levels. Hence, the elements in \( R \) should be minimally comparable.

Mathematically speaking, the elements of a set are minimally comparable if this set is a **Partially Ordered Set**. That means that, for any \( R \), there should be a relationship \( \leq \) that is defined for each pair of elements in \( R \) and which is reflexive, antisymmetric and transitive.

The deviation report is then computed by extending each triple in the diagnostic set with the risk level associated to it by \( L \). In our work, we adopted three risk levels (low, medium, high). We therefore obtain the following deviation report:

\[
R_d = \{(r_1, 2, \bot, \text{medium}), (r_2, 3, \bot, \text{low})\}
\]

The main advantage of \( R_d \) over \( D \) is that now one can compare the deviations in \( R_d \) by comparing the risk levels associated to each deviation. One can then deduce that \((r_1, 2, \bot, \text{medium})\) is a deviation that requires much more timely attention than \((r_2, 3, \bot, \text{low})\).

**Late Deviation Handling**

This section presents our contribution for late deviation handling during process execution. We define it as the ability to tolerate deviations temporarily, so that process agents can delay their handling as long as possible. Instead of forcing the process agent to follow a deviation-free process execution, the PSEE detects every deviation and produces two reports: the **tolerance report** and the **deviation report**. The first one lists the deviations that have been detected so far, whereas the second one lists the deviations that are tolerable (or not) to continue with the process execution. The deviations that appear in the deviation report but do not appear in the tolerance report are considered as handled.

In order to have such report computed by our approach, the process modeler should specify to which extent a deviation is tolerable. As a first step, the process modeler decides the level of tolerance of each **process rule** \( p \) in the **Process LTL Set**.

Tolerance levels are organized in a linear scale from the lowest level (0) to the highest (3). This is intended to reflect the fact that the notion of tolerance varies from “any deviation is forbidden” to “any deviation is allowed”. The lowest tolerance level would represent the first situation, in which only what is defined in the process model is allowed, and the highest level would represent the second one, in which any execution is allowed. In order to represent intermediate situations, we defined two extra tolerance levels: the one in which a deviation is allowed until some precise point in the process enactment, and the one in which a deviation is allowed but it needs to be handled before the end of process enactment.

The number of tolerance levels and their precise semantics are not essential part of the present approach. The only thing we consider essential is to be able to propose a mapping rule that translates process rules derived from a process model into relaxed rules, which indicate to which extent a deviation to each rule is tolerable.

Let \( d \) be a deviation caused by an action \( a \). The deviation report contains \( d \) as soon as it has been detected. However, the content of the tolerance report and the response of the PSEE to \( d \) depends on the tolerance level chosen for it:

- **0th level** is the one in which \( d \) is simply not allowed. Therefore, it should appear as intolerable in the tolerance report and the PSEE has only two choices: forbidding the execution of \( a \) or undoing it immediately.

- **1st level**, the process modeler may define a milestone \( m \), such that \( d \) is tolerable. This deviation will then appear as tolerable in the tolerance report until the first state in which the \( m \) holds. From this state on, this deviation becomes intolerable.

- **2nd level**, no milestone is provided, meaning that this deviation will appear in the tolerance report until it is handled.
- **3rd level**, no milestone is provided but a mitigating condition $c$ is provided. The deviation will appear as tolerable in the tolerance report until the mitigating condition $c$ is established, upon which it will be considered as handled. When the condition $c$ is set to $\top$, any deviation is allowed.

Tolerance levels are formalized by means of a mapping, from a **process rule** into a **relaxed one**. This mapping is presented hereunder:

**Definition 7 (Process Rule Rewriting):** Let $r = G (p \Rightarrow q)$ be a process rule. Let $r_L$ be its rewriting to the tolerance level $L$ defined as follows:

- $r_0 = r$
- $r_1 = G (p \Rightarrow q R m)$ where $m$ is a propositional formula
- $r_2 = G (p \Rightarrow F q)$
- $r_3 = G (p \Rightarrow F (q \lor c))$ where $c$ is a propositional formula

Notice that this definition uses the LTL operators $F$ $p$ and $p R q$ meaning respectively that $p$ holds in some future state and that $q$ holds until the moment $p$ holds in the future. The mapping proposed by this definition formally states that at 0th level, process rules are mapped onto themselves. At the 1st level, they require the condition $q$ to be met before the milestone $m$. Level 2 process rules delay the constraint to be eventually respected, and level 3 process rules relax the original process rule with a condition $c$ that may cancel it. In order to illustrate this notion of Process Rule rewriting in order to add a tolerance level, let us consider the following example:

Two possible rewritings of the process rule $r = G (\text{begin} \Rightarrow X \text{begin} (\text{design}))$ would be:

- $r_2 = G (\text{begin} \Rightarrow F (X \text{begin} (\text{design})))$: This rule states that the design activity should eventually ($F$) be performed during the process execution.
- $r_3 = G (\text{begin} \Rightarrow F (X \text{begin} (\text{design}) \lor \text{holds} (\text{managerAuth})))$: This rule states that the design activity should eventually ($F$) be performed but it may not be necessary if this deviation is authorized by the manager.

