POLITECNICO DI MILANO
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Inconsistencies and Deviations in Process Support Systems

Ph.D. Thesis of:
Gianpaolo Cugola

Advisor:
Prof. Carlo Ghezzi
Tutor:
Prof. Carlo Ghezzi
Supervisor of the Ph.D. Program:
Prof. Sergio Bittanti

X ciclo

To my brother

Acknowledgments

I wish to thank my advisor, Prof. Carlo Ghezzi, for his guidance, encouragement, and support throughout my graduate studies, and for giving me so many opportunities to pursue this and other scientific interests. He was always able to give me the right suggestion when things gave the impression of going in the wrong direction. I enjoyed working with him.

I have a great debt of gratitude with Elisabetta Di Nitto and Alfonso Fuggetta for their important contribution to the work presented in this thesis: thanks.

I wish also thanks Alberto Coen-Porisini, Gigi Lavazza, Dino Mandrioli, Angelo Morzenti, Mauro Pezzé, and Pierluigi San Pietro: I learned a lot from them.

I wish to thank (in strictly alphabetical order), Luciano Baresi, Francesco (Franz) Calzolari, Antonio Carzaniga, Fabiano Cattaneo, Alessandro (Alex) Orso, Sergio Silva Barradas, and Giovanni Vigna, my long-suffering officemate, together with Fabio (Cippino) Condrelli and Mario Motta, my flatmate. They made my time in Milano as a PhD student so enjoyable: "thanks friends".

Last but not least I wish to thank my family (yes, Laura, you are included) who never fail to love, support, and believe in me. The person I am mostly depends on them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Anyone who has never made a mistake
has never tried anything new."
Albert Einstein (1879-1955)

The term *business process* denotes the set of activities carried out by a business organization to pursue its mission. Typical examples of business processes are office procedures and engineering activities such as software development processes.

Business processes are characterized by a great complexity. They usually are composed of several long term activities which involve a large number of people that operate by taking advantage of computerized tools. This complexity has to be managed to guarantee the success of business processes.

To answer the increasing demand of methodologies and tools to manage the complexity of business processes, in the last years several researchers from different application domains started an intense research activity. The result of this research activity was the development of several tools which provide advanced features to support people involved in business processes. In particular, in the software engineering domain, several process-centered software engineering environments (PSEEs) were developed to support people involved in software development. Similarly, in the information systems and data-bases domains, several workflow management systems (WFMSs) were developed to automate and support information processes, i.e, processes concerned with the management of information flows.

In modeling and automating these processes both software engineers and people from information systems and databases areas move from a simple but fundamental observation: a high quality, well-standardized, and easy-to-manage business process is a key factor for the quality of the products and services delivered by a modern business organization. By moving from the same observation they developed two classes of systems that, despite of some differences, share a large number of similarities. In this thesis the unique term *"process support systems"* (PSSs) is used to refer to all these systems.

PSSs are based on an explicit description of the business process to be supported called *process model*. This model can be used for several purposes: (i) to consolidate process knowledge and increase process awareness among people involved in the process; (ii) to support activities of process assessment and improvement; (iii) to automate business activities and to guide people in performing their daily work, ensuring that the constraints expressed in the business process model under examination are not violated. All these tasks are important and a PSS should provide features to support all of them, but this thesis focuses on the last aspect, which is usually called *process model enactment*. The reason of this choice is simple: besides the fact that the ability of enacting a process model is universally recognized as crucial for advanced PSSs, process enactment is the activity in which currently available PSSs are weakest.

Probably the most relevant problem in the area of process enactment is the rigidity that PSSs exhibit in guiding people during enactment. Usually, the sequence of activities codified in the process model represents only the preferred and ideal course of action, but it cannot be considered the only way to pursue the business goals. In some cases people can undertake different choices and sequence of operations, still obtaining the expected results. Conversely, most PSSs tend to impose a course of actions that is strictly derived from the process model without tolerating deviations from this expected behavior.

This problem is directly related to the approach taken by most of the process description languages available. They adopt an operational approach. The business process is described as an algorithm that produces the output of the process from its input following a predefined sequence of steps. By using a language that adopts such an approach the process designers tend to over-specify the process. Conversely, a more descriptive approach helps process designers in identifying the fundamental goals and constraints that characterize the business process, ignoring less relevant constraints, and leaving the humans involved in the process free of choosing the sequence of operations they prefer.

Being free of deviating from the expected course of actions becomes even more important when a situation which was not foreseen at the time of process modeling arises. Not offering any feature to support people in facing these situations is probably the main drawback of modern PSSs in the area of process enactment. To be true, some of them offer their users the chance to modify the process model "on-the-fly" to take care of new conditions in the external environment. This, however, is rarely the best approach to cope with unforeseen situations and to increase users freedom, due to the effort required to modify the process model.

This thesis focuses on the problem of allowing deviations from the process model in PSSs. The problem is analyzed in detail and a solution is proposed, based on an innovative approach to process modeling and enactment that allows people to deviate from the course of actions stated in the process model without the need of preliminary changing the model itself.

It is important to observe that this thesis do not address the issues related to understanding "why" deviations occur. A comprehensive analysis of the motivations and causes of deviations from the process model demands for a detailed study of the cognitive and sociological factors that drive the human behavior and this is clearly out of the scope of this thesis. The goal of the thesis is instead to describe "how" deviations can be defined, observed, and tolerated independent of the causes and motivations that originate them. It provides concepts to understand and characterize the state of a business process and its process support system when deviations from the process model (for some reason) occur, and it describes PROSYT, a PSS which tolerates such deviations, controls that nothing serious could results from these actions, and supports people in reconciling the process actually followed with the process model when necessary.

The software engineering group at Politecnico di Milano has already made some preliminary work in the area of supporting people in deviating from a process model. In particular, two PSEEs have been designed and implemented that take two different approaches in facing this problem: SPADE [12, 10] and SENTINEL [26] (see Section 3.5 for a detailed description of the approaches adopted by these two PSEEs). This thesis gives a precise formalization of the concept of deviation from a process model and of the effects of these deviations on the overall consistency of the environment composed of the business process and its process support system (the results of this research were described in [25]). The thesis also presents PROSYT, a PSS which offers advanced features to tolerate deviations from the process model, supporting people in facing unforeseen situations and in reconciling the process actually followed with the process model when needed (the results of this research are described in the internal report [30]).

1.1 Organization of the thesis

The thesis is structured as follows:

- Chapter 2 gives an informal characterization of the problem of deviations from the process model in PSSs and of the inconsistencies that may result from these deviations between the state of the PSS and the state of the process actually followed. It also gives a simple example of business process and of some deviations from the expected course of actions that may be necessary to cope with unexpected situations. This example is used throughout the paper to better explain the concept introduced.

- Chapter 3 introduces a formal framework to model the concepts of deviation and inconsistency in PSSs. This activity of systematization of the concepts involved in process enactment has been fundamental to understand in detail the problem under examination and to suggest some possible solutions.

  To formalize this framework, the Z formal specification language [74] has been adopted. Note, however, that Z is only used as a linguistic vehicle to express and present the concepts introduced in this thesis. It is not intended to be used as a process description language to model deviations and inconsistencies.

- Chapter 4 describes PROSYT, a PSS capable of tolerating deviations from the process model. Its development took benefit of the insights given by the framework described above into the problem of tolerating deviations from the process model in PSS and of the experience gained by designing and developing SPADE and SENTINEL.

- Chapter 5 compares PROSYT with other PSSs that provide features to solve the problem of supporting people involved in business processes when something unforeseen happens.

- Chapter 6 surveys some recent research results related to the problem of managing inconsistencies. Several areas of computer science faced this problem from different perspectives and the most relevant results of these research efforts are reported. Some of them constituted the basis of our work.

- Finally, Chapter 7 draws some conclusions and indications for future research initiatives.

Chapter 2

Deviations and inconsistencies in PSSs

"Occurrences in this domain are beyond the reach of exact prediction because of the variety of factors in operation, not because of any lack of order in nature."
Albert Einstein (1879-1955)

2.1 Business processes and Process Support Systems

As mentioned in Chapter 1, the term *business process* is conventionally used to indicate the set of activities carried out by people involved in an organization to pursue the organization's business mission. Mature organizations are characterized by a high level of standardization of their business process, which is the result of years of practice in the particular business domain in which the organization operates. People involved in such organizations always operate in the same way because this way has proven to be the most effective.

The importance of an effective business process is a consequence of the fact that the quality of the products and services that an organization delivers to its clients is greatly influenced by the quality of its business process. An effective business process results in high quality products obtained with reduced effort in a cost effective way. This consideration motivated the development of a new class of applications called *Process Support Systems* (PSSs). PSSs aim at supporting an organization in improving and automating its business process. Usually, a PSS provides a *Process Description Language* (PDL) [42, 50, 75], used to develop a model of the business process. This model may be used to consolidate the process knowledge, to support process assessment and improvement, and to communicate the business rules within the organization.

Process modeling and improving are fundamental activities in mature organizations but they are not the only activities that can be supported by PSSs. Most modern business organizations operate by taking advantage of computerized tools. The activities that compose the business process of such organizations are carried out by people that use computerized tools as part of their daily work. Typical examples of such organizations are software factories, banks, and engineering design departments. The business processes of such organizations share common properties that can be summarized as follows:

- *People have a central role in performing the activities needed to accomplish the process goals*. They do so by interacting and cooperating among themselves and with computerized tools. For instance, in a software engineering process, programmers and designers cooperate according to defined rules and procedures to develop and deliver software products. They use several tools such as editors, compilers, and CASE tools to achieve these goals. Tools, however, cannot replace people, but just support their work. Even in software development, where the role of automation is essential (how could we ever compile a program manually?), people are still the key factor to ensure high quality results [55, 73].

- *Large amounts of information are persistently stored and managed by means of computers*. For instance, account data in a banking application or source code in a software engineering department are kept in a database or in the file system.

To support these organizations, a class of advanced PSSs exists, which can *enact* (i.e., execute) process models. During process model enactment, the PSS uses the rules and constraints expressed in the model to *automate* those activities that can be carried out without the intervention of human agents, and to *guide* and support people in carrying out those activities that require their intervention. This thesis focuses on several aspects related to process model enactment. As a consequence, in the remainder of the thesis, only PSSs featuring process model enactment are taken into consideration.

Typical examples of PSSs that offer the chance to enact a process model are *WorkFlow Management Systems* (WFMSs [50, 75]) and *Process-centered Software Engineering Environments* (PSEEs [42, 44, 2]). WFMSs have been developed to support information systems and, more generally, cooperative activities carried out in business processes. PSEEs aim at supporting the engineering process used to conceive, design, develop, and maintain a software product. These PSSs share many characteristics [9, 15]. Together with the standard tools to create process models, analyze them, and improve them according to the results of the analysis, they also provide the ability to enact process models in order to support people in carrying out the business process.

PSSs supporting such kind of process model enactment, in principle, should constitute one of the most useful tools for business organizations. It is therefore surprising to observe that, besides some significant exceptions, they are rarely adopted. This cannot be explained by saying that business organizations are not mature enough to adopt complex tools like PSSs. The costs involved when attempting to formalize a complex activity like a business process are usually high and the fact that the economic benefits of adopting a PSS to guide and automate a business process have never been demonstrated, partially explain why PSSs are rarely adopted. But this explanations do not entirely justify the difficulties encountered in trying to introduce PSSs in business organization. In our opinion a key motivation that restrained the adoption of this technology in business organization is that currently available PSSs are not flexible enough to support complex business processes. They tend to impose a course of actions that is strictly derived from the process model without tolerating deviations from this expected behavior.

Business processes are complex and highly dynamic systems, which involve a large number of people for a long time, and which often cannot be completely defined in advance. The major consequence of this characteristic of business processes is that PSSs have to be able to support people even when an unforeseen situation is encountered, which requires people to deviate from the process model.

2.2 Facing unforeseen situations in business processes

As mentioned before, people play a key role in business processes. The main consequence is that business processes are characterized by a greater dynamism than other, more repetitive, processes like production processes. While most of these processes can be completely automated, business processes cannot even be specified in advance and once for all in a precise way. This means that PSSs must be able to take into account unexpected situations. Unfortunately, currently available PSSs do not offer the appropriate mechanisms to cope with such situations. They offer several facilities to support and automate a business process when all proceeds as expected, but if an unforeseen situation is encountered, which requires a deviation from the specified process model to be managed, they often become an obstacle.

In order to give a more precise idea of the problems encountered with standard PSSs when unexpected situations arise, the next section describes an elementary example of business process and provides some examples of unexpected situations which may occur during the process. This example is also used throughout the thesis to explain in a more clear way the concepts presented.

2.2.1 Business processes and unforeseen situations: an example

The process example described in this section is taken from our daily work: it is the process of writing a technical paper like a research paper or a technical manual in a team using LaTeX [64]. It represents a typical business activity (several organizations are involved in similar activities as part of their business process), which involves a set of people, and is composed of several steps that require careful coordination. At the same time, the example is simple enough to be described in detail.

The process of writing a technical paper can be decomposed into three sub-activities.

- First the set of people involved in the process (i.e., the authors) must be chosen. One of them must play the role of *project leader*. The project leader is in charge of selecting the other authors, deciding the organization of the paper in sections, and deciding the responsibilities of the different authors (i.e., each author is *responsible* for one or more sections). The project leader is also responsible for editing the master document: the LaTeX file that includes all the other sections and contains the title of the paper and the names and affiliations of the authors.

- After choosing the project leader, composing the team, and structuring the paper, the editing phase starts. During this phase each author may decide who will be in charge of writing the sections he or she is responsible for (i.e., who is the *owner* of each section). Each owner edits the sections he or she owns until their status becomes acceptable.

- The last phase of the process is the revision phase. When all the authors have decided that the sections they own are ready to be delivered the project leader chooses one of them as a *reviewer* and starts the revision phase. During this phase the reviewer is allowed to edit all the sections (the reviewer is the owner of all of them) in order to find errors and correct them. When the reviewer is happy with the result, another revision phase may start (a different reviewer is chosen by the project leader) or the paper may be finally delivered in order to be submitted to a conference or a journal, or to be published.

During the process of writing a technical paper several unexpected situations may occur, which force the people involved in the process to deviate from the sequence of activities just described. As an example imagine that the project leader finds that the first version of the paper produced and delivered by the authors is satisfactory. He or she may decide to deliver the paper without performing the review phase.

As another example of a unexpected situation that may require a deviation from the expected process, imagine that the deadline for the submission of the paper approaches while the editing phase has not been completed yet. Imagine also that all the sections have been delivered with the exception of the final one (the "conclusions"). In such a situation the project leader may decide to start the review phase without waiting for the editing phase to complete in order to speed up the process by executing part of the edit and review activities in parallel. Also in this case a deviation from the process model is required in order to cope with an unforeseen situation (i.e., the delay in completing the editing of the conclusions).

2.2.2 Possible solutions to the problem of facing unforeseen situation in business processes

As previously mentioned, most of currently available PSS are unable to support unexpected situations: they adopt an approach to enactment based on a rigid enforcement of the process model being enacted. Users are not allowed to perform an operation unless it has been anticipated and described in the process model. Such an approach is often found to be too prescriptive and rigid. It imposes a pre-defined behavior and does not allow users to cope with unforeseen situations. When an unforeseen situation arises, users have to leave the PSS in order to perform the required actions. As a consequence, the actions needed to cope with the unforeseen situation are performed out of the PSS control. This may result in an inconsistency between the state of the process actually followed and the state of the process observed by the PSS. This is a consequence of the fact that the PSS is not aware of the results of the actions performed to cope with the unforeseen situation.

A possible solution to the problem of facing unforeseen situations, usually adopted in the WFMS domain [50, 75], is to manage what users need to accomplish, but is not specified in the workflow model (called *exception* in the WorkFlow domain) by exploiting exception handling techniques. Unfortunately, this solution is not as general as required. It allows users to cope only with a limited number of situations. Namely, the situations captured by one of the exception handlers provided as part of the model.

A different solution to the problem, adopted by some PSEEs, is to supply mechanisms to support on-the-fly modification of the process model [37, 77]. By using these mechanisms, the project manager may change the process model during enactment in order to introduce an explicit description of the unforeseen situation encountered and of the activities needed to cope with it. To offer this solution, PSSs have to incorporate facilities to describe and support the meta-process (i.e., the process of modifying an executing process) as part of the process itself. This can be obtained by introducing reflective features into PDLs [10, 61, 18].

Unfortunately, this approach is not always satisfactory. In fact, it is often too rigid; it does not provide a reasonable solution to deal with temporary, minor changes in the process. Modifying the process model is a time consuming activity that requires the project manager intervention. Moreover, it is not reasonable to modify the process model in order to cope with an unforeseen situation that is unlikely to occur again in the future and have a limited impact on the overall process. In practice, these situations are often managed by performing the necessary actions out of the PSS control. As already mentioned, the problem with this naive approach is that these actions may result in an inconsistency between the state of the process observed by the PSS and the actual state of the process. The consequence of this inconsistency is usually the impossibility for the PSS of providing further correct support to the process.

An alternative approach, exploited by the PROSYT environment, allows the enacted process to diverge from its model. To support this approach, a PSS has to offer mechanisms to *deviate* from the process model. The advantage of this approach is that it does not require the process model to be changed. Moreover, the deviation is performed under the PSS control that may track the deviating actions and may support the users in reconciling the enacted process and the process model, if necessary.

To better describe the differences between these approaches, the next section analyzes in more detail the problem of deviations in PSSs and the possible remedies.

2.3 Deviations and inconsistencies: an informal account

This section gives a more precise characterization of the concepts of *process model*, *actual process*, and *observed process*. This is useful to better analyze the problem of deviations in PSSs. The section also introduces the concept of process and environment level inconsistency and deviation that will be detailed and formalized in Chapter 3.

Process model. It is a model of the expected business process given in some PDL. It is enacted by the PSS to guide and support the users in carrying out the business process. It is a static entity.

Actual process. It is the process actually performed. At each time instant it may be described by the history of the activities that were performed to carry out the business process from the time it was started. It is a dynamic entity (i.e., it changes each time a new activity is performed). As shown in detail in Chapter 3, the actual process may be modeled statically as a state machine describing all the possible behaviors (i.e., histories) of the business process under examination.

Observed process. During process model enactment, a PSS has a partial view of the actual process. This is a consequence of the fact that the PSS is only aware of the operations that the users performed under its control. Other operations are invisible to the PSS. This partial view of the actual process owned by the PSS is called the “observed process”. At each time instant it may be described by a history of the activities that the users performed under the PSS control to carry out the business process from the time it was started. It is the result of the enactment. Like the actual process, it is a dynamic entity that may be modeled statically as a state machine which describes all the possible histories observable by the PSS.

When the actual process does not satisfy all the rules and constraints stated by the process model we say that a *domain-level inconsistency* has occurred. This name originates from the fact that domain-level inconsistencies are related to the “domain of interest” of the particular business process. As an example of domain-level inconsistency consider the process of writing a technical paper described in Section 2.2.1. If the project leader decides to deliver the paper without performing any review phase, the process enters a domain-level inconsistent state. The state of the process violates a constraint stated by the process model which prescribes that at least one review phase has to be completed before delivering the paper.

When the observed process does not represent a correct view of the actual process we say that an *environment-level inconsistency* has arisen. This name was chosen because environment-level inconsistencies refer to inconsistencies in the “environment” composed of the business process and the PSS which supports it. As an example of environment-level inconsistency, imagine that to deliver the paper the project leader has not invoked the ad-hoc operation provided by the PSS but he or she has simply printed the paper and submitted for publication. As a consequence, the PSS is not aware of the fact that the paper has been delivered and the process has completed. It will be waiting for the review phase to start.

A *domain-level deviation* is an action whose consequence is a domain-level inconsistency. It often occurs when people violate the constraints stated by the process model to cope with a situation that was not taken into account when the process model was developed. The action of delivering the paper without performing any review is an example of domain-level deviation.

Similarly, an *environment-level deviation* is an action whose consequence is an environment-level inconsistency. An environment-level deviation may occur when someone performs an action relevant for the business process out of the PSS control. When this happens, the state of the actual process and the state of the observed process diverge (i.e., an environment-level inconsistency arises). The action of delivering the paper invoking an operation out of the PSS control described above is a typical example of environment-level deviation.

In an ideal world, the actual process is always consistent with respect to the process model and the observed process always reflects the actual process. Deviations never occur. Unfortunately, as shown by the example given in Section 2.2.1, this is not always the case. People behave differently in different situations and the consequence of this fact is that it is hard to capture in advance in a formal model all the possible behavior of a complex dynamic system like a business process which involves several people. Unforeseen situations may always occur. As mentioned before, when this happens three different solutions are possible:

1. Before doing any action the process model is modified. A description of the actions that have to be performed in order to cope with the unexpected situation is added to the model and the PSS is given the new process model to enact.

The result of this approach is that the observed process continues to be a correct description of the actual process, which, in turn, remains consistent with respect to the process model. There is no need to deviate because the modified process model takes into account the "unforeseen situation".

2. The process model is not modified and the actions needed to cope with the unexpected situation are performed out of the PSS control. In this case an environment-level deviation occurs. The observed process is still consistent with the process model (the PSS is not aware of the deviation occurred) but it is not consistent with the actual process, which, in turn, is not consistent with the process model (i.e., a domain-level deviation has occurred, also). As a consequence, the PSS cannot analyze the deviation that occurred and cannot support the users in reconciling the actual process with the process model.

3. The process model is not modified but the PSS offers the chance to execute the actions necessary to cope with the unexpected situation under its control by explicitly deviating from the process model. Like in the previous case a domain-level deviation occurs, but in this case the observed process continues to be a correct description of the actual process (i.e., no environment-level deviation has occurred). In this case, the PSS is aware of the deviation occurred and can analyze it and support its users in reconciling the actual and observed processes with the process model. The PSS *tolerated* the deviation.

In theory, the safest and cleanest approach to cope with unforeseen situations is the first but, as mentioned in Chapter 1, it is often impracticable due to the effort required to modify the process model. In all the situations in which such an approach cannot be adopted, the last should be chosen. Unfortunately, the currently available PSSs, with the exception of the SENTINEL prototype [26] do not support this approach. This thesis shows how an advanced PSS may offer this solution and the advantages it offers in terms of increased flexibility and usability of the PSS.

Chapter 3

The formal framework

"It is theory that decides what can be observed."
Albert Einstein (1879-1955)

3.1 Introduction

This chapter aims at providing a formal framework to describe deviations and inconsistencies in a more precise way. More generally, this chapter formalizes the relationship between a business process and its process support system. A model of the environment composed of these two systems is defined and the behavior of such composite environment is studied by analyzing the way its subcomponents influence each other. To formalize this framework, the Z formal specification language [74] was adopted.

The roots of this work are in the research activity carried out by the author in the SPADE and SENTINEL projects. In particular, I started working on the issue of tolerating and controlling inconsistencies in the SENTINEL project [26]. In parallel, the work carried out within the SPADE project to support dynamic process model changes [10] originated a first informal discussion of some of the issues presented in this chapter [8].

Before describing the framework to formalize deviations and inconsistencies it is important to underline the fact that the ultimate goal of the chapter is not to introduce a new approach to process modeling. Rather, it aims at providing a general framework in which the concepts of deviation, inconsistency, and reconciliation are defined and formalized. By using this framework, different existing PSSs systems can be analyzed according to their ability to deal with these issues.

Formal models are used in engineering to reason about complex problems [62, 78]. The main difficulty found in the definition of a formal model is the identification of the entities to be modeled. This choice is influenced by the characteristics and properties to be studied. In this chapter the symbols $\mathcal{BP}$ and $\mathcal{PSS}$ are used to indicate a business process and its process support system, respectively. $\mathcal{BP}$ and $\mathcal{PSS}$ are modeled by two state machines, BP, and PSS, respectively, possibly with infinite states.

$\mathcal{BP}$ and $\mathcal{PSS}$ constitute an *environment*, denoted by $\mathcal{E}$, whose behavior depends on the way $\mathcal{BP}$ and $\mathcal{PSS}$ interact. The model of $\mathcal{E}$, E, is composed of BP and PSS, and two relations R_s and R_t. R_s relates the states of PSS and BP. In particular, a pair

(s_{pss}, s_{bp}) is in R_s if s_{pss} is a state of PSS, s_{bp} is a state of BP, and s_{pss} "correctly describes" s_{bp}. Similarly, R_t relates state transitions in PSS and BP.

If states and transitions in BP have a corresponding element in PSS according to the above relations, then $\mathcal{BP}$ and $\mathcal{PSS}$ may potentially progress in such a way that the behavior of the business process is properly and consistently reflected in the process support system. However, even if the relations cover all of BP's states and transitions (in this case we say that we have state and transition *completeness*), this does not ensure that the two models always evolve through related and consistent states. State and transition completeness are static properties. They do not ensure that each state transition in BP is always mirrored, at execution time, by a corresponding transition in PSS that properly describes it. State and transition completeness are necessary conditions, but do not guarantee that BP and PSS (and consequently $\mathcal{BP}$ and $\mathcal{PSS}$) progress in a consistent way. If, during execution, states and transitions in BP are properly mirrored in PSS according to relations R_s and R_t, we say that the evolution of the two models is *coherent*.

It is possible to distinguish between the concepts of *deviation* and *inconsistency*, even though they are related. Moreover, it is possible to distinguish between deviations and inconsistencies occurring within $\mathcal{BP}$ and those due to a mismatch between $\mathcal{BP}$ and $\mathcal{PSS}$. Deviations occurring within the business process $\mathcal{BP}$ are called *domain-level deviations* (DDs). A domain-level deviation is an event that deviates from the expected behavior of $\mathcal{BP}$. This behavior is statically captured by the business process model which describes all the constraints that the business process has to fulfill. For instance, a DD occurs if a developer involved in a software process writes the source code of a module before writing its design, assuming that a process rule prescribes that source code creation of each module "must" be preceded by the definition of its design. Domain-level deviations might bring the business process in a *domain-level inconsistent* state. For instance, in the example above, the deviation brings the software process in an inconsistent state, in which a source code file does not have a corresponding design file. Such deviations and inconsistencies may occur even when no process support system is used. Indeed, a process support system may (and should) describe even domain-level inconsistent states and deviations, in order to guarantee support to exceptional (but even possible) situations that should be tolerated.

Deviations occurring between the business process $\mathcal{BP}$ and the process support system $\mathcal{PSS}$ are called *environment-level deviations* (EDs). An environment-level deviation occurs when the event performed in $\mathcal{BP}$ is not mirrored by an event in $\mathcal{PSS}$ that correctly describes it. This means that the behavior of the business process is not properly reflected in the process support system. As a result of an environment-level deviation, the state of E is *environment-level inconsistent*, that is, the states of BP and PSS do not belong to R_s. For instance, a design document exists, but the process support system is not aware of it.

Before concluding this introductive section, it is important to observe that the concepts presented in this chapter can be applied in a more general context to describe the relationship between a computerized system and (parts of) the real world. For instance, similar problems and issues are encountered when describing a digital control system and the physical device it is controlling. This chapter focuses on the application of the framework to business processes since this is the domain where the author has more experience. A generalization of the approach presented in this chapter may be found in [25].

3.2 The model of $\mathcal{E}$

A *business process* ($\mathcal{BP}$) and a *process support system* ($\mathcal{PSS}$) constitute an *environment*, called $\mathcal{E}$. $\mathcal{BP}$ and $\mathcal{PSS}$ may be modeled as two state machines, BP and PSS, respectively. In the model of $\mathcal{E}$ our attention focuses on the states of $\mathcal{BP}$ and $\mathcal{PSS}$, and on the relationship between these states, described by two relations, R_s and R_t.

3.2.1 BP

The model BP of a business process $\mathcal{BP}$ is defined as a Z schema that uses the following sets:

$$[BPSTATE, BPTRANSITION]$$

$BPSTATE$ is the domain of the states of any model BP. Given any business process $\mathcal{BP}$, at any time, the states of its model BP are elements of $BPSTATE$. The definition of the state structure is not provided, since it strictly depends on the particular business process being considered. Moreover, it is not relevant for the rest of the presentation. In our formalization, indeed, a state is viewed as an elementary entity. It may characterize a variety of information, such as artifacts, roles, deadlines, and any other information that is relevant to the business process under consideration.

$BPTRANSITION$ is the domain of the state transitions of any model BP. Function *BPTrans* relates each state transition to its source and target state, respectively:

$$BPTrans : BPTRANSITION \rightarrow (BPSTATE \times BPSTATE)$$

Based on these definitions, the model BP for a specific business process $\mathcal{BP}$ is described by the following schema:

$$
\begin{array}{l}
\textbf{BP} \\
\hline
S_{bp} : \mathbb{P}\, BPSTATE \\
T_{bp} : \mathbb{P}\, BPTRANSITION \\
s^i_{bp} : BPSTATE \\
\hline
\forall\, t : BPTRANSITION \mid t \in T_{bp} \bullet BPTrans\ t \in (S_{bp} \times S_{bp}) \\
s^i_{bp} \in S_{bp} \\
\hline
\end{array}
$$

where:

- S_{bp} is the set of BP states.
- T_{bp} is the set of BP state transitions.
- s^i_{bp} is initial state of BP.

The schema predicate part states the following propositions:

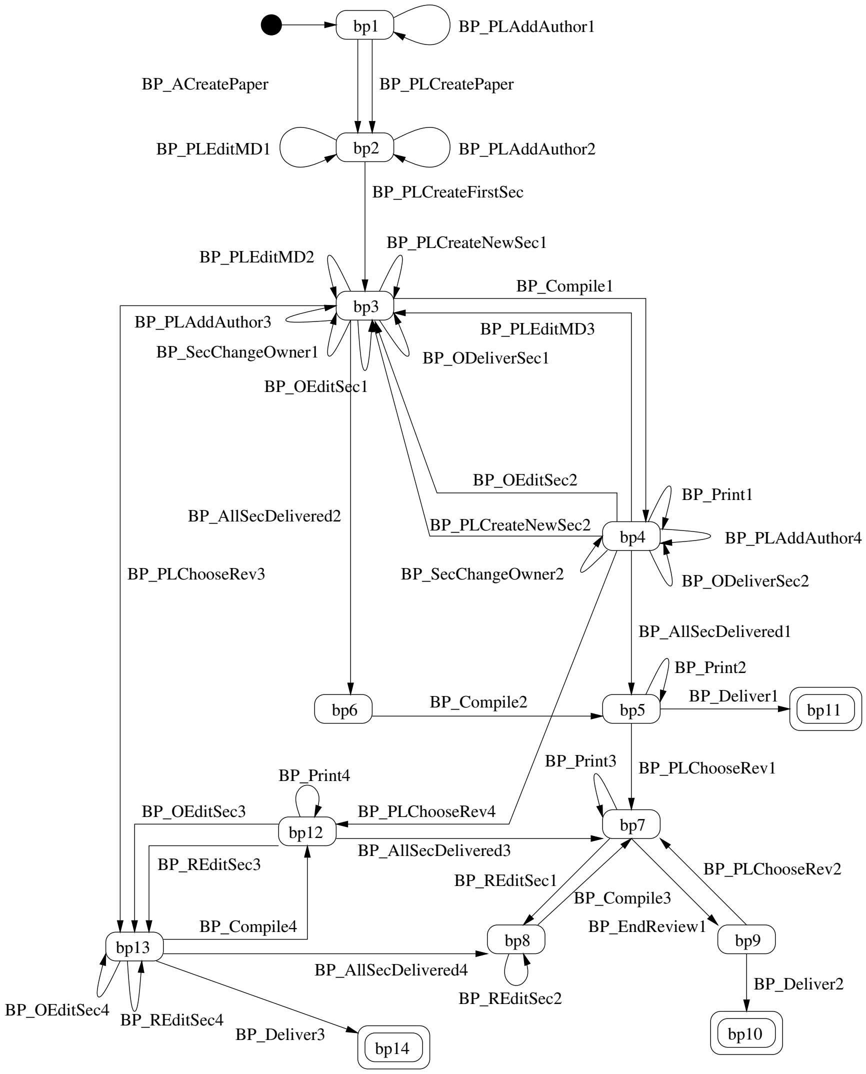

Figure 3.1: The model BP for the example process.

1. Every transition has a source and a target state through function $BPTrans$.

2. The initial state of BP belongs to S_{bp}.

Example 1 *Let us consider the business process $\mathcal{BP}$ described in Section 2.2.1. A model BP of this system is the state machine shown in Figure 3.1. This state machine is assumed to correctly describe all the possible behaviors of $\mathcal{BP}$.*

In Figure 3.1, `bp1` *is the initial state of BP. It represents the fact that the activity of writing the paper has not been started yet. In this state the project leader may add new authors to the project (transition* `BP_PLAddAuthor1`*) in order to compose the team that is in charge of carrying out the process. The project leader action of creating the initial version of the master document is modeled by transition* `BP_PLCreatePaper`*. Transition* `BP_ACreatePaper` *describes the action of creating the initial version of the master document done by an user different from the project leader (i.e., an author). When the master document has been created (state* `bp2`*) the project leader may edit it (transition* `BP_PLEditMD1`*) or may add new authors (transition* `BP_PLAddAuthor2`*). State* `bp3` *describes the fact that at least one section has been created and it has been assigned to an author. From this state new sections may be added (transition* `BP_PLCreateNewSec1`*), the project leader may edit the master document (transition* `BP_PLEditMD2`*), the owner of a section may be changed (transition* `BP_SecChangeOwner1`*), sections may be edited or delivered by their owners (transitions* `BP_OEditSec1` *and* `BP_ODeliverSec1`*, respectively), new authors may be added (transition* `BP_PLAddAuthor3`*), and the document may be compiled (transition* `BP_Compile1`*).*

When the document has been compiled, the business process reaches state `bp4`*. In this state sections may be delivered (transition* `BP_ODeliverSec2`*), new authors may be added to the project (transition* `BP_PLAddAuthor4`*), the document may be printed (transition* `BP_Print1`*), and the owner of a section may be changed (transition (transition* `BP_SecChangeOwner2`*). From state* `bp4` *the business process moves back to state* `bp3` *if a new section is created or if one of the sections (or the master document) is edited.*

When all sections have been delivered (transitions `BP_AllSecDelivered1` *or* `BP_AllSecDelivered2`*) and the document has been compiled (transitions* `BP_Compile1` *or* `BP_Compile2`*), the business process reaches state* `bp5`*. From this state the project leader may decide to deliver the paper (transition* `BP_Deliver1`*). Otherwise, he or she may start the review phase by choosing a reviewer (transition* `BP_PLChooseReviewer1`*). From state* `bp7` *(i.e., reviewer chosen and document compiled) the reviewer may edit the different paper sections (transition* `BP_REditSec1`*) and print the paper (transition* `BP_Print2`*). When the review phase ends (transition* `BP_EndReview1`*) the business process enters state* `bp9`*. From this state the project leader may start a new review phase (transition* `BP_PLChooseReviewer2`*) or may deliver the paper (transition* `BP_Deliver2`*) thus concluding the process.*

Observe that the review phase may be started from states `bp3` *and* `bp4` *also (i.e., when some sections have not been delivered yet). If this happens the business process reaches state* `bp12`*. From this state the process continues as if all the sections had been delivered. In principle the paper may be delivered even if some sections have* *not been delivered. Otherwise, when all the sections have been delivered (transitions* `BP_AllSecDelivered3` *or* `BP_AllSecDelivered4`*) the process reaches one of the state described before and may continue regularly.*

Notice that in this example several (but not all) pairs of states are linked by a state transition. Some behaviors, in fact, never occur. For instance, it never happens that an author different from the owner edits a section.

Notice also that some of the steps modeled in BP are not compliant with the "official" steps described in Section 2.2.1. For instance, the review phase may start even if some sections have not been delivered. Similarly, the paper may be delivered without performing any review session (transition `BP_Deliver1`*).*

The steps of Section 2.2.1 represent the "expected behavior" of the business process, but there are also other situations that can occur even if they have not been anticipated in the official definition of the process. □

3.2.2 PSS

The Z schema defined for the model PSS of a process support system $\mathcal{PSS}$ is analogous to BP's schema defined in Section 3.2.1:

$$[PSSSTATE, PSSTRANSITION]$$

$$PSSTrans : PSSTRANSITION \rightarrow (PSSSTATE \times PSSSTATE)$$

PSS

$$S_{pss} : \mathbb{P}\, PSSSTATE$$
$$T_{pss} : \mathbb{P}\, PSSTRANSITION$$
$$s^i_{pss} : PSSSTATE$$

$$\forall\, t : PSSTRANSITION \mid t \in T_{pss} \bullet PSSTrans\ t \in (S_{pss} \times S_{pss})$$
$$s^i_{pss} \in S_{pss}$$

Notice that PSS is not a "process model". Rather, it is a description of the "virtual machine" $\mathcal{PSS}$ which is supporting the business process $\mathcal{BP}$. Usually, this virtual machine is obtained by enacting a process model [8]. Drawing an analogy with the traditional world of programming, the process model can be viewed as the program, $\mathcal{PSS}$ is the virtual machine executing the program, and PSS is a description of this virtual machine.

BP and PSS are two distinct machines that model different entities of the real word, $\mathcal{BP}$ and $\mathcal{PSS}$. BP describes what is really happening in the business process, that is, it is a "perfect (or ideal) view" of $\mathcal{BP}$. PSS is the description of the virtual machine enacting a concrete process model which, in practice, can only represent a limited subset of all possible behaviors of $\mathcal{BP}$. Therefore, PSS is a "perfect (or ideal) view" of $\mathcal{PSS}$, where $\mathcal{PSS}$ represents $\mathcal{BP}$ only in a partial way. BP and PSS are usually related. In particular, they interact during their progress in order to allow the state of PSS to mirror the state BP assumes at the same time. Intuitively, the "closer" PSS mirrors BP, the "better" the process support system supports the business process.

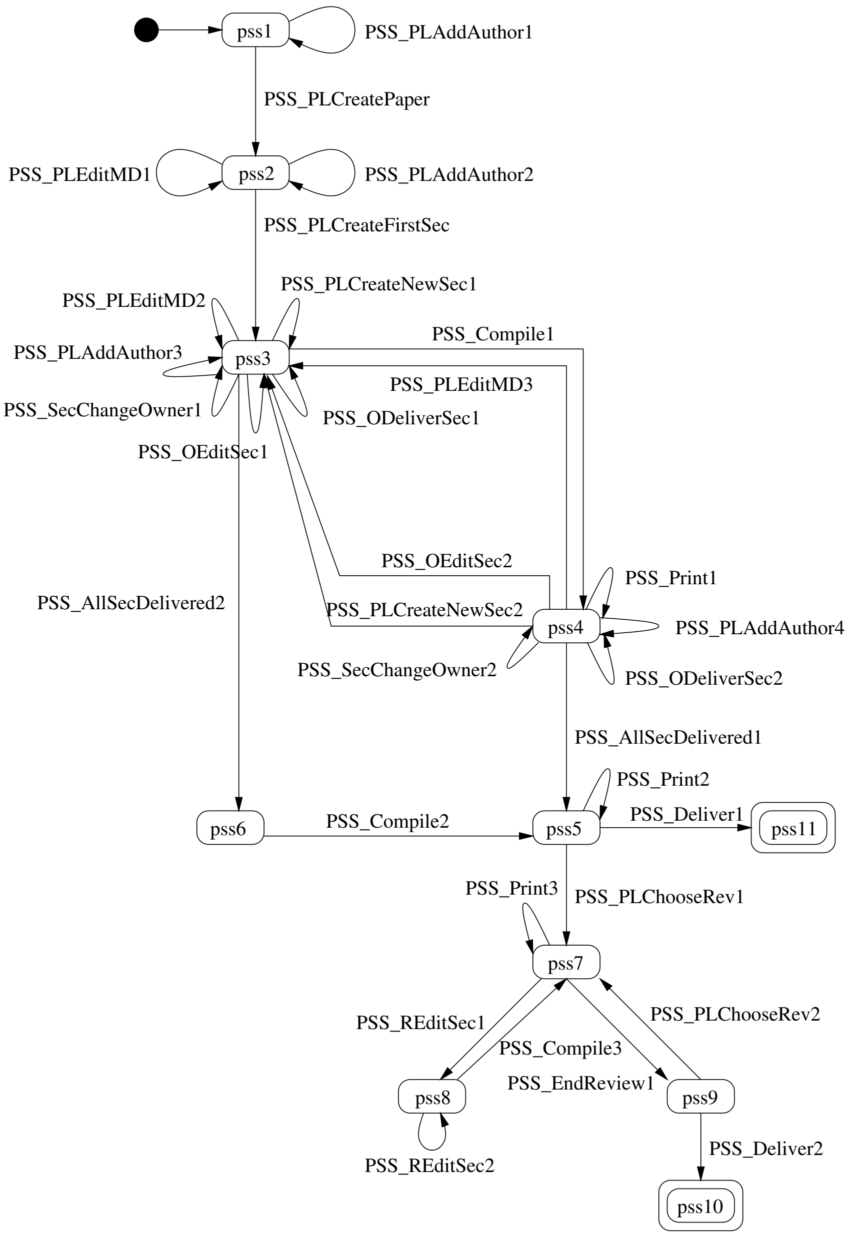

Figure 3.2: The model PSS for the example process.

Example 2 *In the previous section, a model BP for the example presented in Section 2.2.1 was defined. Figure 3.2 shows a model PSS for the corresponding process support system. In Section 3.2.3 the relationships between BP and PSS is formalized, and this example is used to illustrate them in detail. Here we can intuitively say that each state and transition in Figure 3.2 corresponds to some state and transition in Figure 3.1. For instance, transition* `PSS_PLCreatePaper` *describes, in the process support system model, the project leader action of creating the initial version of the master document (i.e., transition* `BP_PLCreatePaper`. □

3.2.3 E

E is the model of the environment $\mathcal{E}$. E is obtained by composing BP and PSS. The relationships between BP and PSS are defined by two relations R_s and R_t. A PSS state and a BP state belong to relation R_s if the former "correctly describes" the latter. The same applies to relation R_t on transitions. Clearly, the meaning of "correctly describes" depends on the characteristics and properties of the environment being observed. It is not possible to define *a-priori* how a process support system relates to the corresponding business process. These relationships depend on the application domain being studied, and on the characteristics of the modeling language and process support system being used. They define the semantics of a process support system with respect to the business process it is supporting.

An E state is defined as a pair of PSS and BP states:

$$ESTATE == PSSSTATE \times BPSTATE$$

Similarly an E transition (called *event*) is a pair of PSS and BP transitions:

$$ETRANSITION == PSSTRANSITION \times BPTRANSITION$$

Events are pairs of transitions of PSS and BP that occur at the same time.[1] Identity transitions may be used to represent events that leave one of the two systems, PSS or BP, in the same state.