At this point, the PSEE needs to classify the deviations that it detected into tolerable or not. It provides users the necessary information about the risk level implied by each deviation. Hereunder an example of the **Deviation Report ($R_d$)** and the **Tolerance Report ($R_t$)** as generated by the PSEE in case of two deviations are detected (rules $r_a$ and $r_b$ are violated). In this case for instance, and according to the tolerance levels set by the process modeler (during the process rule rewriting phase), the tolerance report states that the first deviation is still tolerable ($?$) and that the second one is not tolerated any more ($\bot$). The $\top$ symbol can be used to indicate that the deviation has been handled (a condition became true for instance).

$$R_d = \{(r_a, 2, ?, \text{medium}), (r_b, 3, \bot, \text{low})\}$$
$$R_t = \{(r_a, 2, ?, \text{medium}), (r_b, 3, \bot, \text{low})\}$$

Thanks to these reports (Deviations and Tolerance), the project manager can keep track of the different deviations that occurred during process execution, and most of all, can act accordingly and at the right moment to anticipate any blocking state of the process. The Tolerance report will help her to warn process agents and to pinpoint the deviations that are not tolerable and that should be handled immediately.

So, to sum-up in order to adopt our approach one has to follow this workflow:

- The process modeler defines the **software process model**.
- The process modeler automatically generates the **set of process rules** from the process model.
- The process modeler associates a **tolerance level** to every process rule and a **risk level**.
- The process agent executes the process model by the means of the PSEE.
- The PSEE continually computes the **deviation and tolerance reports** and displays them to the process agent.
However, spotting the detected deviations to the process agent, their risk and tolerance levels could be not enough, especially for novice developers that might need some hints to resolve the negative impact of their deviations on the process continuity. This is the issue we tried to address and that we present in the next section.

**Correction Plans**

In order to provide correction guidance either to the project manager or to the process agents, a planning algorithm is used to compute the set of possible correction plans that are presented to the process agent in case of deviation. A correction plan is a sequence of actions that, when executed, fixes some deviations found in the observed process. Computing these plans is an inherently incomplete process, because there are an infinite number of potential sequences to be considered as correction plans.

The present approach reuses the IDDFSS algorithm to explore these possibilities. This is a generic algorithm for finding the shortest path between two nodes in a graph. It requires three inputs: the initial node, called \textit{origin}; a function to determine the searched node, called \textit{objective}; a function to compute the neighbors of a given node in the graph, called \textit{neighbor}; and finally, the maximum depth in the search tree we need to look at, called \( k \).

In order to adapt this algorithm to the context of process enactment, we use the IDDFSS to navigate in a graph where each node corresponds to an enactment trace, and transitions correspond to appending a new action at the end of a given sequence. In order to compute correction plans for deviations, the search starts in the current process enactment trace, and finishes in a trace where all deviations have been handled. If the maximum search depth \( k \) is reached without finding a process enactment state which corrects all deviations, all traces in the deepest level are returned as options to the process agent, since they are supposed to contain solutions to some of the deviations.

To avoid the combinatorial explosion due to the large number of possible plans, two strategies are adopted in this work:

- First of all, the \textit{generator functions} are used to propose actions to be appended to a given state, based on the cause of the deviations currently detected. This avoids exploring the complete set of possible actions, and makes it more focused on the actions that intend to fix deviations. The concept of Generator Functions we used is an adaptation of the Generator Functions used in the Consistency Management domain [Egyed 08]. In the present context, we replace “inconsistency locations” by deviations causes, i.e. actions that cause deviations. Generator functions therefore define a mapping from deviation causes in a process enactment trace to correction.

- As a second strategy, the overall risk represented by the deviations present in a sequence is used to heuristically classify all possible correction plans. We consider that one solution is better than another if it has fewer high-risk deviations. If both have the same number of high risk deviations, the one with fewer medium risk deviations is better. If both have the same level of high and medium risk deviations, the one with the least risk deviations is better. They are not comparable otherwise. Notice that this is just a heuristic we have adopted in the current work. In different contexts other strategies for comparing proposed correction plans may be more suitable.

The planning algorithm therefore proceeds in three steps. In the first step, a set of process rules is used to detect deviations, along with their causes and risks. In the second step, a set of potential correction plans for fixing each cause of deviation is computed by means of so-called \textit{generator functions}. Finally, in the third step, the potential correction plans computed in step two are composed into final correction plans that are then presented to the process agent.

The risk information attached to each deviation is used to sort the returned plans, from the one containing fewer risks to the one containing more risks. This assures that the algorithm will return the plans that take the enactment state closer to a less risky solution on the top of the list.

This section concludes the brief yet detailed presentation of our contributions to the problem of detecting and handling deviations but also concludes this chapter. Of course, due to space restriction, many aspects have not been presented here and the reader is kindly invited to refer to the published papers and to the thesis document in order to learn more about our approach.