Function $ETrans$ associates each event with its initial and final states:

$$\begin{array}{|l}
ETrans : ETRANSITION \to (ESTATE \times ESTATE) \\
\hline
\forall\, t_e : ETRANSITION \bullet ETrans\ t_e = \\
\qquad ((first \circ PSSTrans \circ first)\ t_e, (first \circ BPTrans \circ second)\ t_e) \mapsto \\
\qquad ((second \circ PSSTrans \circ first)\ t_e, (second \circ BPTrans \circ second)\ t_e)
\end{array}$$

The axiom on $ETrans$ relates $ETrans$ to $PSSTrans$ and $BPTrans$. It basically says that the components of the pair of E states resulting from the application of $ETrans$ to an event t_e are themselves a pair of PSS and BP states obtained applying $PSSTrans$ and $BPTrans$ to the transitions composing t_e.

first and *second* are Z built-in operators that return the first and second component of a pair, respectively.

The following Z schema describes environment E:

[1] We do not make any assumption about time granularity: in general, it depends on the specific application domain.

$$
\begin{array}{l}
\textbf{Schema } E \\
\hline
BP \\
PSS \\
T_{ef} : \mathbb{P}(ETRANSITION) \\
s_e^i : ESTATE \\
R_s : PSSSTATE \leftrightarrow BPSTATE \\
R_t : PSSTRANSITION \leftrightarrow BPTRANSITION \\
\hline
T_{ef} \subseteq (T_{pss} \times T_{bp}) \\
s_e^i = (s_{pss}^i, s_{bp}^i) \\
s_e^i \in R_s \\
\mathrm{dom}\, R_s \subseteq S_{pss} \\
\mathrm{ran}\, R_s \subseteq S_{bp} \\
R_t \subseteq T_{ef} \\
\forall\, t_{pss} : PSSTRANSITION;\ t_{bp} : BPTRANSITION \bullet (t_{pss}, t_{bp}) \in R_t \Leftrightarrow \\
\quad first(PSSTrans\ t_{pss}) \mapsto first(BPTrans\ t_{bp}) \in R_s \wedge \\
\quad second(PSSTrans\ t_{pss}) \mapsto second(BPTrans\ t_{bp}) \in R_s \wedge \\
\quad (t_{pss}, t_{bp}) \in T_{ef}
\end{array}
$$

where:

- BP and PSS are the schemas defined before.
- T_{ef} is the set of feasible events of E. It is a subset of $T_{pss} \times T_{bp}$.
- s_e^i is the initial state of E. It is defined by the pair of PSS and BP initial states.
- R_s and R_t are the relations defined before. R_t is a subset of T_{ef}.

In general, not all the possible pairs of transitions in PSS and BP are *feasible* for E. Some of them are not allowed by the intrinsic behavior of $\mathcal{E}$, that is, by the way $\mathcal{BP}$ and $\mathcal{PSS}$ interact.

For instance, consider the business process presented in Section 2.2.1. Assume that the only way to accomplish the creation of the first version of the master document is to invoke an operation provided by the process support system. Suppose also that this operation can be invoked by the project leader only. In this case, the occurrence of transition `BP_PLCreatePaper` (see Example 1), which models in BP the creation of the master document by the project leader, always causes the occurrence of transition `PSS_PLCreatePaper` (see Example 2) that models in PSS the same operation. Suppose, however, that in $\mathcal{BP}$ an user different from the project leader can create the first version of the master document using some other mechanism which is not directly controlled by the process support system (for instance, by directly invoking the emacs editor). This behavior is represented in Fugure 3.1 by the transition `BP_ACreatePaper` from `bp1` to `bp2`. If this operation is not detected and controlled by the process support system, its execution does not cause transition `PSS_PLCreatePaper` in PSS to occur. That is, in the first case the only feasible event is (`PSS_PLCreatePaper`, `BP_PLCreatePaper`). In the second case, event (I_{pss1}, `BP_ACreatePaper`) would also be feasible, where I_{pss1} is the identity transition that keeps PSS in the initial state `pss1`. In such a case, the first version of the master document is created, but the process support system has not detected this event.

In conclusion, the set of feasible events T_{ef} is, in general, a proper subset of $T_{pss} \times T_{bp}$. Moreover, only a subset of E feasible events belongs to relation R_t. Such events are called *correct events*.

It is reasonable to assume that whenever an event belongs to R_t (i.e., it is a correct event), then its initial and final states are related by relation R_s, and vice-versa. This is expressed by the last formula in the E schema, which formalizes the following axiom:

Axiom 1 (Axiom on R_t) *Any event of E belongs to the relation R_t iff the following propositions hold:*

1. *the event belongs to T_{ef},*
2. *the corresponding initial state belongs to R_s,*
3. *the corresponding final state belongs to R_s.*

□

This axiom allows us to assert that, if E progresses through correct events, its state always belongs to R_s and vice versa.

Example 3 *Referring to Figures 3.1 and 3.2, relation R_s is defined by the pairs:*
R_s = {`(pss1, bp1), (pss2, bp2), .... , (pss11, bp11)`}
BP states `bp12`, `bp13`, *and* `bp14` *do not have any corresponding PSS state through relation R_s. This means that the process support system is not able of describing these states. As a result, whenever the business process reaches one of these states it will be "inconsistent" with respect to the actual PSS state. Notice that an inconsistency between BP and PSS may also occur when, given the current states of BP and PSS, s_{bp} and s_{pss}, s_{bp} is actually described by some other PSS state $\overline{s_{pss}}$, such that $\overline{s_{pss}} \neq s_{pss}$.*

For the sake of simplicity, similar names have been used to represent every pair of BP and PSS transitions belonging to R_t (e.g., `(PSS_Compile1,BP_Compile1)`*). From Figures 3.1 and 3.2 it results that several BP transitions do not have corresponding transitions in PSS (e.g.,* `BP_Deliver1` *and* `BP_PLChooseRev3`*).*

T_{ef} contains all the events belonging to R_t and the pairs:

(I_{pss1},`BP_ACreatePaper`)

(`PSS_PLCreatePaper`, I_{bp2})

(I_{pss3},`BP_T1`)

(I_{pss4},`BP_T2`)

where

- `BP_T1` *is any transition that belongs to the paths that start with transition* `BP_PLChooseRev3` *and ends in one of the process' final states;*
- `BP_T2` *is any transition that belongs to the paths that start with transition* `BP_PLChooseRev4` *and ends in one of the process' final states;*

- *I_{bp2} denotes the identity transition starting from state $bp2$;*
- *I_{pss1}, I_{pss3}, and I_{pss4} denote identity transitions starting from states $pss1$, $pss3$, and $pss4$, respectively.*

As a consequence of the definition of T_{ef}, $\mathcal{PSS}$ is unable to detect that the project leader decide to start the review phase even if some of the sections have not been delivered yet. If this happens, the $\mathcal{PSS}$ cannot support its users anymore. They have to continue the process without its help. □

3.2.4 Event sequences and histories

E progresses through *event sequences*, i.e., ordered sets of events. A *history* is an event sequence starting from the initial state s_e^i.

Not all of the E states may be *reachable*. A state s_e is reachable by the environment E if a history exists having s_e as final state. These definitions are formalized by enriching the E definition schema:

$E\_SequencesAndHistories$

E
$S_{t_e} : \mathbb{P}(\mathrm{seq}_1\ ETRANSITION)$
$PH_e : \mathbb{P}(\mathrm{seq}_1\ ETRANSITION)$
$S_{er} : \mathbb{P}\ ESTATE$
$S_{pssr} : \mathbb{P}\ PSSSTATE$
$S_{bpr} : \mathbb{P}\ BPSTATE$
$T_{unr} : \mathbb{P}\ ETRANSITION$

$$S_{t_e} = \{s_{t_e} : \mathrm{seq}_1\ ETRANSITION \mid (\forall\, i : \mathrm{dom}\, s_{t_e} \bullet s_{t_e}\ i \in T_{ef}) \wedge (\forall\, i : 2 \mathinner{..} \# s_{t_e} \bullet (second \circ ETrans \circ s_{t_e})\ (i-1) = (first \circ ETrans \circ s_{t_e})\ i)\}$$

$$PH_e = \{s_{t_e} : \mathrm{seq}_1\ ETRANSITION \mid s_{t_e} \in S_{t_e} \wedge (first \circ ETrans \circ s_{t_e})\ 1 = s_e^i\}$$

$$S_{er} = \{s_e : ESTATE \mid s_e = s_e^i \vee (\exists\, h : \mathrm{seq}_1\ ETRANSITION \mid h \in PH_e \bullet (second \circ ETrans \circ h)\ \# h = s_e)\}$$

$$S_{pssr} = \{s_{pss} : PSSSTATE \mid (\exists\, s_{bp} : BPSTATE \bullet (s_{pss}, s_{bp}) \in S_{er})\}$$

$$S_{bpr} = \{s_{bp} : BPSTATE \mid (\exists\, s_{pss} : PSSSTATE \bullet (s_{pss}, s_{bp}) \in S_{er})\}$$

$$T_{unr} = \{t : ETRANSITION \mid t \in T_{ef} \wedge (\forall\, s_{t_e} : PH_e \bullet \neg\, (\exists\, i : \mathrm{dom}\, s_{t_e} \bullet s_{t_e}\ i = t))\}$$

where:

- S_{t_e} is the set of all the possible event sequences for E.[2]
- PH_e is the set of all the possible E histories, that is, all the possible event sequences having s_e^i as initial state.

[2]Notice that seq_1 is a Z operator used to define sequences. It is defined as a function from integers to E events. When it is applied to a number i, it returns the $i - th$ event of the sequence.

- S_{er} is the set of E reachable states.

- Sets S_{bpr} and S_{pssr} are defined as the projection of S_{er} on S_{bp} and S_{pss}, respectively.

- T_{unr} contains all feasible events that do not belong to any history.

3.3 Deviations and inconsistencies

Using the concepts introduced in Section 3.2, it is now possible to provide formal definitions for the terms *inconsistency* and *deviation*.

Inconsistency is a concept related to states. For instance, in a software development process, the state of BP is inconsistent when the implementation and the interface of a module do not match. As another example, the state of E is inconsistent when the development of a module has been completed but the process support system is still waiting for the delivery of the module. Intuitively, this means that the process support system has an image of the business process that does not describe its actual state correctly.

Deviation is a concept related to transitions. When BP or E enters an inconsistent state, this means that a deviating transition has occurred.

3.3.1 Domain-Level Inconsistencies and Deviations

In Section 2.3 inconsistencies and deviations occurring in the business process have been called *domain-level inconsistencies* (DIs) and *domain-level deviations* (DDs), respectively. They can occur even when no process support system is used.

As a practical example of DI, suppose one issues a check but forgets to register this operation on the notebook of bank account operations. Thus, a domain-level deviation occurs. The notebook and the bank account define a domain-level inconsistent state. The "real world" is "internally" inconsistent. As mentioned in Section 2.3, whether a state is domain-level inconsistent depends on the business process under consideration.

Formally, the concept of domain-level inconsistency can be modeled by partitioning the state set S_{bp} in two subsets, S_{bp_i} and S_{bp_c}, describing the set of domain-level inconsistent and consistent states of BP, respectively. Similarly, the concept of domain-level deviation is modeled by partitioning set T_{bp} in two subsets, T_{bp_d} and T_{bp_g}. T_{bp_d} represents the set of domain-level deviations, T_{bp_g} represents the set of the expected behaviors (i.e., the "good" transitions).

These concepts may be formalized by enriching the original BP schema:

$$
\begin{array}{|l}
\hline
BP_1 \\
BP \\
S_{bp_c}, S_{bp_i} : \mathbb{P}\, BPSTATE \\
T_{bp_g}, T_{bp_d} : \mathbb{P}\, BPTRANSITION \\
\hline
s^i_{bp} \in S_{bp_c} \\
\langle S_{bp_c}, S_{bp_i} \rangle \text{ partition } S_{bp} \\
\langle T_{bp_g}, T_{bp_d} \rangle \text{ partition } T_{bp} \\
\forall\, t_{bp} : T_{bp} \bullet t_{bp} \in T_{bp_g} \Rightarrow (\mathit{first}\ t_{bp}) \in S_{bp_c} \wedge (\mathit{second}\ t_{bp}) \in S_{bp_c} \\
\hline
\end{array}
$$

From the schema it follows that:

- The initial state of BP, s^i_{bp}, is supposed to be domain-level consistent.

- Deviations and inconsistencies are related concepts. In particular, the last axiom of schema BP_1 asserts that a necessary condition for a transition to be "good" (i.e., not to be a deviation) is that it starts and ends in a consistent state. Notice that this is just a necessary condition. In fact, transitions between two consistent states may still represent an undesired action (i.e., a deviation from the desired process).

In the Example 1, states `bp11`, `bp12`, `bp13`, and `bp14` belong to S_{bp_i}. This is intuitively justified by observing that they does not satisfy some of the constraints stated by process model described in Section 2.2.1. In particular, state `bp11` describes the fact that the paper has been delivered without performing any review. Similarly, states `bp12`, `bp13`, and `bp14` belong to a review phase started when some of the sections where not delivered yet.

From the axiom reported in schema BP_1 it follows that all transitions starting or ending in an inconsistent state (e.g., `BP_PLChooseRev3`, `BP_OEditSec4`, ...) are domain-level deviations.

As a final remark, it can be observed that DIs have been simply defined as a subset of S_{bp}. A richer model might go further by classifying inconsistencies according to their severity level. This can be formalized by partitioning S_{bp_i} into equivalence classes defined by relation Sev: given two states bp_1 and bp_2, $bp_1\ Sev\ bp_2$, if both have the same severity level. It is easy to show that Sev can be defined as an equivalence relation. Again, Sev should be defined according to the requirements and properties of the application domain.

$\mathcal{PSS}$ does not necessarily describe only domain-level consistent states and "good" transitions, but it can also describe domain-level inconsistent states and deviations. This is true, for instance, in Example 3, where the domain-level inconsistent state `bp11` is properly described in PSS by the state `pss11` while the domain-level deviation `BP_Deliver1` is modeled in PSS by the transition `PSS_Deliver1`. Therefore, in general, the process support system $\mathcal{PSS}$ may be able to control the progress of the business process $\mathcal{BP}$ even when it deviates and reaches a domain-level inconsistent state.

3.3.2 Environment-Level Inconsistencies and Deviations

Environment-level Inconsistencies (EIs) are inconsistencies between $\mathcal{BP}$ and $\mathcal{PSS}$. When the environment $\mathcal{E}$ reaches an environment-level inconsistent state, the state of the process support system does not mirror the state of the business process.

An *incorrect event* is a feasible event of E, $t_e = (t_{pss}, t_{bp})$, such that the transition t_{pss} performed by the process support system does not correctly describe the corresponding transition t_{bp} performed by the business process. In other words, an incorrect event is a feasible event that does not belong to R_t.

From the definition above, feasible events set is partitioned in two subsets: correct and incorrect events. The incorrect events whose initial state belongs to R_s are called *Environment-level Deviations* (EDs).

Recalling the previous example and the definition of ED, one may notice that the E feasible events $(I_{pss3}$,`BP_PLChooseRev3`$)$ and $(I_{pss4}$,`BP_PLChooseRev4`$)$ are typical examples of environment-level deviations.

Domain-level and environment-level deviations are closely related concepts. If the process support system $\mathcal{PSS}$ does not describe a DD, the occurrence of this deviation results in an ED. For instance, in Example 3, the occurrence of DD `BP_PLChoose-Rev3`, results in ED $(I_{pss3}$,`BP_PLChooseRev3`$)$. On the other hand, an ED may occur even if a DD does not occur. This happens if the process support system does not completely describe the expected behavior of $\mathcal{BP}$. Enhancing *completeness*, that is, increasing the number of $\mathcal{BP}$ transitions (both DDs and "good" transitions) described by $\mathcal{PSS}$, reduces the chance of an ED to occur (see Section 3.4).

When an ED occurs, BP and PSS have to be reconciled in some way. A *reconciling sequence* is a sequence of E feasible events starting from an environment-level inconsistent state and terminating in a environment-level consistent state.

Schema $E\_DeviationsAndRecSeqs$ formalizes the concepts of incorrect event, environment-level deviation, and reconciling sequence. In particular, set I_e is the set of all incorrect events, set ED is the set of environment-level deviations, and set RS is the set of all reconciling sequences:

$$
\begin{array}{|l}
\hline
E\_DeviationsAndRecSeqs \\
E\_SequencesAndHistories \\
I_e : \mathbb{P}\, ETRANSITION \\
ED : \mathbb{P}\, ETRANSITION \\
RS : \mathbb{P}(\mathrm{seq}_1\, ETRANSITION) \\
\hline
I_e = T_{ef} \setminus R_t \\
ED = \{t_e : T_{ef} \mid (first \circ ETrans)t_e \in R_s \wedge (second \circ ETrans)t_e \notin R_s\} \\
RS = \{s : \mathrm{seq}_1\, ETRANSITION \mid s \in S_{t_e} \wedge \\
\quad (first \circ ETrans \circ s)\ 1 \notin R_s \wedge (second \circ ETrans \circ s)\ \#s \in R_s\} \\
\hline
\end{array}
$$

EIs can be modeled in E as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad E\_Inconsistencies \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
E \\
WI_s : \mathbb{P}\, ESTATE \\
SI_s : \mathbb{P}\, ESTATE \\
\hline
WI_s = \{s_e : ESTATE \mid s_e \in S_{pss} \times S_{bp} \wedge s_e \notin R_s \wedge \\
\qquad (\exists\, s : RS \mid (first \circ ETrans \circ s)\ 1 = s_e)\} \\
< R_s, WI_s, SI_s > \ partition\ S_{pss} \times S_{bp} \\
\hline
\end{array}
$$

This schema defines two additional sets:

- WI_s is the *set of weakly inconsistent states of E*. A state of E belongs to WI_s if the state of PSS does not correctly describe the state of BP, but a reconciling sequence exists that starts from it and takes E in a state that belongs to R_s. This means that the inconsistency can be fixed without modifying the structure of BP and PSS (and, consequently, $\mathcal{BP}$ and $\mathcal{PSS}$ do not need to be modified).

- SI_s is the *set of strongly inconsistent states of E*. It includes all states of E in which the state of PSS does not correctly describe the state of BP, and from which it is not possible to execute any reconciling sequence. If E reaches one of these states, the structure of either BP or PSS has to be modified to reconcile them. In particular, either a new state is added to PSS to represent the state reached by BP and a reconciling sequence is defined for E, or BP is changed in order to enter a state that can be represented by some existing state of PSS.

The union of WI_s and SI_s represents the set of all environment-level inconsistent states.

In Example 3, the state (`pss1`, `bp2`), reachable from state (`pss1`, `bp1`) through event (I_{pss1},`BP_ACreatePaper`), is weakly inconsistent. In fact, feasible event (`PSS_PLCreatePaper`, I_{bp2}) brings E in state (`pss2`, `bp2`), that belongs to R_s. Conversely, state (`pss3`, `bp13`), reachable from state (`pss3`, `bp3`), through event (I_{pss3},`BP_PLChooseRev3`) is a strongly inconsistent state.

The following new schema for E contains all the definitions given so far:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad E_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
E\_Inconsistencies \\
E\_DeviationsAndRecSeqs \\
\hline
\end{array}
$$

3.4 $\mathcal{E}$ completeness and coherence

One of the main requirements for a process support system is to support the corresponding business process, avoiding the occurrence of deviations and, consequently, of inconsistencies.

In this section two properties of an environment $\mathcal{E}$ are defined, completeness and coherence, that are indicators of the ability of a process support system $\mathcal{PSS}$ to support a business process $\mathcal{BP}$.

3.4.1 Definition of completeness and coherence

As mentioned, often, process support systems do not completely describe the corresponding business process because of its intrinsic complexity. Therefore, the two systems are likely to diverge. In our framework, when PSS does not completely describe BP, E is said not to be state and/or transition complete.

Formally, E is *state complete* iff, for each state $s_{bp} \in S_{bpr}$, there is at least one state $s_{pss} \in S_{pss}$ that correctly describes it. The following schema defines a state complete environment:

$$
\begin{array}{l}
\textit{E\_StateComplete} \\
\hline
E_1 \\
\hline
\forall s_{bp} : S_{bpr} \bullet \exists s_{pss} : S_{pss} \bullet s_{pss} \mapsto s_{bp} \in R_s
\end{array}
$$

Similarly, E is *transition complete* iff every BP transition t_{bp} of some event that belongs to a history of E is described by at least one transition of PSS. The following schema defines a transition complete environment:

$$
\begin{array}{l}
\textit{E\_TransitionComplete} \\
\hline
E_1 \\
\hline
\forall t_{bp} : T_{bp} \mid (\exists h_e : PH_e \bullet \exists i : \mathrm{dom}\, h_e \bullet t_{bp} = (\textit{second} \circ h_e) i) \bullet \\
\quad (\exists t_{pss} : T_{pss} \bullet t_{pss} \mapsto t_{bp} \in R_t)
\end{array}
$$

State and transition completeness are static concepts describing the ability of PSS to represent BP's states and transitions. These properties do not ensure that EDs never occur. An ED may still occur even in a state and transition complete environment, when the transition executed in PSS does not correctly describe the transition occurred in BP.

If an environment E is built in such a way that all events of any possible history belong to R_t, EDs never occur. A similar environment is *coherent*. In a coherent environment any transition occurring in BP is always mirrored by the transition of PSS that correctly describes it.

A coherent environment is formally described by the following schema:

$$
\begin{array}{l}
\textit{E\_Coherent} \\
\hline
E_1 \\
\hline
\forall h_e : PH_e \bullet \forall i : \mathrm{dom}\, h_e \bullet h_e\ i \in R_t
\end{array}
$$

It is easy to prove that the state of a coherent environment E, during its evolution, always remains in R_s.

It is also possible to prove the following theorems which relate state completeness, transition completeness, and coherence.

Theorem 1 *State completeness is a necessary condition for transition completeness.*

Proof (by contradiction). Suppose that E is transition complete, but it is not state complete. Then, a state $s_{bp} : S_{bpr}$ must exists, such that the following condition holds:

$$\neg\ (\exists\, s_{pss} : S_{pss} \bullet s_{pss} \mapsto s_{bp} \in R_s) \tag{3.1}$$

There are two possible cases:

i. $s_{bp} = s^i_{bp}$ or

ii. $s_{bp} \neq s^i_{bp} \land \exists\, s'_{bp} : BPSTATE, t_{bp} : T_{bp} \bullet s'_{bp} \in S_{bpr} \land s'_{bp} = (\mathit{first} \circ BPTrans) t_{bp} \land s_{bp} = (\mathit{second} \circ BPTrans) t_{bp}$.

If case (i) holds, from the definition of R_s, it follows that at least the initial state s^i_{pss} of PSS, exists such that $s^i_{pss} \mapsto s_{bp} \in R_s$. This contradicts Formula 3.1.

If case (ii) holds, from the transition completeness hypothesis, we deduce that there are two states $s_{p_1}, s_{p_2} : S_{pss}$, and a PSS transition $t_{pss} : T_{pss}$ such that:

$$s_{p_1} = (\mathit{first} \circ PSSTrans) t_{pss} \land s_{p_2} = (\mathit{second} \circ PSSTrans) t_{pss} \land$$
$$t_{pss} \mapsto t_{bp} \in R_t$$

Therefore, from axiom on R_t, we can deduce that $s_{p_2} \mapsto s_{bp} \in R_s$. This contradicts Formula 3.1. □

Theorem 2 *Transition completeness is a necessary condition for coherence.*

Proof. Suppose that E is coherent. Then for each transition $t_{bp} : T_{bp}$, such that:

$$t_{bp} = (\mathit{second} \circ h_e)\ i$$

where $i : \mathrm{dom}\, h_e$ and $h_e : PH_e$, at least the transition of PSS:

$$t_{pss} = (\mathit{first} \circ h_e)\ i$$

exists that correctly describes it. This proves that E is transition complete. □

Remark According to the previous theorems, it is possible to assert that state completeness, transition completeness, and coherence are increasing levels of environment quality.

3.4.2 Metrics for completeness and coherence

Using the concepts introduced so far, partial order relationships among environments may be defined. These partial orders are related to the ability of a process support system $\mathcal{PSS}$ to describe and support a business process $\mathcal{BP}$. In order to facilitate the formal presentation of these partial orders, some new definitions in the schema that defines the model of E are introduced:

$$
\begin{array}{|l}
\hline
\quad E_2 \\
E_1 \\
S_{undes} : \mathbb{P}\, BPSTATE \\
T_{undes} : \mathbb{P}\, BPTRANSITION \\
PH_{incorr} : \mathbb{P}(\mathrm{seq}_1\, ETRANSITION) \\
\hline
S_{undes} = \{ s_{bp} : S_{bpr} \mid (\neg\, (\exists\, s_{pss} : S_{pss} \bullet s_{pss} \mapsto s_{bp} \in R_s)) \} \\
T_{undes} = \{ t_{bp} : T_{bp} \mid \\
\qquad ((\exists\, h_e : PH_e \bullet \exists\, i : \mathrm{dom}\, h_e \bullet t_{bp} = (second \circ h_e) i) \\
\qquad\quad \wedge \\
\qquad \neg\, (\exists\, t_{pss} : T_{pss} \bullet t_{pss} \mapsto t_{bp} \in R_t)) \} \\
PH_{incorr} = \{ h_e : PH_e \mid (\exists\, i : \mathrm{dom}\, h_e \bullet h_e\ i \notin R_t) \} \\
\hline
\end{array}
$$

S_{undes} is the set of all the reachable states of BP that are not described by any PSS state. T_{undes} is the set of all state transitions of BP belonging to some E history that are not described by any PSS transition. Finally, set PH_{incorr} is the set of all non-coherent E histories.

Partial orders can be defined with respect to state completeness, transition completeness, and coherence.

Let $\mathcal{P}\hat{\mathcal{S}}\mathcal{S}$ and $\mathcal{P}\tilde{\mathcal{S}}\mathcal{S}$ be two process support systems that support the same business process $\mathcal{BP}$. Let $\hat{\mathcal{E}}$ and $\tilde{\mathcal{E}}$ be the environments composed of $\mathcal{P}\hat{\mathcal{S}}\mathcal{S}$ and $\mathcal{BP}$, and of $\mathcal{P}\tilde{\mathcal{S}}\mathcal{S}$ and $\mathcal{BP}$, respectively. Also, let $\hat{E}$, $\tilde{E}$, $P\hat{S}S$, $P\tilde{S}S$, and BP be the models of $\hat{\mathcal{E}}$, $\tilde{\mathcal{E}}$, $\mathcal{P}\hat{\mathcal{S}}\mathcal{S}$, $\mathcal{P}\tilde{\mathcal{S}}\mathcal{S}$, and $\mathcal{BP}$, respectively. $\hat{\mathcal{E}}$ is *more state complete* than $\tilde{\mathcal{E}}$ (written $\hat{E} \geq_{sc} \tilde{E}$) iff the number of BP states that belong to the set S_{undes} of $\hat{E}$ is less or equal than the number of BP states that belong to S_{undes} of $\tilde{E}$. Formally, operator $\geq_{sc}$ is defined as an infix relation on $E_2 \times E_2$, that satisfies the following axiom:

$$
\begin{array}{|l}
\_ \geq_{sc} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall\, \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{sc} \tilde{E} \Leftrightarrow \\
\qquad (\hat{E}.S_{bp} = \tilde{E}.S_{bp} \wedge \hat{E}.T_{bp} = \tilde{E}.T_{bp} \wedge \hat{E}.s^i_{bp} = \tilde{E}.s^i_{bp} \wedge \\
\qquad (\exists\, f_{sc} : \hat{E}.S_{undes} \rightarrowtail \tilde{E}.S_{undes}))
\end{array}
$$

Similar definitions are given for operators $\geq_{tc}$ and $\geq_c$, which define an ordering relation between different environments with respect to transition completeness and coherence, respectively:

$$
\begin{array}{|l}
\_ \geq_{tc} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall\, \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{tc} \tilde{E} \Leftrightarrow \\
\qquad (\hat{E}.S_{bp} = \tilde{E}.S_{bp} \wedge \hat{E}.T_{bp} = \tilde{E}.T_{bp} \wedge \hat{E}.s^i_{bp} = \tilde{E}.s^i_{bp} \wedge \\
\qquad (\exists\, f_{tc} : \hat{E}.T_{undes} \rightarrowtail \tilde{E}.T_{undes}))
\end{array}
$$

$$
\begin{array}{|l}
\_ \geq_{c} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall\, \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{c} \tilde{E} \Leftrightarrow \\
\qquad (\hat{E}.S_{bp} = \tilde{E}.S_{bp} \wedge \hat{E}.T_{bp} = \tilde{E}.T_{bp} \wedge \hat{E}.s^i_{bp} = \tilde{E}.s^i_{bp} \wedge \\
\qquad (\exists\, f_{c} : \hat{E}.PH_{incorr} \rightarrowtail \tilde{E}.PH_{incorr}))
\end{array}
$$

Example 4 *Let $\hat{PSS}$ be the process support system described in Example 2 (see Figure 3.2) and supporting the business process BP of Example 1 (see Figure 3.1). The resulting environment $\hat{E}$ can be compared with the environment $\tilde{E}$, composed of BP itself and of the model $\tilde{PSS}$ described by the state machine shown in Figure 3.3. Using the previous definitions, it can be shown that $\hat{E} \geq_{sc} \tilde{E}$ and $\hat{E} \geq_{tc} \tilde{E}$. In fact, it is easy to find a function f_1 from $\hat{E}.S_{undes} = \{$*`bp12`, `bp13`, `bp14`*$\}$ to $\tilde{E}.S_{undes} = \{$*`bp11'`, `bp12'`, `bp13'`, `bp14'`*$\}$ and a function f_2 from $\hat{E}.T_{undes} = \{$*`BP_ACreatePaper`, `BP_PLChooseRev3`, `BP_PLChooseRev4`, `BP_OEditSec3`, `BP_OEditSec4`, `BP_REditSec3`, `BP_REditSec4`, `BP_Deliver3`, `BP_Compile4`, `BP_Print4`, `BP_AllSecDelivered3`, `BP_AllSecDelivered4`*$\}$ to $\tilde{E}.T\_undes = \{$*`BP_Deliver1'`, `BP_ACreatePaper'`, `BP_PLChooseRev3'`, `BP_PLChooseRev4'`, `BP_OEditSec3'`, `BP_OEditSec4'`, `BP_REditSec3'`, `BP_REditSec4'`, `BP_Deliver3'`, `BP_Compile4'`, `BP_Print4'`, `BP_AllSecDelivered3'`, `BP_AllSecDelivered4'`*$\}$ that are injective but not surjective, as required by the formal definition of operators $\geq_{sc}$ and $\geq_{tc}$, respectively.* □

If the level of completeness and coherence of an environment is low, then EDs are likely to occur. In this case, the level of support provided to a business process may be enhanced by replacing the corresponding process support system with another system that provides a higher level of state completeness, transition completeness, and coherence. If the new PSS only adds states and transitions to the older one and does not eliminate any of the existing states and transitions, then the resulting environment (let it be $\hat{E}$) *extends* the older one (let it be $\tilde{E}$), and we write $\hat{E} \sqsupseteq \tilde{E}$. Formally:

$$
\begin{array}{|l}
\_ \sqsupseteq \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \sqsupseteq \tilde{E} \Leftrightarrow \\
\qquad (\hat{E} \geq_{sc} \tilde{E} \wedge \hat{E} \geq_{tc} \tilde{E} \wedge \hat{E} \geq_{c} \tilde{E} \wedge \\
\qquad \hat{E}.S_{undes} \subseteq \tilde{E}.S_{undes} \wedge \hat{E}.T_{undes} \subseteq \tilde{E}.T_{undes} \wedge \\
\qquad \hat{E}.PH_{incorr} \subseteq \tilde{E}.PH_{incorr})
\end{array}
$$

Example 5 *Consider again the environment $\hat{E}$ composed of the process support environment $\hat{PSS}$ shown in Figure 3.2 and of the business process BP shown in Figure 3.1. This environment may be extended by replacing $\hat{PSS}$ with the new process support system, $\overline{PSS}$, shown in Figure 3.4. In fact, all the BP states and transitions described in $\hat{PSS}$ have a correspondent state or transition in $\overline{PSS}$. Moreover, all the BP states and transition are described in $\overline{PSS}$ with the exception of transition* `BP_ACreatePaper`.
□

3.4.3 A classification of possible extensions

The previous section shown that if an environment $\mathcal{E}$ does not offer the level of completeness and coherence that is required by the application domain, it should be modified. In order to modify $\mathcal{E}$, $\mathcal{PSS}$ can be changed to improve its ability to describe $\mathcal{BP}$, or $\mathcal{BP}$ can be changed to reduce the set of reachable states (and/or transitions)

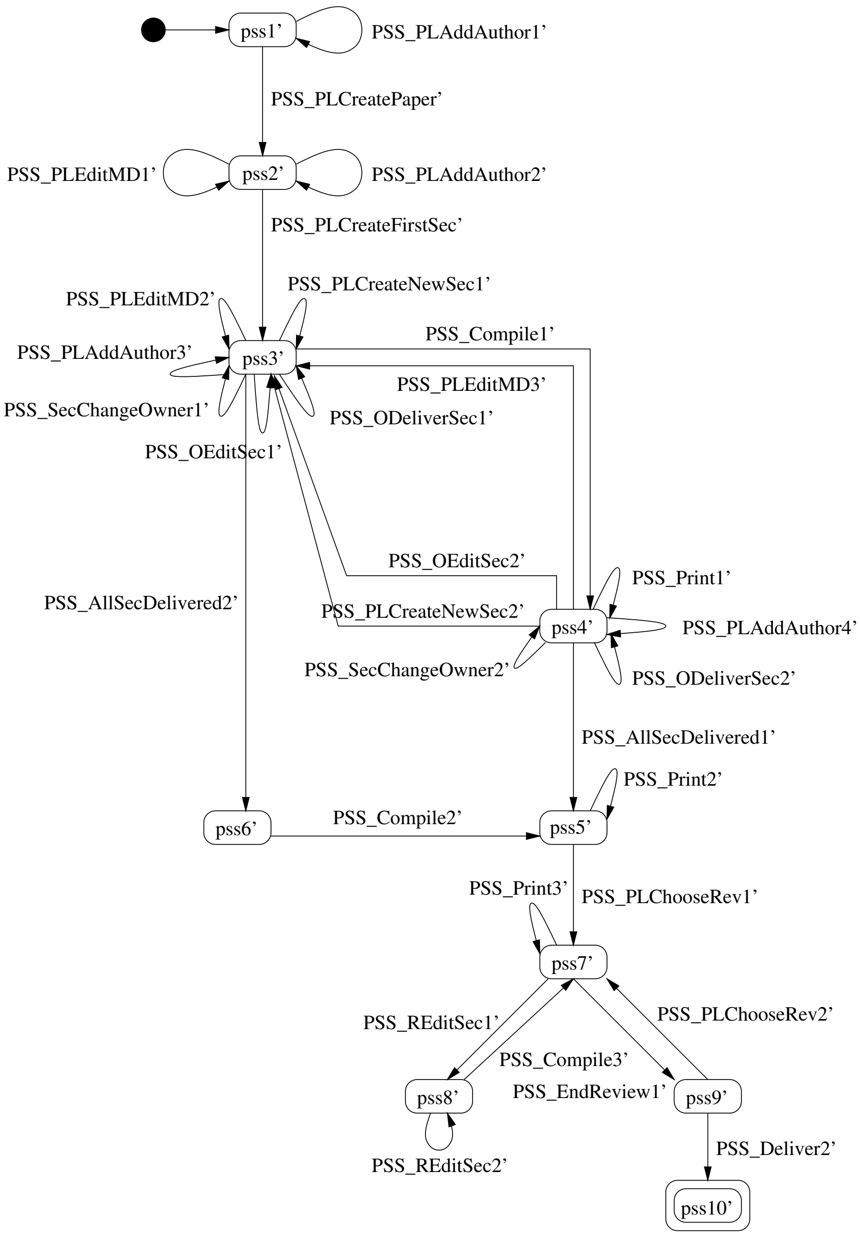

Figure 3.3: The process support system model $\tilde{PSS}$.

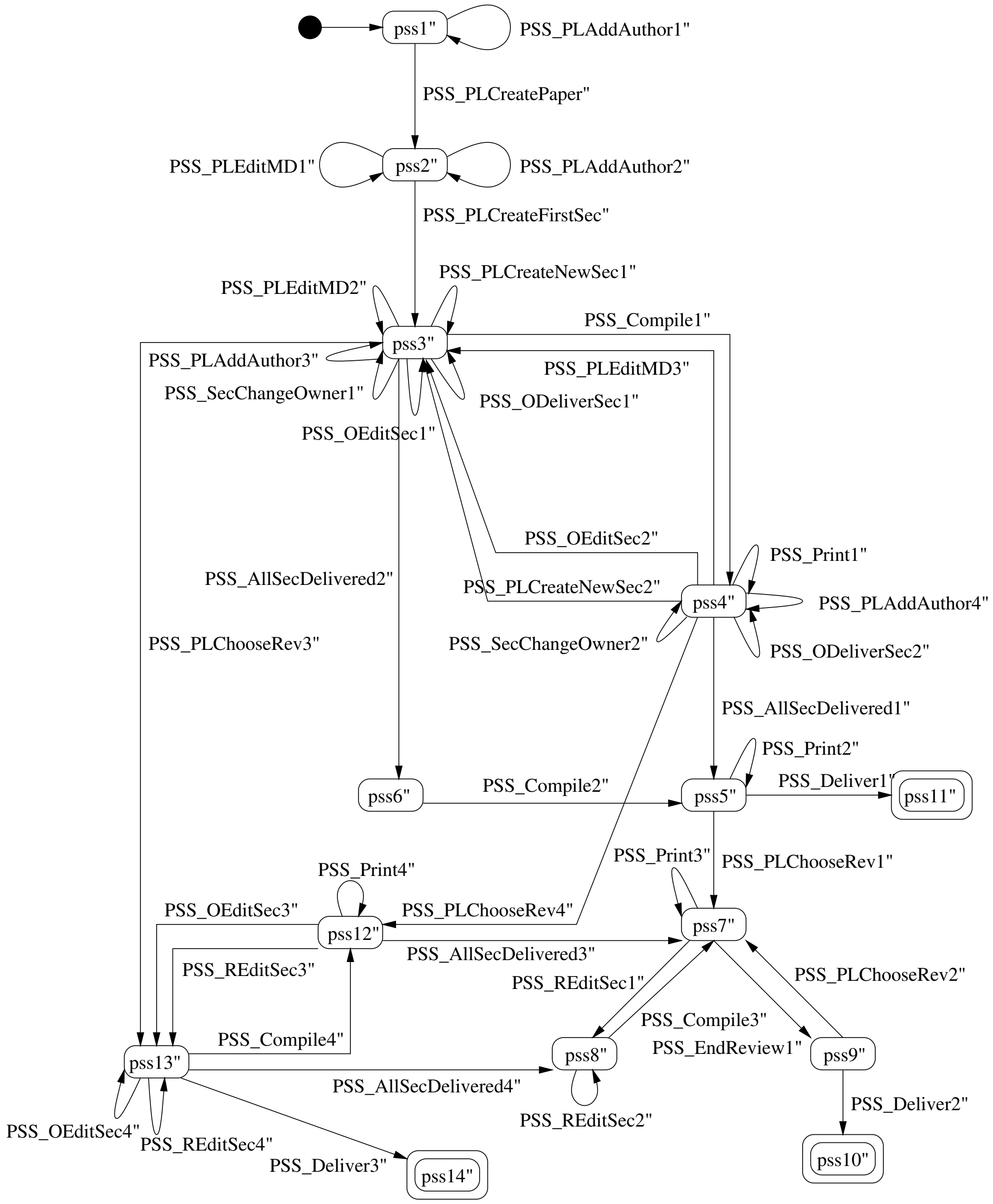

Figure 3.4: The process support system model $\overline{PSS}$.

that $\mathcal{PSS}$ does not represent. Finally, environment coherence can be improved by reducing the difference between T_{ef} and R_t in order to decrease the number of possible incorrect histories.

More specifically:

1. PSS (and consequently $\mathcal{PSS}$) can be extended in two ways:
   (a) S_{pss} is extended to reach state completeness. In this case, transitions to T_{pss} need to be added, to make the new states reachable. R_s and R_t are also modified in order to relate the newly added states and transitions with the corresponding ones in S_{bp} and T_{bp}.
   (b) T_{pss} is extended to reach transition completeness. R_t is also extended to describe the relationships between the new PSS transitions and the corresponding ones in T_{bp}.
2. BP (and consequently $\mathcal{BP}$) can be modified in two ways:
   (a) The set of BP states not described by any PSS state is reduced in order to increase state completeness.
   (b) The set of BP transitions not described by any PSS transitions is reduced in order to increase transition completeness.

   These solutions represent changes to the business process to match its computerized support. This can be accomplished by limitating the freedom of users. For instance, they could be prevented from using tools whose behavior is not controlled by the process support system.
3. Coherence can be enhanced by reducing the set of feasible events T_{ef} that do not belong to R_t, or by extending set R_t.

3.5 Applying the model

This chapter defined the concepts of deviations and inconsistencies that can be used to precisely characterize the problem of tolerating and controlling deviations and inconsistencies in business processes. In this section, these concepts are applied to describe and compare the characteristics of three PSEEs, namely, SPADE [12, 10], SENTINEL [26], and Provence [17] with a special attention to the features they offer in order to help their users when unforeseen situations arise. This comparison will allow us to identify the critical issues to deal with deviations and inconsistencies in PSSs.

These PSEEs were chosen because they are representative of the three different ways PSEEs (and, more in general, PSSs) are related to the business process they support. In particular, as the majority of existing PSEEs, SPADE assumes that the humans involved in the development process do not change the way they work before changing the process model (using the process evolution mechanisms offered by the PSEE).

Conversely, SENTINEL, like PROSYT, assumes that humans can deviate from the specified process model, and relies on its flexibility to represent deviations in its internal state.

Finally, Provence provides a mechanism for monitoring the business process in a non-intrusive way, and, therefore, can detect, in principle, both correct and deviating events.

Sections from 3.5.1 to 3.5.3 provide a description of these three PSEEs, exploiting the framework presented in this paper. Section 3.6 briefly draws some lessons derived from the comparison of the above PSEEs, highlighting the relationship between the completeness and coherence properties, and the architecture and features of these PSEEs.

3.5.1 SPADE

SPADE [12] is a software engineering environment that supports software process analysis, design, enactment [11] and evolution [10]. SPADE is based on a process modeling language, called SLANG [14, 13], which, in turn, is based on high-level Petri nets. In SLANG process artifacts, including process models, are modeled as tokens of a Petri net and are implemented as objects in an object-oriented database. Transitions of the Petri net are associated with a *guard* and an *action*. The guard is a precondition that has to be satisfied by a tuple in order to be selected to enable the firing of the transition. The action describes how the firing of the transition generates new token values in the output places as a function of the values of the token removed from the input places. Suitable constructs are also provided to describe the interaction with software engineering tools and process agents in a uniform style.

The SPADE environment

The SPADE architecture [7] is structured in three different layers (see Figure 3.5): the Repository, the Process Enactment Environment (PEE), which includes the Process Engine, and the User Interaction Environment (UIE), which includes the SPADE Communication Interface (SCI).

The Repository stores both process model fragments and process artifacts. A process artifact can be any document or information created during the software development process. The Repository is implemented on top of the object-oriented database management system O_2 [71].

The PEE supports execution of a SLANG process model, which can be composed of different activities (i.e., Petri net fragments). During enactment, different instances of the same activity, called *active copies*, may be created. Active copies are concurrently executed by different SLANG interpreters, implemented as separate threads of a Process Engine.

The UIE is the SPADE front-end to its users. It is composed of the tools used by the process agents involved in the development activities and controlled by the PEE. Black-box tools are seen by the PEE as a single function that receives some input and produces some output. SPADE controls only the invocation and the termination of these tools. Often, these tools use their own repository for data (typically the file system). This means that all the process data stored in the O_2 database have to be explicitly converted and mirrored in the file system to make it possible for tools to operate on them. Service-based tools offer an interface through which it is possible to invoke all the individual services they provide. Moreover, each tool can invoke

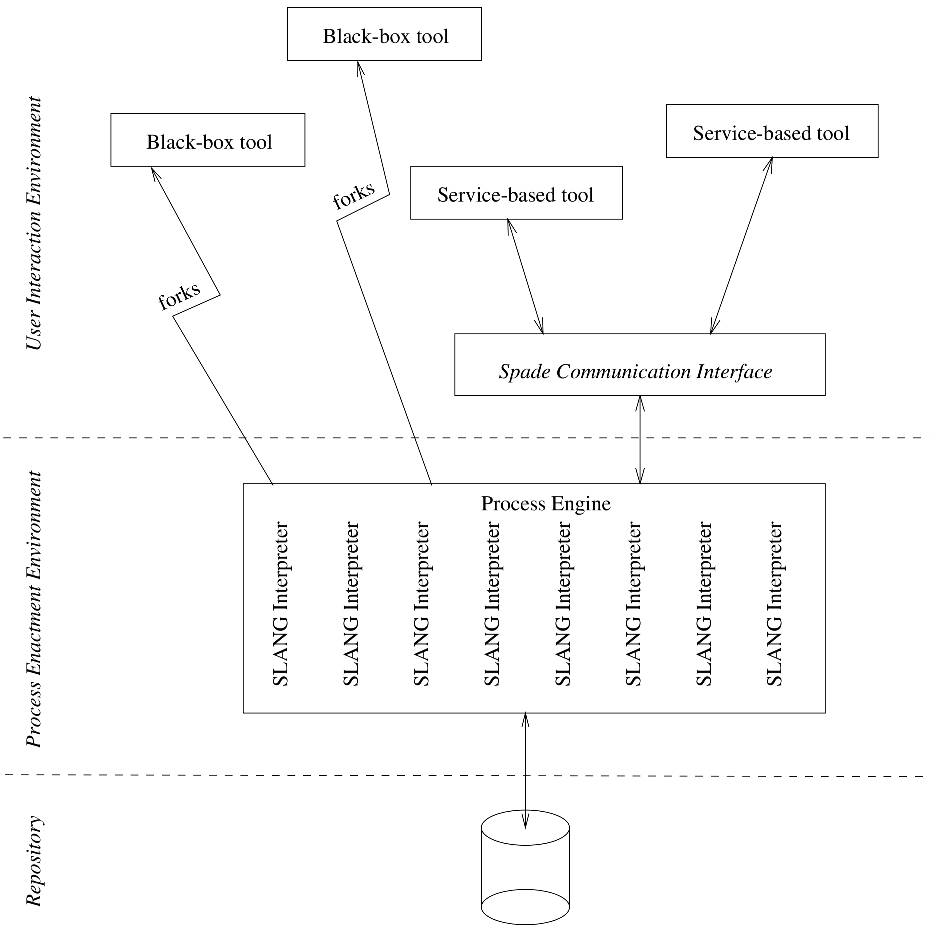

Figure 3.5: The SPADE Architecture.

external services offered by other tools and notify the PEE of specific events or tool state changes. SCI manages the communication between service-based tools and the PEE, through the SPADE Communication Protocol. SCI is connected to all service-based tools and to the Process Engine executing the different SLANG Interpreters; it is able to redirect messages from the UIE to the enacting active copies.

SPADE offers support to process evolution through the reflectivity features of its PDL, SLANG, that is, process models may be modified as any data by other processes (*metaprocesses*). A detailed presentation of this issue may be found in [10].

Completeness and coherence in SPADE

SPADE (as most PSEEs) is based on two principles:

1. All operations performed in the software process must be *explicitly* described in the process model. Whenever the process manager wants to enable the PSEE to accomplish some new operation, he or she needs to introduce a specific process model fragment that implements it.

   The process model can deal with DIs and DDs. This can be accomplished by explicitly introducing new process model fragments describing such situations. Referring to the process example introduced in Section 2.2.1, suppose that the different sections which compose the paper being written are represented in the process model as SLANG tokens. Suppose also that when an author delivers the section he or she is responsible for, the corresponding token reach a place `P`. With these premises, the DD `BP_Deliver1` may be explicitly described by adding a transition that is allowed to fire when all the token that represent the paper's sections have reached the place `P` and that terminates the process by delivering the paper.

2. SPADE assumes that the users operate under its full control. That is, any operation related to the software process is accomplished through SPADE. This allows SPADE to consistently update its internal state according to the events occurring in the business process.

   To support this principle, SPADE offers mechanisms to integrate tools (the SCI and the SCI protocol), so that any tool service request issued by a user can be detected and handled by SPADE. For instance, if the user requests to open a file using an editor command, the process engine is notified of this request and can decide whether the operation can be authorized.

   SPADE also offers a specific tool (called the SPADEShell) to replace the Unix shell. This tool enables the user of SPADE to issue requests as in the traditional shell. These requests are filtered by the Process Engine that executes them according to the rules and constraints of the process model being enacted.

Clearly, when these principles are violated, then EDs occur. As an example of ED in SPADE, consider the SLANG process fragment shown in Figure 3.6, which describes activity `CreatePaper` that crates the first version of the master document. Suppose that, according to the process description given in Section 2.2.1, only the project leader is allowed to perform this activity. In the SLANG fragment of Figure 3.6, a guard is associated with transition `StartActivity` to guarantee that both

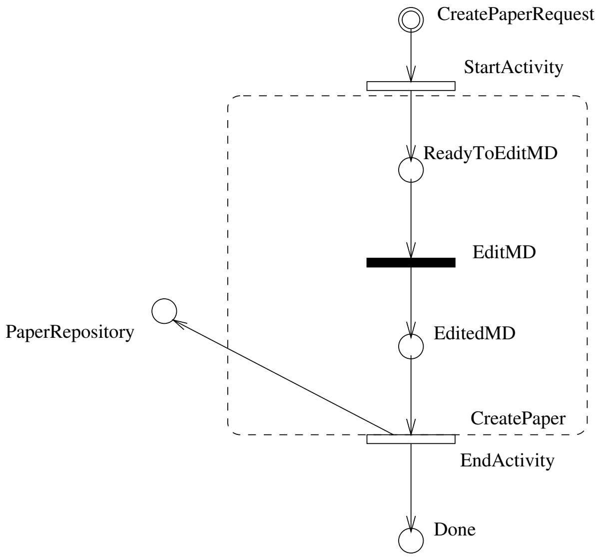

Figure 3.6: SLANG activity `CreatePaper`.

the requests to create the paper and to add it to the repository are performed by the project leader. Suppose that a different agent wants to perform this activity, and that he or she issues this command through a tool that has been integrated in SPADE. The command is reified in place `CreatePaperRequest` as a token, but it does not cause the firing of transition `StartActivity`, since its guard evaluates to false. That is, by using SPADEShell or any other tool that has been integrated in SPADE, this agent will never be allowed to accomplish the operation.

Let us assume that for some reason the process manager wants to enable this user to create the first version of the master document, even if this operation violates some process constraints. In SPADE, the process manager may explicitly (and dynamically) introduce a new process fragment that enables this operation. This fragment might be defined so that the master document is suitably annotated to record the anomalous situation and to keep it under control. Thus, completeness would be increased by "explicitly" taking the DD and DI into account.

Even without introducing this new process fragment, the master document can be created outside the control of SPADE if the agent is allowed to access a standard Unix shell. In such a case, the state of SPADE would not change, while the state of the business process would be different because now the master document exists. Therefore, an EI can still be introduced because of the inability of the system to guarantee coherence (e.g., there is a way to bypass SPADE). To eliminate this EI, the process manager should preclude the possibility for the agent to execute any operation which creates and edits files outside the control of SPADE. This choice would reduce the set of feasible events that do not belong to R_t (see Section 3.4.3).

Using the framework introduced in this chapter, it is possible to describe this situation more precisely. Figure 3.7 shows the E model describing activity `CreatePaper`. Transition `PLCreatePaper` in the business process model BP describes the activity

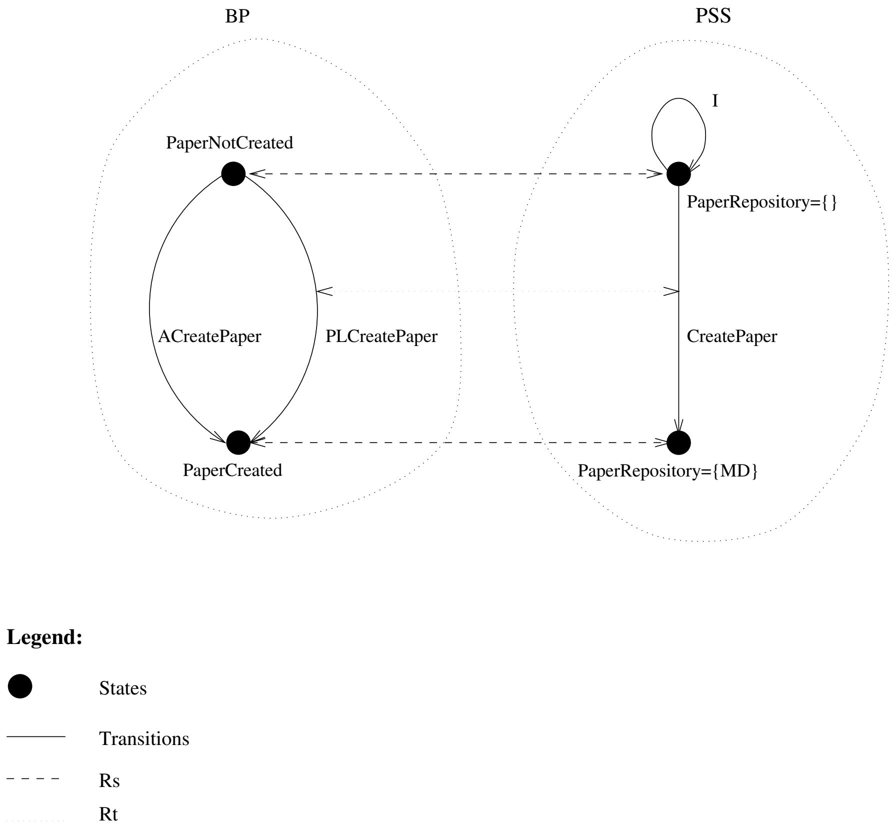

Figure 3.7: The model of the environment composed of SPADE enacting activity `CreatePaper` and the corresponding business process.

of creating the first version of the master document performed by the project leader by invoking the SPADE activity `CreatePaper`, while transition `ACreatePaper` describes the same activity performed by a different user by directly invoking an editor (i.e., out of the SPADE control). Transition `CreatePaper` in the PSS model PSS describes the actions performed by SPADE when the project leader invokes the activity `CreatePaper`.

When a user, other than the project manager, creates the master document by using the Unix editor outside control of SPADE, $\mathcal{BP}$ deviates from $\mathcal{P}$. This ED is described by the event $(I, ACreatePaper)$ and brings E in an environment-level inconsistent state.

The SPADE approach to achieve completeness and coherence can therefore be summarized as follows:

Completeness SLANG supports the *explicit* modeling of the process. SLANG can be used to model DIs, but completeness can be enhanced only by explicitly modeling each specific situation. This means that extending R_s and R_t may be rather cumbersome and expensive.

Coherence SPADE provides powerful mechanisms for controlling the tools integrated in SPADE and used by software developers. Operations supported by tools integrated in SPADE are detected by the Process Engine, and used to update the SPADE internal state. Coherence is therefore supported by *exploiting the control integration facilities* offered by the PSEE. Clearly, if the user can access tools that are only partially integrated within SPADE, EIs can occur. In general, the difference between T_{ef} and R_t is reduced by prohibiting those actions that are not controlled by SPADE.

Although our discussion has been based on the specific features offered by SPADE, these conclusions hold for most existing PSSs, since they are based on a similar underlying philosophy.

3.5.2 SENTINEL

The goal of the SENTINEL (Software-engineering Environment to Tolerate INconsistencies Employing Logic) project [26] is similar to the goal of PROSYT[3]: to provide mechanisms to avoid EDs by automatically enhancing state and transition completeness thanks to an innovative approach to enact a process model. SENTINEL does not force users to rigidly follow the process model. Rather, it allows them to somehow deviate from the stated process model.

The SENTINEL environment

The SENTINEL environment is based on a process modeling language called LATIN (LAnguage to Tolerate INconsistencies). Software processes are modeled in LATIN as collections of task types[4]. Each task type describes a software process activity as a state

[3]As mentioned, PROSYT can be considered the final result of the research effort started by developing SENTINEL, whose goal was to find the best mechanisms to increase PSSs usability by allowing users to deviate from the process model when an unexpected situation is encountered.

[4]LATIN has been defined extending ASTRAL [51], a formalism for specifying real time systems.

machine. State transitions are characterized by a *precondition*, called ENTRY, and a *body*. The ENTRY is a logical proposition defining the property that must be satisfied to execute the corresponding transition. In the body, two different classes of operations can be performed: *actions*, and *value assignments* (EXIT clause). Actions produce results in the business process (e.g., invoking a tool). Value assignments modify the values of the state variables of the task.

In LATIN, *invariants* are logical propositions that have to be true in any state of the system. Invariants can be global or local depending on whether they define the critical requirements for the whole process or for a single task type.

The run-time representation of the process model is composed of a set of concurrently executing *task instances*, that are dynamically instantiated from the corresponding task type.

LATIN offers two kinds of transitions: *normal transitions* and *exported transitions*. A normal transition is executed as soon as its ENTRY evaluates to true. If more than one transition precondition evaluates to true, then one of them is chosen nondeterministically. An exported transition is executed if the user requests it and its ENTRY is true. However, the user can still force the execution of an exported transition if its ENTRY is not verified. In such a case, the transition *fires illegally*.

If a user forces an exported transition to fire when its precondition is false, a deviation from the prescribed process model occurs. This deviation does not result in an environment-level deviation. In fact, SENTINEL is able to represent such deviation in its internal state. In particular it records relevant events occurred during enactment in a knowledge base and uses this information to determine (through logical reasoning) the occurred deviations, and the possibly corrupted data (*polluted data*). The enactment is suspended only if an invariant is violated. In such a situation SENTINEL automatically starts a *pollution analysis*. The results of this analysis are a complete description of deviations from the defined process model occurred during enactment and the identification of polluted data caused by such deviations. This information can be used by the process manager in order to drive the process that tries to reconcile $\mathcal{BP}$ and $\mathcal{PSS}$.

Completeness and coherence in SENTINEL

The ability of SENTINEL to describe deviations from the process model stems from the degree of state and transition completeness that it is able to ensure. SENTINEL's enactment mechanism automatically increases the number of transitions of PSS (i.e. the set T_{pss}) with respect to the number of transitions explicitly defined in the process model, by "implicitly" including exported transitions whose preconditions are not satisfied. Moreover, SENTINEL increases the number of modeled states, by "implicitly" adding states in which data are polluted.

As a consequence, even DDs and DIs not explicitly described in the process model, can have in SENTINEL corresponding transitions and states, respectively. Therefore, SENTINEL is able to tolerate DDs and DIs, thus reducing the chance that EDs occur.

The following example shows SENTINEL's ability to automatically tolerate DDs even if they are not explicitly described in the process model, by increasing state and transition completeness. Consider the activity `CreatePaper` described in the example of Section 3.5.1. This activity can be modeled in LATIN by the transition shown in Figure 3.8.

```
EXPORTED_TRANSITION CreatePaper( u: User; MDName: STRING )
ENTRY
        NOT EXISTS Doc IN PaperRepository (Doc.Name=MDName)
        AND u.Role=PL
LOCAL
        md: MasterDocument;
ACTION
        Call(``vi '' + MDName + ``.tex'');
EXIT
        md.Name=MDName; md.File=MDName + ``.tex'';
        Add(PaperRepository,md);
```

Figure 3.8: LATIN transition `CreatePaper`.

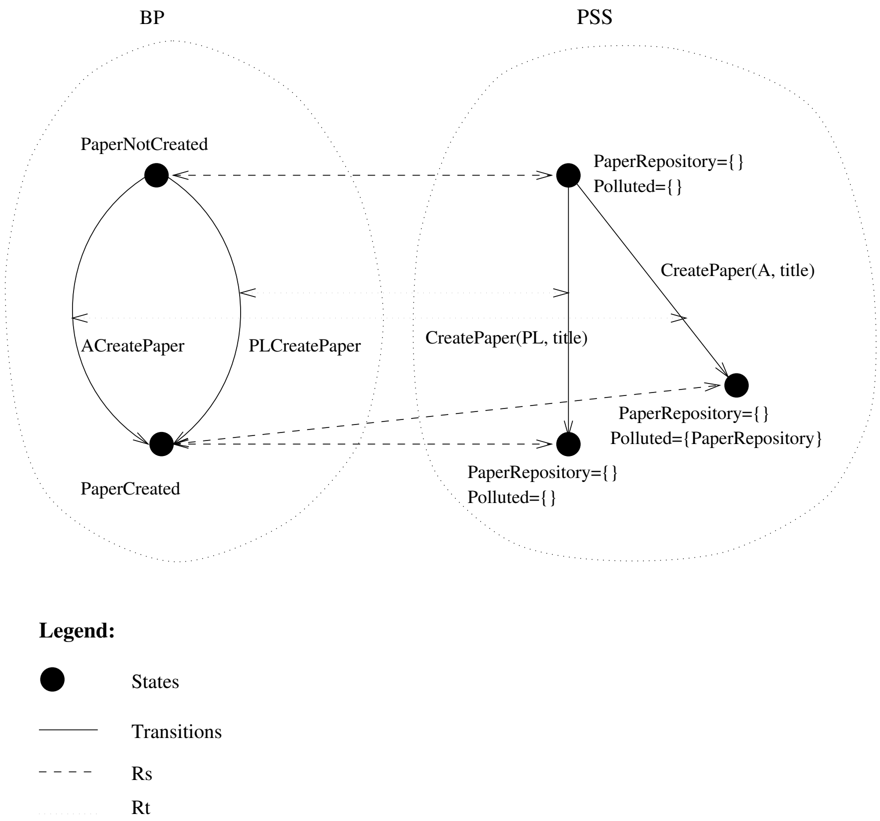

Figure 3.9: The model of the environment composed of SENTINEL enacting transition `AddRFC` and the corresponding business process.

Figure 3.9 shows the PSS model implemented by SENTINEL enacting this specification fragment. By comparing this figure with Figure 3.7 we may observe that transition `ACreatePaper` (which does not have a corresponding transition in SPADE) is described in SENTINEL by transition `CreatePaper(A, MDName)`, and that state `PaperCreated` has two states of SENTINEL describing it.

This example shows that the user can deviate from the modeled process by firing transition `CreatePaper` under control of SENTINEL, even if he or she is not the project leader. The final state of E is consistent and the PSS has all the necessary information to determine the effects of the deviation. Thus, SENTINEL allows state and transition completeness of $\mathcal{E}$ to be easily enhanced.

As for coherence, the present SENTINEL prototype has very simple tool integration facilities. Thus, it is quite easy to bypass it and generate EIs.

In conclusion, the SENTINEL completeness and coherence characteristics can be summarized as follows:

Completeness LATIN makes it easy to enhance completeness, by *implicitly* adding all the states describing illegally fired transitions.

Coherence No specific mechanisms are offered, except for basic tool integration aids. SENTINEL assumes that the user does not try to perform actions outside its control.

3.5.3 Provence

Provence [17] is a PSEE that explicitly separates the enactment of the software process model from the actual execution of the software process. Provence is based on an event monitor that detects the occurrence of events in the business process and notifies the process support system. The PSEE uses the information provided by the event monitor to define the actual state of the process support system and to determine whether software process execution is consistent with the process model.

The Provence environment

Provence is based on an event-action monitor called Yeast [63, 72], which provides mechanisms to detect significant events occurring in the business process and to trigger specific actions that have been associated with those events. Provence monitors the execution of the software process by translating high-level process tasks into low-level event-action specifications, involving file-based events and tool invocation events that are relevant to the process. The occurrence of a sequence of low-level events indicates the occurrence of a particular task in the process. Yeast is responsible for notifying the process model interpreter whenever an event occurs. The process model supplied to Provence describes the tasks that must be performed and the dependencies among these tasks. Process modeling is accomplished using MARVEL [16].

Thus, in Provence there is a component (Yeast) which is in charge of observing all the events occurring in the business process, and mapping them to specific process steps. This monitor is implemented by trapping the invocations of operating system services. It is thus possible to observe even low-level events occurring in the business process (e.g., file system operations). As an example, if a file named `main.c` is

	SPADE	SENTINEL	Provence
Completeness	Explicit modeling	Explicit/Implicit modeling	Explicit modeling
Coherence	Control integration of tools	Basic tool integration	OS services trapping

Table 3.1: A preliminary comparison of SPADE, SENTINEL, and Provence.

opened, written, and finally closed, using any editing tool, the monitor can reasonably deduce that the modification of a source code has occurred.

Completeness and coherence in Provence

The ability of Provence to monitor system events has a great impact on environment coherence. Provence is able to detect a large number of events, since it can observe the operations accomplished at the operating system level. This means that even standard Unix tools can be monitored, and their effects controlled.

As for completeness, Provence does not differ significantly from SPADE and most PSEEs (indeed, Provence is based on MARVEL). Each specific situation has to be explicitly anticipated and described in the process model as a set of rules. Moreover (and obviously), Yeast must be explicitly given the set of rules that have to be used to map the sequences of events triggered by the monitor to the actual process steps.

Therefore, the Provence approach can be summarized as follows:

Completeness Being based on MARVEL, Provence uses an *extensive* description of any process-related operation. Thus completeness can be enhanced only by explicitly enriching the process model. Again, this approach is similar to SPADE and to most existing PSEEs.

Coherence The adoption of a monitor makes it possible to *trigger a large number of events at the operating system level*. Coherence can therefore be guaranteed even when users adopt tools that are not control-integrated within the PSEE.

3.6 Conclusions

The informal application of the proposed framework to the above PSSs makes it possible to draw some preliminary conclusions (summarized in Table 3.1).

1. In order to enhance completeness, it is necessary to enrich the semantics of the process modeling language, to facilitate the representation of a larger number of states and transitions. The advantage of the SENTINEL approach is based on its ability of implicitly representing a number of states and transitions that are automatically derived by relaxing the process model constraints expressed by the preconditions of the exported transitions.

2. Coherence can be improved by enriching the architecture of the process support system with mechanisms that trigger all the events occurring in the business

process and map them onto the process model being enacted. SPADE operates at the tool control-integration level, by monitoring the messages being exchanged. Provence operates at the operating system level, by monitoring the OS services being invoked.

This preliminary comparison indicates that *dealing with completeness is mainly a linguistic issue while dealing with coherence is mainly an architectural issue*.

This conclusion motivates the approach taken to develop PROSYT. From the language and enactment mechanism standpoint, it adopts an approach similar to the approach adopted by SENTINEL. A process model which describes the basic actions that can be performed and the constraints on this actions is enacted in a flexible way by relaxing the constraints it states, thus allowing users to deviate when needed to cope with unexpected situation. From the architecture standpoint, PROSYT adopts a new approach besed on a distributed, event-based, architecture which facilitates the integration of several tools into the environment in order to increment coherence.

Chapter 4

The PROSYT environment

“Problems cannot be solved at the same level of awareness that created them.”
Albert Einstein (1879-1955)

4.1 The PROSYT environment: an overview

Supporting users in facing unforeseen situations, controlling that nothing bad happens, and tracking users’ actions to help them during the reconciling activity is a difficult and challenging task, which requires advanced techniques to be carried out [33].

When an unforeseen situation arises, the PSS must provide its users with an adequate set of basic operations that allow the process to proceed. These operations have to be general enough to be used in solving all the problems that may result from the unforeseen situation but, at the same time, they have to be tailored to the specific process adopted. To satisfy these conflicting requirements, a prototype PSS called PROSYT has been designed and implemented, which adopts an original approach both for process modeling and for process enactment.

- As for process modeling, PROSYT adopts an *artifact-based* approach to describe business processes. Process models developed using the PROSYT PDL (called *PLAN*, which stands for Prosyt LANguage) are centered around a description of the artifacts produced during the process. Each artifact is characterized by a set of attributes whose values define its internal state, and a set of basic operations that may be invoked upon the artifact. Boolean expressions are used to express the conditions that have to hold in order for each operation to be allowed to start. By using these expressions, the process designer may impose a partial ordering among actions the that characterize the business process, thus describing the expected sequence of process activities.

  In our experience, the artifact-based approach guides process designers in identifying the best range of basic operations that can be used both during the standard execution of the business process and when unexpected situations arise. Moreover, this approach helps process designers in keeping the description of these operations separate from the description of the correct order of their invocation, which characterizes the particular process adopted.

This distinction between the operations and their ordering, is the basis to satisfy the above mentioned conflicting requirements on the right choice of operations to be proposed to the PSS's users when an unexpected situation arises. In fact, when an unforeseen situation arises, PROSYT users are allowed to change the flow of activities stated by the model by invoking artifacts' operations independently from the fact that the constraints that regulate their correct execution are satisfied. This means that when an unforeseen situation arises, PROSYT proposes to its users a list of operations composed of all the basic operations which may be invoked upon process artifacts. These operations are general enough to face unexpected situations and depend on the adopted business process.

On the other hand, allowing users to execute any operation described in the process model with no care of the constraints that regulate its execution may seem to be a very undisciplined approach. To have a more fine grained control on the deviations allowed, PLAN organizes the constraints on operations in different classes, depending on the type of condition they express. As an example, a class of constraints is used to describe the preconditions of operations and another class is used to control the agents that are allowed to invoke each operation. This helps in distinguishing among different classes of deviations from the model, in order to better control the progress of the process.

- As for process enactment, the approach adopted by PROSYT is similar to the approach adopted by SENTINEL: process constraints are automatically relaxed during enactment to increase state and transition completeness. PROSYT users are not forced to obey the constraints stated in the process model. Instead, they can invoke the artifacts' operations described in the process model even if the constraints that characterize them are not satisfied. PROSYT keeps track of the results of these deviations and controls that the overall invariants that characterize process correctness are not violated as a result of such deviations.

  In the area of process enactment the main difference between PROSYT and SENTINEL is that PROSYT may adopt different *deviation handling policies*. They state the level of enforcement adopted (i.e., the classes of constraints that can be violated during enactment) and the actions that have to be performed when users try to deviate from the model. The deviation handling policy may change at enactment-time and may vary from user to user. As an example, some actions may be allowed during some phases of the process while they may be disallowed during other, more critical, phases. Similarly, an expert user may be allowed to perform deviations which are forbidden to beginners.

- As for system architecture, in order to increase coherence, PROSYT models are enacted in an open run-time environment in which third-party tools may be easily integrated. This important feature has been obtained by adopting an event-based communication paradigm between PROSYT and the tools it needs to control. As explained in [24], the event-based communication paradigm increases dynamic reconfigurability of the system architecture and is perfectly suited to support an easy integration of existing tools, such as editors and other office-automation tools.

Another feature of PROSYT is the approach adopted to implement process enactment. While most of the currently available PSSs adopt an approach based on interpretation to enact process models, PROSYT adopts an approach based on compiling. A PLAN process model is first compiled to obtain a *process implementation*, which is executed during enactment.

The language adopted for process implementation is Java. Business process models written in PLAN are translated into Java and then compiled into Java byte code using the standard Java compiler. The standard Java interpreter, together with a set of Java classes, which provide the process designers with an API to implement process activities, plays the role of the process engine. This approach provides several advantages: the resulting run-time environment is freely available, light, scalable, and based on a standard, open technology. Moreover, it provides better performance with respect to a pure interpreted approach. Finally, PROSYT can benefit from the Java approach: "write once, run everywhere" to obtain portable process implementations whose components may be freely distributed across a heterogeneous network.

4.2 PLAN: the Prosyt LANguage

4.2.1 Artifact-based vs. activity-based PDLs

Most of the existing PDLs, including SPADE and SENTINEL, can be classified as activity-based. They support the same abstractions of standard, imperative programming languages. Business processes are modeled by adopting a top-down approach: the entire process is first described as a single activity, which is progressively refined into a collection of sub-activities. This process of refinement ends when the basic operations that compose the business process are identified. Artifacts produced during the process are described as inputs and outputs of activities.

This approach has proven to be effective in describing simple, well standardized, repetitive processes, characterized by a simple flow of control, possibly composed of a sequence of operations. Conversely, this approach presents some problems when it is used to model complex, highly dynamic, business processes characterized by a complex flow of control, with a large number of users choices. Moreover, the activity-based approach does not help process designers in correctly separating the description of the basic operations that compose the process from the description of the flow of control among these operations. It is usually the latter that needs to be changed to cope with unexpected situations.

Conversely, PLAN is an artifact-based PDL. In PLAN, the activity of business process modeling focuses on the description of the artifacts produced during the process. The basic activities that compose a business process are described as operations which may be invoked upon artifacts. This approach is similar to the one adopted by object-oriented programming languages. In fact, PLAN can also be classified as an object-oriented PDL. It supports encapsulation, classification, information hiding, inheritance, and polymorphism.

To understand the advantages of the artifact-based approach against the activity-based approach, in particular when unexpected situations arise, it is important to observe that usually the operations associated with artifacts are fairly basic (e.g., edit, compile, deliver) and depend only marginally on the particular process adopted (at least when a single business domain is taken into consideration). What characterizes the particular process adopted by a business organization is the partial ordering among these operations. Given these premises and remembering the fact that in an artifact-based PDL, like PLAN, there is a clear distinction between the operations that can be invoked upon artifacts and the constraints that characterize their ordering, it is easy to understand why an artifact-based approach is ideal to support users both during standard operations and during deviations.

4.2.2 PLAN: an overview

A PLAN model describes a *project type*, a set of *repository types*, a set of *folder types*, and a set of *artifact types*. They give a static description of different entities, namely, the overall project, the project repositories, the folders contained in such repositories, and the artifacts produced during the process, respectively. The overall structure of these types is similar: they describe the internal structure, the behavior, and the constraints which characterize their instances. They are similar to classes in an object-oriented language.

A PLAN model starts with the description of a project type. It describes a particular business process, as an example the process of developing a new software product. It defines the set of *groups* involved in the process (each user belong to one or more groups), the set of *repositories* that hold the artifacts produced during the process, and a set of operations which describe global activities like adding a new user to the project or starting the project.

Each repository is an instance of some repository type, which describes the set of attributes that characterize its internal state, the set of operations that can be invoked by users upon it, and the constraints that describe the conditions that have to hold for the repository state to be considered correct.

At enactment-time each repository contains a set of folders, which, in their turn, contain other folders and artifacts. Each folder is an instance of some folder type, which statically describes its internal structure (i.e., the set of its attributes), its behavior (i.e., the set of operations that may be invoked upon it), and the constraints that regulate its behavior. Similarly, each artifact is an instance of some artifact type which describes its internal state and behavior.

The sections which follow describe artifact types, folder types, repository types, and project types in detail.

4.2.3 Artifact types

As mentioned in Section 4.2.1, a PLAN description of a business process is centered around the description of the artifacts that are produced during the process. Each artifact belongs to a particular *artifact type*. All the artifacts that belong to the same artifact type are characterized by the same internal structure and by the same external behavior, like the objects of the same class in an object-oriented programming language.

Moreover, each artifact has a name and an icon. They are used to identify the artifact when the set of all the artifacts that compose a business process is browsed through the PROSYT graphical user interface. There is no semantics associated to artifact names and icons.

In PLAN, each artifact type is described by a 6-tuple of sequences:

$$\langle attributes, states, exported\ op., automatic\ op., methods, invariants\rangle$$

Attributes. They determine the internal structure of the artifacts belonging to this type. Each attribute is defined by using the same syntax adopted by Java for object attributes. At enactment-time, attribute values determine the *concrete state* of artifacts.

States. This is a sequence (i.e., an ordered set) of *logical state* definitions. Each logical state is defined by a tuple: $\langle identifier, predicate\rangle$. The identifier defines the symbolic name of the logical state. The predicate is a boolean expression on the values of the attributes, which specifies the conditions that characterize the state.

In order to simplify the modeling of logical states, the predicates that define different logical states of the same artifact are not required to be mutually exclusive. At enactment-time, the first state definition whose predicate is satisfied is chosen and the artifact enters the logical state whose identifier is associated with the selected state definition[1]. If none of the predicates is satisfied, the logical state of the artifact is *partially inconsistent*.

Exported op. These are the operations that may be invoked by the human agents upon artifacts. They represent the basic activities that compose the business process. Each exported operation is characterized by a 5-tuple:

$$\langle name, formal\ parameters, agents, guard, precondition, body\rangle$$

- The name and formal parameters have the usual semantics of method names and parameters in Java. The only peculiarity is that the type of formal parameters is left unspecified. In fact, exported operations are invoked by users who interact with the system through a graphical interface that allows them to pass only strings at invocation time.
- "Agents" is an expression used to determine the agents allowed to invoke the operation. It is composed of a list of users and/or groups.
- The guard is composed of a sequence of *guard entries* which, in turn, are composed of a tuple: $\langle identifier, predicate\rangle$. The identifier must be one of the names used for the artifact's logical states. The predicate is a boolean expression on the values of the artifact's attributes and operation parameters.
- The precondition is also described as a boolean expression on the values of artifact's attributes and operation parameters. The difference between the guard and the precondition is that the latter describes a condition that

[1] Observe also that requiring the predicates which define the logical states of an artifact type to be mutually exclusive does not really make sense due to the fact that in PLAN this property cannot be verified statically.

does not depend on the logical state of the artifact. It must be verified in order for the exported operation to be allowed to execute independent of the state of the artifact. As a result, usually preconditions describe more critical requirements for the operations they belong to.

- Finally, the body is a sequence of Java statements that describe the actions performed when the exported operation is invoked. Any correct Java statement may be used. A set of predefined operations (i.e., methods) is provided to perform basic operations like launching a tool, creating a new artifact, and so on. The complete set of these operations can be found in Appendix A.

An exported operation E invoked upon an artifact A by one of the users allowed to invoke it is executed if (i) the logical state of A is one of the logical states mentioned in E's guard, (ii) the boolean expression associated to the corresponding guard entry is satisfied, and (iii) the E's precondition is also satisfied.

Automatic operations. These are operations that are not explicitly called by users but occur "automatically" as a result of an *event*. There are *predefined events* and *user defined events*. Predefined events are:

- creation of a new artifact (in such case the `initialize` operation is automatically called),
- invocation of exported operations of other artifacts,
- start of execution of exported operations of other artifacts, and
- end of execution of exported operations of other artifacts.

User defined events may be sent by tools integrated in PROSYT or may be explicitly signalled during the execution of operations (both exported and automatic) through the statement `sendEvent` (see Appendix A). The actions performed when an automatic operation executes are described in Java.

User defined event sent by tools or by using the `sendEvent` statement are described by a string in the form `"<name>(<arg_1>, ..., <arg_n>)"`. This event triggers the automatic operation: `on <name1>(<par_1>, ..., <par_n>)` if `<name1>` equals `<name>` and either `<par_i>` equals `<arg_i>`, or `<par_i>` is a *free parameter* (described in PLAN by using a single capital letter). When an event E triggers an automatic operations, the operation's body is executed with all the operation's free parameters, if any, bound to the corresponding arguments of E. As an example, the event represented by the string: `"delivered(module1, 11-6-1997)"` triggers the exported operation: `on delivered(X,Y)`. As a result of the match between the event and the exported operation, the free parameter `X` is bound to the string `"module1"` and the free parameter `Y` is bound to the string `"11-6-1997"`. These parameters can be used in the body of the exported operation.

Exported operations do not require neither guards nor preconditions because they cannot be invoked by users and consequently their execution does not require to be controlled. They are used to automate the process.

Methods. These are procedures local to the artifact where they are defined[2]. They are mainly used to simplify the implementation of the artifact's exported and automatic operations.

Invariants. These are boolean expression that describe the overall constraints that have to be satisfied by the concrete state of the artifact. Each invariant has an associated label chosen by the process designer and used to identify the type of invariant (e.g., the label `strong` may be used for strong invariants while the label `weak` may be used for weak invariants). The same label cannot be used for different invariants of the same artifact type (i.e., given an artifact type, a label uniquely identifies an invariant). Labels are user-defined (there is no set of pre-defined labels). By using labels, the process designer is able of defining as many type of invariant as needed.

As it will be described in Section 4.4, at enactment-time different actions may be associated with the violation of different types of invariants. As an example, violating invariants of type `weak` may result in sending a mail to the process manager, while violating invariants of type `strong` may result in stopping enactment.

When an invariant of type I is violated the corresponding artifact is said to be *I inconsistent* (e.g., weak inconsistent or strong inconsistent).

Figure 4.1 shows the PLAN description of the artifact type `TextDocument` that describes the class of text documents. As mentioned in Section 2.2.1, each text document has an owner and a manager. The manager is responsible for the document quality and may choose the owner (by invoking the exported operation `changeOwner`) who is in charge of editing the article. During initialization of a new text document (operation `initialize`) the artifact's icon (used by the "projectBrowser"; i.e., by the PROSYT's user interface) is set and the value of the `manager` and `owner` attributes are initialized. Moreover, a file having the same name of the artifact and the extension "`tex`" is associated with the text document under the logical name "source". This name may be used to refer to the file when a tool operating on the document is called (as an example, see the body of the exported operation `edit`).

A text document may be in one of two states: `readyToBeEdited` or `edited`, depending on the value of the boolean attribute `edited`. It exports three operations: `changeOwner`, `show` and `edit`. They may be called by the document manager, by the document owner, and by any author, respectively.

The finite-state machine in Figure 4.2 formalizes the behavior of text documents, which is determined by the guards of its exported operations.

4.2.4 Inheritance in PLAN

As mentioned, PLAN is an object-oriented language that supports single inheritance. The semantics of inheritance in PLAN is the following. Let S be an artifact type inheriting from the artifact type P; then the following conditions hold:

- P's attributes are also S's attributes. S may add new attributes.

[2] They are also visible to the artifacts that inherit from the artifact where they are defined. They are similar to Java "protected" methods.

```
artifactType TextDocument {
attributes:
  String owner, manager;
  boolean edited=false;
states:
  readyToBeEdited : !edited;
  edited : edited;
exportedOperations:
  changeOwner(newOwner) to users manager groups ProjectLeader;
  {owner=newOwner;}
  edit() to users owner;
  {
    String[] args=new String[1]; args[0]="{source}";
    callUserTool("notepad", args);
    edited=true;
  }
  show() to groups Authors;
  guard edited;
  {
    String[] args=new String[1]; args[0]="{source}";
    callUserTool("less", args);
  }
automaticOperations:
  initialize(Object[] args) {
    setIconFileName("TextDocument.gif");
    manager=(String) args[0]; owner=manager;
    associateFile("source",getName()+".tex");
  }
}
```

Figure 4.1: The artifact type `TextDocument`.

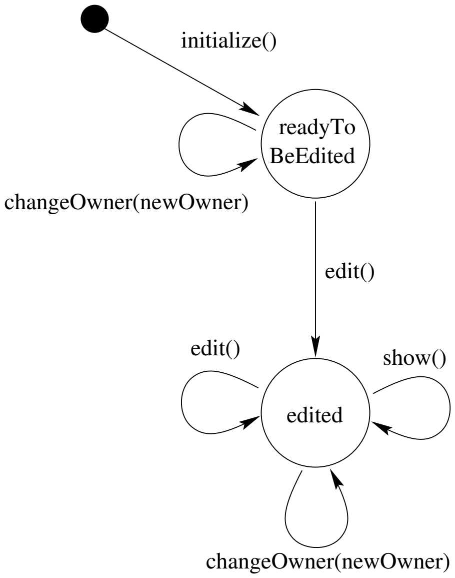

Figure 4.2: The FSM which describes the behavior of text documents.

- P's states are also S's states. S may redefine P's states (using the `redefine` statement) and may add new states. If S adds new states then their definition conceptually precedes the definition of the states inherited from P. This, combined with the fact that the logical state of an artifact is chosen by evaluating the sequence of state definitions in order until a state definition whose predicate is satisfied is found, makes it possible to add sub-states of existing states. As an example, suppose P defines the states:

  ```
  s1 : a && b;
  s2 : a;
  s3 : !a;
  ```

  where `a` and `b` are two boolean attributes, and suppose also that S defines a new boolean attribute `c`, redefines the state `s1`, and adds a new state `s4` as follows:

  ```
  redefine s1 : a && !b;
  s4 : a && !b && c;
  ```

  The resulting definition of S's states is equivalent to having defined them as:

  ```
  s4 : a && !b && c;
  s1 : a && !b;
  s2 : a;
  s3 : !a;
  ```

  This means that if `a` is true, `b` is false, and `c` is true, then S's state is `s4`. If `a` is true and both `b` and `c` are false, then S's state is `s1`. If both `a` and `b` are true, then S's state is `s2`, independent of the value of `c`. Finally, if `a` is false, then S's state is `s3` independent of the values of `b` and `c`.

- P's exported operations, automatic operations, and methods belong also to S. S may add new exported operations, automatic operations, and methods and may redefine inherited ones. In redefining an exported operation E inherited from P, S can use the following statements:

  – `super.to`. It may be used to refer to the original list of users that can legally invoke the operation as defined in P.

  – `super.guard`. It may be used in the guard of the redefined operation to refer to the original guard of E. This may be useful if the redefined version of E needs to add new guard entries to the list of guard entries defined in P.

  – `super.predicate`. It may be used in the predicate part of E's guard entries to refer to the predicate defined in P for the corresponding guard entries. This may be useful if the redefined version of E needs to change some of the predicates of the E's guard entries defined in P.

  – `super.precondition`. It may be used in the precondition of E to refer to the original precondition of E. This may be useful if the redefined version of E needs to add some conditions to the original precondition of E.

```
artifactType Section extends TextDocument {
attributes:
  MasterDocument master;
  boolean delivered=false;
states:
  delivered : delivered;
exportedOperations:
  changeOwner(newOwner) to super.to;
  precondition !delivered;
  show() to super.to;
  guard super.guard, delivered;
  deliver() to users owner, manager;
  guard edited;
  {owner=manager; delivered=true;}
automaticOperations:
  initialize(Object[] args)
  {super.initialize(args); master=(MasterDocument) args[1];}
  on starting review(X)->master
  {owner=X;}
  on starting endOfReview()->master
  {owner="";}
}
```

Figure 4.3: The artifact type `Section`, a sub-type of `TextDocument`.

Finally, the pseudo-variable `super` keeps the usual Java semantics. It may be used in the operations' body to explicitly refer to the body of the corresponding operations inherited from P. It is important to observe that if the body of the operations is absent, the original body is used (i.e, the body defined in P).

- P's invariants apply also to S. S may redefine them or may add new ones. In redefining inherited invariants, S may refer to the original (i.e., inherited) version through the pseudo-variable `super`. As an example, S could strengthen an invariant `strong` as follows:

```
strong : super.strong && c==4;
```

Figure 4.3 shows the PLAN definition of the artifact type `Section`, a sub-type of the artifact type `TextDocument` shown in Figure 4.1. The finite-state machine that formalizes the behavior of this class of artifacts is shown in Figure 4.4

4.2.5 Folder, repository, and project types

One of the major drawbacks of artifact-based PDLs is the difficulty of describing activities and invariants that refer to a collection of artifacts. To describe such kind of activities and invariants, PLAN provides the concepts of *repository* and *folder*. Each repository is an instance of some *repository type* and contains a set of *folders* organized in a tree structure. In turn, each of these folders (instance of some *folder type*) is a container of artifacts and other folders. A folder cannot exist alone; it has to belong to some repository (i.e., to the tree of folders that belongs to the repository). Repositories are analogous to partitions of a file system, while folders are analogous to directories.

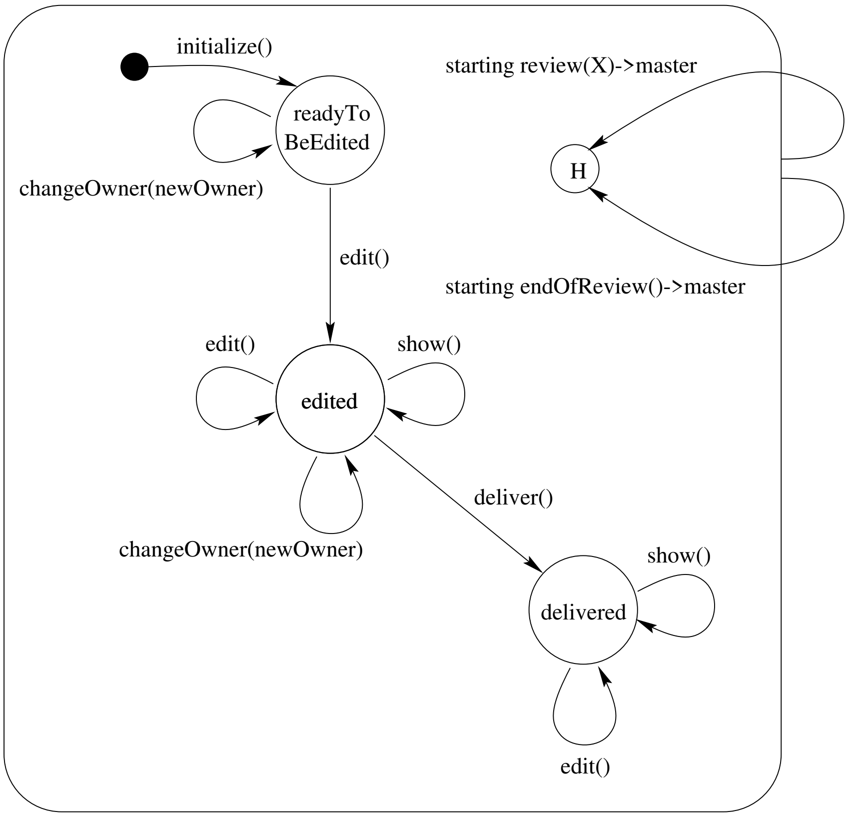

Figure 4.4: The FSM which describes the behavior of sections.

Attributes, states, exported operations, automatic operations, methods, and invariants may be associated either to repository types or to folder types using a syntax similar to that adopted for artifacts. In particular, exported operations and invariants may be used to describe business activities and invariants which refer to structured collections of artifacts.

To help process designers in describing conditions that refer to a collection of artifacts, both repository types and folder types provide a set of predefined predicates that can be used to query the logical state of the artifacts and folders they contain at enactment-time. As an example, predicate `allArtifacts(<artifact type>, <state>)` is used to evaluate the logical state of a set of artifacts (of the same type) belonging to a given folder. It returns true if all the artifacts that belong to the folder (or repository) whose type is "`<artifact type>`" are in the logical state "`<state>`". The complete set of these predicates is given in Appendix A.

A difference between folders and repositories is that folders can be moved while repositories cannot. As described in Section 4.3, repositories are associated to a particular host and cannot be moved at run-time.

The last concept present in PLAN is the concept of *project type*. Each business process is described in PLAN as an instance of some project type. It is characterized by a statically defined set of repositories, by a set of *groups*, and by a set of exported operations, automatic operations, and invariants, which refer to the entire process. Each user belongs to one or more groups. Users may be added to or removed from groups by calling the predefined methods `addUserToGroup` and `removeUserFromGroup` from any of the project's operations.

Figure 4.5 models the activity of writing a paper described in Section 2.2.1 using PLAN. The project type that describes such process, together with the repository and folder types needed to model the process are defined there. Figure 4.6 shows the PLAN definition of the artifact type `MasterDocument` also used to model the activity of writing a paper, while the artifact types `Section` and `TextDocument` were already defined (see Figures 4.3 and 4.1, respectively).

The project type `PaperWriting` defines two groups (`ProjectLeader` and `Authors`), a repository of type `PaperRep` that holds the paper to be written, and an operation `addAuthor` that may be called by the project leader to add a new author to the team that is in charge of writing the paper. The automatic operation `initialize`, called when a new instance of the repository type `PaperRep` is created, uses the procedure `enableGroup` (which is part of the PROSYT API described in Appendix A) to make the repository visible to the project leader and to all the authors.

The exported operation `createPaper` can be invoked upon the paper repository to start the process. When it is invoked, two folders are instantiated: one of type `PaperFolder`, used to hold the sections that compose the paper, and the other of type `PicturesFolder` used to hold the pictures that have to be inserted into the paper.

When the paper folder is created a new artifact of type `MasterDocument` is added to the folder (see the automatic operation `initialize` of the folder type `PaperFolder`). It represents the master document described in Section 2.2.1. Several exported operations may be invoked upon the master document. They are used to automate all the basic activities that compose the process of writing a paper. In particular, it is possible to add new sections, edit the master document, show its content,

```
projectType PaperWriting {
states:
  ready : true;
groups:
  ProjectLeader, Authors;
repositories:
  PaperRep mainRep;
exportedOperations:
  addAuthor(name, passwd) to groups ProjectLeader;
  {addUser(name,passwd); addUserToGroup(name,"Authors");}
}

repositoryType PaperRep {
attributes:
  boolean paperFolderCreated=false;
states:
  readyToCreatePaper : !paperFolderCreated;
  readyToStartEditingPhase : paperFolderCreated;
exportedOperations:
  createPaper(name) to groups ProjectLeader;
  guard readyToCreatePaper;
  {
    createNewFolder("PaperFolder", name);
    createNewFolder("PictureFolder", name+".pict");
    paperFolderCreated=true;
  }
automaticOperations:
  initialize()
  {enableGroup("ProjectLeader"); enableGroup("Authors");}
}

folderType PaperFolder {
attributes:
  MasterDocument master;
states:
  ready : true;
automaticOperations:
  initialize() {
    master=createNewArtifact("MasterDocument", getName());
    enableGroup("ProjectLeader"); enableGroup("Authors");
  }
invariants:
  strong : allArtifacts("Section","delivered") ||
           !state(master,"delivered");
}

folderType PicturesFolder {
states:
  ready : true;
automaticOperations:
  initialize() {
    setIconFileName("PicturesFolder.gif");
    enableGroup("ProjectLeader"); enableGroup("Authors");
  }
}
```

Figure 4.5: The PaperWriting process model.

```
artifactType MasterDocument {
attributes:
  boolean edited=false, compiled=false, delivered=false;
  String revisor=""; Hashtable sections=new Hashtable();
states:
  readyToEdit : !edited;
  readyToCompile : edited && !compiled;
  readyToPrint : edited && compiled && !delivered;
  delivered : delivered;
exportedOperations:
  review(revisorName) to groups ProjectLeader;
  guard readyToPrint and revisor.equals("") && allSectionsDelivered();
  {revisor=revisorName;}
  endOfReview() to users revisor;
  guard readyToPrint;
  {revisor="";}
  deliver() to groups ProjectLeader;
  guard readyToPrint and revisor.equals("") && allSectionsDelivered();
  {delivered=true;}
  addSection(sectionName, manager) to groups ProjectLeader;
  {
    Object[] o=new Object[2]; o[0]=manager; o[1]=this;
    sections.put(createNewArtifact("Section", sectionName, o),
                 "not delivered");
  }
  edit() to groups ProjectLeader users revisor;
  precondition !delivered;
  {
    String[] args=new String[1]; args[0]="{source}";
    callUserTool("notepad", args); edited=true; compiled=false;
  }
  compile() to groups Authors;
  guard readyToCompile, readyToPrint;
  {
    String[] args=new String[1]; args[0]="{source}";
    callTool("latex", args);
    args=new String[3]; args[0]=getName()+".dvi"; args[1]="-o";
    args[2]=getName()+".ps"; callTool("dvips", args);
    if(!compiled)
      {compiled=true; associateFile("postscript",getName()+".ps");}
  }
  show() to groups Authors;
  guard readyToPrint, delivered;
  {
    String[] args=new String[1]; args[0]="{postscript}";
    callUserTool("ghostview", args);
  }
  print(printerName="LaserHPSE") to groups Authors;
  guard readyToPrint, delivered;
  {
    String[] args=new String[2]; args[0]="-P"+printerName;
    args[1]="{postscript}"; callTool("lpr", args);
  }
automaticOperations:
  initialize() {
    setIconFileName("TextDocument.gif");
    associateFile("source",getName()+".tex");
  }
  on starting deliver()->X
  {if(sections.containsKey(X)) sections.put(X,"delivered");}
  on starting edit()->X
  {if(sections.containsKey(X)) compiled=false;}
methods:
  boolean allSectionsDelivered() {
    boolean res=true;
    for(Enumeration e=sections.elements(); e.hasMoreElements(); ) {
      res = res && e.nextElement().equals("delivered");
    }
    return res;
  }
}
```

Figure 4.6: The artifact type `MasterDocument`.

compile the paper (i.e., generate the postscript file of the entire paper from the LaTeX sources), print the paper, start and end the review phase. The guards and preconditions of these operations are used to guarantee that the order of their execution satisfies the constraints which characterize the expected process defined in Section 2.2.1.

More detail on this model will be given in Section 4.4.4. There the approach taken by PROSYT to support authors in performing the deviations from the process model needed to cope with unexpected situations is described.

4.2.6 Process correctness

A PLAN model is *correct* if every state reachable by executing it without violating the constraints that characterize its exported operations satisfies the invariants of every object created during execution. This can be proved inductively by showing that:

- The initial state of each repository, folder, and artifact satisfies the invariants.
- For each exported operation, if it is invoked by an authorized user, in a state that satisfies the invariants, its guard, and its precondition, then the process constraints are satisfied upon completion[3].

Due to the complexity of the PLAN language, it is not possible to check process correctness automatically. It is the responsibility of the process designer to check model correctness. If an incorrect process model is enacted it may happens that an invariant is violated even if no deviation has been performed (i.e., even if all the exported operations were called by obeying to the process constraints). As it will be better describe later, in PROSYT this is a minor problem since it is possible to continue process enactment even if some of the invariants are violated.

4.3 Process model enactment in PROSYT

This section describes the PROSYT enactment mechanism and the PROSYT run-time architecture. The focus is on the general aspects of PROSYT. The aspects directly related to the mechanisms to tolerate deviations and support users in facing unforeseen situations will be detailed in Section 4.4.

As mentioned in Section 4.1, a process model written in PLAN is enacted by first translating it into Java. As a consequence, the activity of process model writing and enacting in PROSYT (i.e., the PROSYT meta-process) consists of the steps described in Figure 4.7. The business process modeled in PLAN is translated into Java by invoking the PROSYT compiler. This translation step creates a set of Java source files together with a set of script files that can be used to start process enactment. Java source files are than compiled by using the standard Java compiler. As a result of this operation a set of Java class files is obtained. They compose the *process implementation* that is executed during the process enactment step. The next step is process deployment. During this step, the process implementation is tailored to the specific organization that is in charge of carrying out the process. In particular, each repository is associated with a particular

[3]Exported operations are viewed as atomic entities. States entered while an operation is being executed are considered invisible.

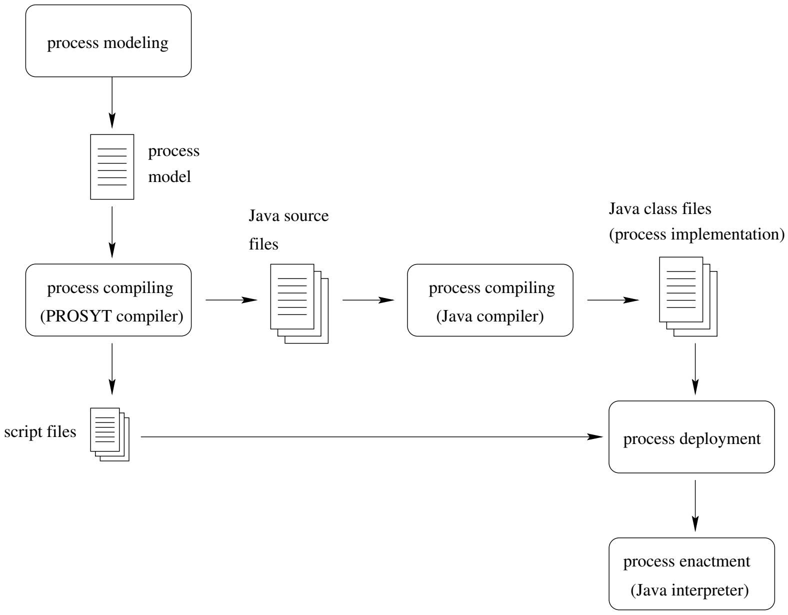

Figure 4.7: The PROSYT meta-process.

host, the default deviation handling policy is chosen (see Section 4.4), and the information on the starting set of users participating to the process (their login, password, and the groups they belong to) are added. Finally, process enactment is started by executing the script files generated by the PROSYT compiler.

4.3.1 An event-based architecture supporting process model enactment

The run-time architecture of PROSYT uses an event-based communication paradigm. It is composed of a set of active, distributed components, which interact by delivering and receiving events. Events are a particular type of message. While conventional messages are sent from a source to one or more recipients, as specified by the source itself, events do not include any information about their recipients. An event is generated by a component and it is automatically notified to all the components that declared their interest in receiving it.

The main advantage of using an event-based communication style is that the resulting system may be easily reconfigured. New components may be added, existing components may be removed, and components may be moved from a host to another without impact on the remaining components. This is very important for PROSYT because the components that constitute the PROSYT environment are the artifacts, folders, repositories, and the project defined in the process model being enacted, whose number and location changes at enactment time. As an example, new artifacts may be created and existing artifacts may be moved from a repository to another (i.e., from a host to another).

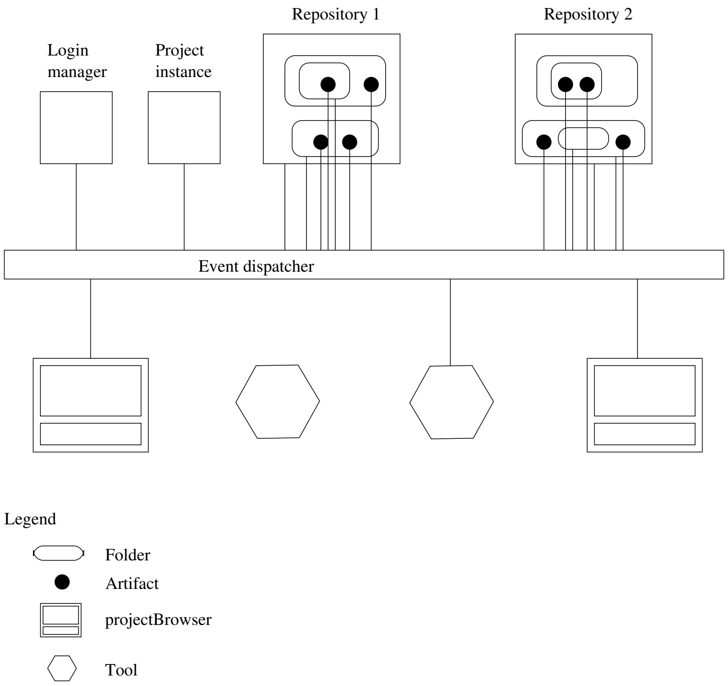

Figure 4.8: The PROSYT run-time architecture.

To support event delivering PROSYT takes benefit of the services provided by the JEDI distributed infrastructure [24]. The resulting PROSYT run-time architecture is described in Figure 4.8. It is composed of the following distributed, active components:

- A login manager, which is in charge of controlling login and logout of users. It manages all the information about the human agents that are involved in the process. In particular, it records their names, their login, their password, and the groups they belong to. This information may change at run-time as a result of enacting the process model.

- A set of repositories. The number of these repositories, their names, and their type are statically determined by the process model being enacted. Each repository is associated with a particular host chosen during the deployment step.

- A set of folders. Each folder belongs to a repository. Folders may move from a repository to another during process model enactment. If a folder moves from a repository α to a repository β then all the folders and artifacts it contains are moved from α to β.

  Because different repositories are usually associated with different hosts, moving a folder from a repository to another means moving a running process from an host to another. This is possible obtained by taking advantage of the features provided by the JEDI infrastructure to support the implementation of mobile applications [28, 29].

- A set of artifacts. Each artifact belongs to a folder. Artifacts may move from a folder to another (belonging to the same repository or not) during enactment (also artifacts like folders are capable of moving from an host to another when needed).

- A project instance, whose behavior is determined by the process model being enacted (it is an instance of the project type that constitutes the process being enacted).

- A set of "projectBrowser" instances, one for each user logged in. They are the user interface to the system. Using the projectBrowser application, users may invoke the exported operations provided by the project, repositories, folders, and artifacts that compose the process. Figure 4.9 shows the projectBrowser in action during enactment of the `PaperWriting` project described in Section 4.2.

- A set of other tools which may be invoked to accomplish the operations that compose the process. Some of these tools are integrated in the PROSYT environment and are able of receiving and delivering events; others are not. The execution of the former may be directly managed by PROSYT, while only starting and ending of the latter can be controlled by PROSYT.

- An event dispatcher which is in charge of delivering events to the interested components. As mentioned, the event dispatcher is part of the JEDI infrastructure.

Each of the above components is active, that is, each of them hold its own thread of control. In the current implementation of the PROSYT environment there is a different

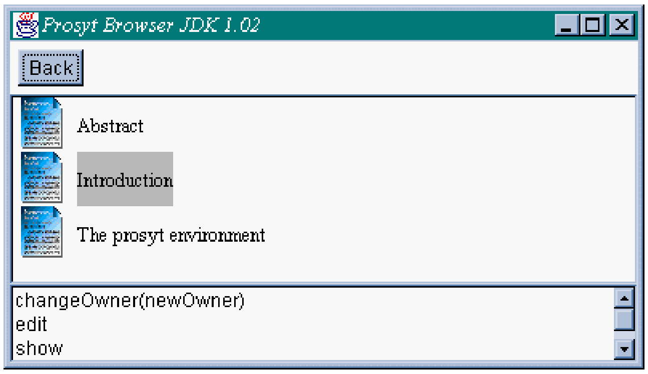

Figure 4.9: The projectBrowser during enactment of the `PaperWriting` project.

operating system process for each repository, one for each project browser instance, and one for each tool. Another process exists for the login manager, and another one for the project instance, while folders and artifacts are Java threads running into the operating system process of the repository they belong to.

When a user invokes an exported operation E upon an object (i.e., an artifact, a folder, a repository, or a project) O using the projectBrowser, a new event ev_E is signalled which describes such invocation. The event dispatcher delivers this event to all the components interested in receiving it. In particular, the event is dispatched to the object O and to all the other objects that have an automatic operation associated with the event ev_E. If the operation E is allowed to execute, a new event describing the starting of the operation is signalled and dispatched to the interested components and E's body is executed. Finally, when execution of E ends an event signalling this fact (i.e., the ending of E's execution) is dispatched to the interested components.

As mentioned, the JEDI event-based infrastructure supports transparent migration of running components from a host to another. Taking advantage of this feature, PROSYT allows both folders and artifacts to move from a repository to another (by calling the `moveTo` procedure which is part of the PROSYT API). This may be useful to reduce network traffic (related artifacts which exchange a lot of events among them can be moved closer) and to balance the computational loading of different hosts. Moreover, this feature is currently being investigated in order to support nomadic users.

4.4 Tolerating deviations in PROSYT

In order to support users even in presence of unforeseen situations, PROSYT adopts a special approach to process model enactment. During process model enactment, users are allowed to execute the exported operations described in the model even if the constraints that regulate their correct execution (i.e., the constraint on the users allowed to invoke them, their guard, and their precondition) are violated. By so doing, PROSYT allows users to deviate from the process model, changing the order of execution of operations or violating other, process specific, constraints.

Deciding exactly what kind of deviations to support and how to support them are very crucial design decisions. The freedom to deviate is fundamental to cope with unforeseen situations without the need for escaping from the PSS control but, at the same time, it has to be controlled in some way. A PSS that allows its users to perform any action in an uncontrolled way is not useful at all.

To control the type of deviations allowed and to manage the inconsistencies resulting from these deviations, PROSYT provides two different features: (1) the possibility of changing the *deviation handling policy* at enactment-time and (2) the possibility of using invariants to control the results of deviations and of changing the corresponding *consistency checking policy* at enactment-time. The following sub-sections describe these features in detail.

4.4.1 Deviation handling policy

Four types of deviations are defined in PROSYT:

User deviation. It is the result of executing an exported operation even if the user who invoked it was not allowed to do so.

Condition deviation. It is the result of executing an exported operation E upon an object O when the current state of O is one of the states listed in the E's guard but the corresponding predicate is not satisfied.

State deviation. It is the result of executing an exported operation E upon an object O when the list of states occurring in E's guard does not include O's current state.

Precondition deviation. It is the result of executing an exported operation whose precondition is not satisfied

In general, an user deviation is less critic than a condition deviation, which, in turn is less critic than a state deviation, which, in turn, is less critic than a precondition deviation. Obviously, this is not always true and it depends on the process being modeled. However, the form of guards and preconditions tends to force process designers to introduce the more critical requirements in preconditions and the less critical requirements in guards. In fact, preconditions have to be satisfied independent from the state of the object (i.e., artifact, folder, repository, or project) they belong, while guards depend on the object's state.

The deviation handling policy specifies the actions that have to be performed when a user invokes an exported operation whose execution could result in a deviation. For each deviation type and for each user one among five different actions may be specified:

Abort If this action is specified, the operation is not executed at all.

Inform user and abort In this case, the user who invoked the operation is informed of the fact that executing the operation results in a deviation and that, for this reason, the operation is aborted.

Ask user If this action is specified, the user is informed of the fact that executing the operation results in a deviation and he or she is prompted to confirm the invocation.

Inform user and continue The result of specifying this action is similar to the result obtained by specifying the "inform user and abort" action but, in this case, the operation is executed anyway.

Continue If this action is specified, the operation is executed without informing the user of the fact that he or she is deviating from the model.

The consequence of specifying a deviation handling policy that adopts the "abort" action for deviations of type T, for an user U, is that U is not allowed to perform such type of deviations. As an example, suppose U invokes an operation O upon an artifact A. Suppose also that the precondition of O is not satisfied and that the deviation handling policy for the user U specifies the action "abort" for precondition deviations. If this is the case, when U invokes the operation O using the projectBrowser tool, an event is generated to notify the artifact A that the operation O has been invoked but the artifact A simply ignores this event.

A similar behavior is obtained if the "inform user and abort" action is chosen. The only difference is that the user is informed of the fact that the action cannot be invoked (a dialog window with a diagnostic message appears). Conversely, the "ask user" action lets the user decide if he or she really wants to deviate. In the previous example, if the deviation handling policy for the user U specifies the action "ask user" for precondition deviations, when U invokes the operation O a dialog window appears informing him or her that is going to deviate from the model and asking for a confirmation. If the user confirm the operation is executed, otherwise it is not. Finally, the last two actions allow users to deviate from the process model without requiring any further confirmation.

It is easy to observe that the behavior of standard PSSs may be obtained by specifying the "abort" action for each type of deviation, for each user.

PROSYT allows the deviation handling policy to be changed at enactment-time by using an ad-hoc management tool. This tool allows the process manager to specify a different deviation handling policy for each user and to modify this policy dynamically to adapt it to the different situations encountered during the enactment of the process.

4.4.2 Consistency checking policy

The ability of choosing the types of deviation allowed on a per-user basis and the ability of changing this deviation handling policy at enactment-time, taking into account all the factors which may influence the level of enforcement that is more appropriate in each situation, offers a great control to the process manager, but another form of control may be required. In fact, by simply choosing a certain deviation handling policy, the process manager does not have any control on the results of a deviation. Executing the same operation under different conditions may result in different kind of deviations whose results may be greatly different. A mechanism is required to control that the state of the system does not become "incorrect" as a result of deviating from the process model. This mechanism exists in PROSYT and is based on the use of the invariants specified in the process model and on the possibility of specifying different consistency checking policies.

By specifying a set of invariants for each artifact, folder, repository, and project, the process designer defines the boundaries which distinguish the set of states of the system considered acceptable from the states considered incorrect. This fact can be

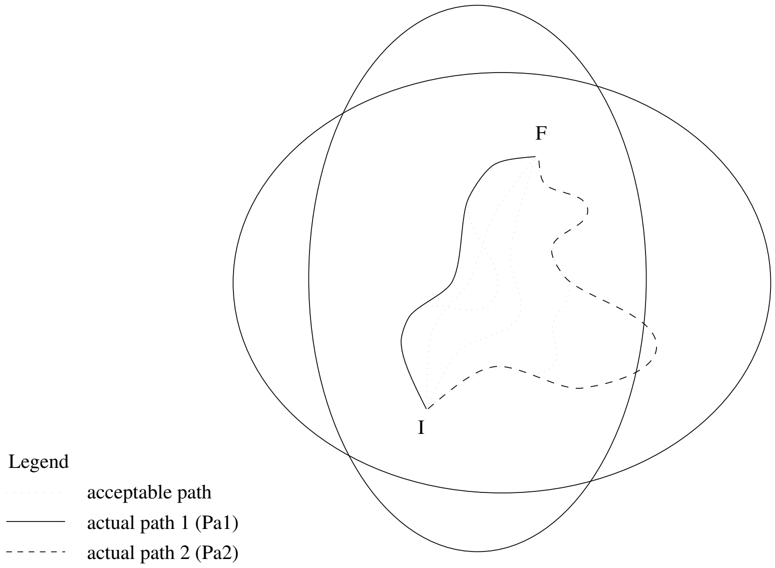

Figure 4.10: A graphical description of the constraint stated by a PLAN model.

graphically described as in Figure 4.10[4]. The figure describes the space of process states. A business process consists of moving from an initial state I to a final state F. A PLAN model describes:

- A set of acceptable paths to move from I to F. They are described by the set of basic actions (i.e., the exported operations) defined in the process model, and by the constraints that characterize them, which specify the correct order of their execution.
- One of more set of states, one for each type of invariant defined in the model. They are the states that satisfy the corresponding invariant.

As shown in Figure 4.10, the paths described by a correct PLAN model M pass only through states which belong to all the sets of states defined by M's invariants.

When a deviation occurs, the path P which describes the actual process diverges from any of the acceptable paths described by the model (as an example see paths $Pa1$ and $Pa2$ in Figure 4.10). Depending on the severity of the deviation, the path P may also cross the boundaries determined by the model invariants (e.g., path $Pa2$ in Figure 4.10). This means that P goes through states that violate one or more invariants. This suggests an approach to control critical deviations based on controlling the satisfaction of invariants during deviations. PROSYT adopts this approach by giving the process manager the ability of changing the consistency checking policy at enactment-time. It determines the action that the system has to perform when an invariant is violated. For each type of invariant a different action may be specified. One of the following predefined actions may be chosen:

[4]The figure provides an intuitive, informal description. It is meant to help the reader understand the concepts presented in this section by representing them graphically.

Stop enactment. Enactment of the entire process is stopped and the process manager is given the chance to start the reconciling activity. This is the safest action but it is also the action that mostly reduces the flexibility of the users who are not allowed to violate this type of invariants.

Stop object enactment. Enactment of the object whose invariant has been violated is stopped. This means that users are not allowed to invoke operations upon such object until the inconsistency between the model and the observed process is solved. The other objects which compose the process (i.e., artifacts, folders, and repositories) are not influenced by this action and their enactment is not stopped.

When this action is specified, only the inconsistent portion of the process is frozen until the consistency is not restored, the remainder of the process may advance. To solve inconsistency, the process manager may start the reconciling activity.

Inform process manager and continue. An e-mail is sent to the process manager to inform her or him that an invariant has been violated. The mail includes information regarding the deviation occurred. Process enactment continues. This and the next one are the actions that leave most freedom to the users which are allowed to violate this type of invariants with no impact on process enactment.

Continue. Invariant violation is ignored. In general adopting this action is very dangerous: the process manager does not have any mean to detect that the process has become inconsistent. On the other hand, in some situations it can be useful to adopt this consistency checking policy. As an example when the process manager discovers that an invariant is too restrictive but he or she does not want to change the process model (remember that PROSYT does not allow on-the-fly modification of the process model).

Observe that PROSYT re-evaluates invariants each time an operation completes its execution. As a consequence, it is able to track if the execution of an exported operation on an inconsistent artifact has moved it back to a consistent state. Obviously, in order for this behavior to be possible, the enactment of the artifact must continue even if an invariant has been violated (i.e., a consistency checking policy different from "stop enactment" and "stop object enactment" has to be chosen). This behavior constitutes the basis of a strategy to restore consistency described in the next section.

4.4.3 The reconciling activity

Independent of the consistency checking policy adopted, when part of the process becomes inconsistent due to a deviation, the process manager may need to restore consistency. The reconciling activity is the activity which restores consistency between the actual and observed process (which are supposed to be consistent) and the process model.

The approach taken by PROSYT to support users during the reconciling activity moves from the consideration that reconciliation is an activity which requires human intervention. It depends on a number of conditions that are not necessarily known to the PSS. They include the business process adopted, the reasons that lead the users to deviate from the expected sequence of activities, and the consequences of this deviation on the process actually performed. The PSS can only help the users in analyzing the deviation occurred in order to choose the best strategy to reconcile the process actually followed with the process model.

In general, two strategies exists to restore consistency between the actual process and the process model:

1. modify the process model by relaxing the constraints it states in order to include the state of the actual process into the set of consistent states or

2. perform some actions to move the actual process from the inconsistent to a consistent state.

Obviously a mixed approach may be also chosen.

In PROSYT the second strategy was adopted, which usually is less time consuming (changing the process model is often a hard task).

When an invariant is violated the process manager has several choices; such as:

- if the violation of the invariant does not stop enactment (i.e., either the "inform process manager and continue" or the "continue" consistency checking policies have been adopted) the process manager may let enactment continue expecting that the natural evolution of the process restores consistency;

- if the violation of the invariant stop enactment the process manager may decide to relax the consistency checking policy (adopting either the "inform process manager and continue" or the "continue" consistency checking policies) and restart enactment expecting that the process naturally evolves towards a consistent state;

- the process manager may directly access the artifacts, folders, repositories, and project objects, changing the values of their attributes in order to restore consistency[5]. These actions may be combined with actions performed out of the PROSYT control to also change the state of the actual process. The goal of these actions is to restore consistency of the actual process with the process model without breaking consistency between the actual and the observed processes (i.e., without going in an environment-level inconsistent state).

PROSYT supports the process manager during the reconciling activity by providing her or him with articulated information regarding the deviations that occurred. In particular, an history of the operations called by users upon artifacts from the first deviating action until the current time is presented to the process manager. This history may be organized in an artifact view (only the operations regarding a particular artifact are shown) or in a global view (all the operations are shown). In both views, for each operation, the time of invocation and the parameters passed at invocation time are shown.

By examining these information the process manager may have a clear view of the deviation happened and may decide the best approach to reconcile consistency.

[5]PROSYT provides the process manager with a tool to perform these changes.

4.4.4 An example showing the PROSYT ability in dealing with unexpected situations

As a first example of unexpected situation that may occur during the process of writing a paper and that requires a deviation from the model, imagine that an author A, different from the project leader, needs to start the process of writing a paper. This action is modeled by transition `BP_ACreatePaper` in Figure 3.1. To perform this action, the author A has to invoke the exported operation `createPaper` upon the paper repository. In principle, authors are not allowed to invoke this operation and the PSS should not execute it. PROSYT, however, allows this deviation to occur. In fact, provided that a deviation handling policy different from `abort` and `inform user and abort` is chosen for user deviations, PROSYT allows A to execute the operation `createPaper` in order to start the process as needed. This deviation is recorded by PROSYT and the information regarding it are presented to the process manager if necessary. In this case, giving users the chance to deviate from the model under their own responsibility, has solved in an efficient way a critical situation with a minimal impact on the overall correctness of the process.

Consider now the second example of unexpected situation described in Section 2.2.1. In order to start the review phase when some section has not been delivered yet, the project leader must invoke the operation `review` upon the master document even if the predicate `allSectionsDelivered` is false. If the deviation handling policy adopted for the project leader is liberal enough (i.e., if a policy different from `abort` and `inform user and abort` is chosen for condition deviations), the project leader is allowed to execute such operation, thus starting the reviewing phase. Observe that an adequate consistency checking policy (e.g., the `stop enactment` policy) prevents the reviewer from delivering the paper until the editing phase is concluded. In fact, delivering the paper (by executing the operation `deliver` upon the master document) when one of the sections has not been delivered yet, results in a violation of the master document invariant. If this happens, the enactment is stopped and the process manager is allowed to modify the process state (i.e., the values of the attributes of the artifacts, folders and repositories) or to relax the consistency handling policy before restarting process enactment. The information provided by PROSYT to make the right choice during the activity of reconciling is the history of the operations carried out by the users from the time when the first deviation occurred.

Chapter 5

Comparison

"Knowledge of what is does not open the door directly to what should be."
Albert Einstein (1879-1955)

This chapter takes advantage of the formal framework described in Chapter 3 to compare PROSYT with other existing PSSs.

5.1 PROSYT vs. SENTINEL

The SENTINEL PSEE has been described in Section 3.5.2. Here a detailed comparison of SENTINEL and PROSYT is performed, which highlights the differences that distinguish these two PSSs and explains why PROSYT is an improvement over SENTINEL's ability of supporting users in facing unforeseen situations.

PROSYT differs from SENTINEL in several ways:

- PROSYT uses an object-oriented, artifact-based, PDL that helps process designers in identifying the basic operations that compose the process model, whereas SENTINEL adopts an activity-based PDL (i.e., LATIN). This solves one of the major problems of SENTINEL: the difficulty of choosing the right granularity in describing process activities in order to provide the best support to users during deviations.
- PROSYT distinguishes among different types of constraints and invariants and allows the process manager to adopt the deviation and consistency handling policies more appropriate for the particular process being supported. Moreover, by introducing an arbitrary number of different classes of invariants, it allows a more flexible control of the results of a deviation.
- PROSYT allows the deviation and consistency handling policies to change during enactment and to be specified on a per-user basis. This allows the process manager to distinguish in a more precise way between the allowed and the forbidden deviations. The level of enforcement adopted may change from user to user and may depend on the phase of the process.

- PROSYT adopts an event-based distributed architecture that has three main advantages over the SENTINEL client-server architecture:

  1. process enactment can be distributed across different machines connected in a network. This reduces the load of each machine and increases the scalability of the entire system;
  2. components can be added or removed at run-time, and they can be moved from host to host. In other words, the architecture of the running system can be changed dynamically to follow the requirements of the process to be supported;
  3. existing tools may be easily integrated into the environment and this helps in achieving higher system coherence, which is a fundamental quality for PSSs.

- Finally, PROSYT is more efficient in capturing deviations and storing the information that are needed during the reconciling activity. In SENTINEL, a history of the entire process execution is stored and analyzed off-line to discover deviations. Conversely, PROSYT checks process constraints at run-time and stores a history of process execution starting from the time when first deviation occurs. This reduces memory usage and improves performances.

The combination of these differences results in a more flexible support to the business process modeled, which allows the process manager to better distinguish among different type of deviations and to better control the actions that have to be performed when a deviation occurs.

Formally, this means that given the same business process P, the process manager may take advantage of the PLAN characteristics to specify a model M whose enactment in PROSYT results in an environment more state and transition complete than the environment resulting from enacting a LATIN model of the same process P. Finally, the PROSYT architecture enhances environment coherence thus we can assert that usually PROSYT owns a view of the actual process that is more correct than the view owned by SENTINEL.

5.2 PROSYT vs. SPADE and Provence

SPADE and Provence have been described in details in Section 3.5.1 and 3.5.3, respectively. There, they have been compared with SENTINEL with respect to their ability in tolerating deviations from the process model. Besides some important differences, the overall approach adopted by PROSYT to tolerate such situations is similar to the approach adopted by SENTINEL. As a consequence, the considerations made in Section 3.5 when comparing SPADE and Provence with SENTINEL, may be easily adapted to PROSYT.

In particular, it is possible to say that PROSYT's enactment mechanism automatically increases the number of transitions of PSS with respect to the number of "transitions" (i.e., basic operations) explicitly defined in the process model, by "implicitly" including exported operations whose preconditions are not satisfied. Moreover, PROSYT increases the number of modeled states, by "implicitly" adding states which result from invoking an exported operation when some of the conditions that regulate its correct execution are not satisfied.

Conversely, SPADE and Provence do not differ significantly from most PSEEs with respect to state and transition completeness (indeed, Provence is based on MARVEL). Each specific situation has to be explicitly anticipated and described in the process model. In order to deviate from the expected sequence of activities the process model has to be changed and the new sequence of activities has to be explicitly modeled.

5.3 The OASIS environment

In the OASIS project an object-oriented, reflective framework for the definition, customization, and evolution of software process meta-models and the software process models that are their instances has been developed [61].

In developing this framework, the authors start from the consideration that every process modeling approach relies on some specific set of abstractions that defines a process meta-model. They observe that having the chance to model in a uniform way both the process model and the process meta model is essential to capture complex processes like software development processes and to manage the customization and evolution of process models and their meta-models.

This approach requires that both process models and process meta-models are made explicit. Such goal is pursued in OASIS by taking advantage of a set of kernel classes that can be used to describe both process models and process meta-models in a uniform, reflective way.

The process model and its meta-model are both described as specialization of kernel classes using an object-oriented definition toolkit. Classes at one level become object-instances at the upper level. This reflective approach is similar to the approach adopted by SPADE, and allows the meta-process (i.e., the process of modifying the process model itself) to be modeled as any other process. This is a very powerful approach to manage process model evolution and can be profitably used to cope with unexpected situations which require major changes on the process model. At the same time, as observed in Sections 2.2.2 and 3.5.1, this approach has the drawback that modifying the process model is a time consuming activity that usually requires the intervention of the process designer. As a consequence, this approach should be followed to cope with major deviations which will probably occur again in the future while the PROSYT approach is best suited to cope with minor deviations. The same consideration is valid also for Endeavors, the PSS described in the next section.

5.4 The Endeavors PSS

Endeavors [18] is an open, distributed, extensible process support system infrastructure, which provides mechanisms to support distribution of processes and users, integration of third party tools, customization and reuse of the process system components and process objects, and support for dynamic change of types and behaviors.

This section focuses on Endeavors' features supporting dynamic change of the process model because they can be used to cope with unexpected situations. In fact, the Endeavors' approach to the problem of facing unexpected situations is similar to the approach adopted by SPADE and OASIS: the process model must be changed in order to include the actions needed to cope with the new situation.

Endeavors allows the object-oriented definition and specialization of *activities*, *artifacts*, and *resources* associated with a software development process. The specification of processes and objects can be defined hierarchically; an activity can be further sub-divided into sub-activities; a software development team resource can be further defined to include team member or sub-team resources. Endeavors' *activity networks* define the inter-relationships between activities, artifacts, and resources as well as sub-networks. Networks include the definition of control flow, data flow, and resource assignments, and can be easily defined using a graphical network editor. Activity networks can be executed by an interpreter which sends events to particular objects, by triggering handlers which define the object's behavior.

One of the main features of Endeavors is its ability to allow an easy modification of the process model being executed. To support this, Endeavors allows dynamic modification of object fields, methods, and behaviors at runtime. In particular, object data and behaviors are kept separately and handlers are dynamically loaded and executed at the time a message is received, allowing behaviors to be easily exchanged during execution. Stakeholders can customize the abstraction levels for behavior and data that are appropriate for their site and their customization skill and authorization level. For example, technically sophisticated stakeholders may customize behaviors while non-technical people may be limited to simply setting the value of some fields (essentially, parameterization). Activity networks may be also changed at run-time by using an easy-to-use graphical interface, thus allowing users to change the control and data flow.

Endeavors is a reflective system. Endeavors' system components are themselves modeled as artifacts, allowing the system to keep a dynamically, distributable, and customizable internal model of itself. As mentioned in the previous chapters, this is useful to control the way an existing process model may be customized and changed to cope with unforeseen changes in the environment being supported. In Endeavors, even the activity network interpreter (the core of the system supporting process model enactment) is modeled as an activity object within the system, allowing it to be sent across networks, automatically made persistent, and dynamically spawned as needed. This reflexivity gives the Endeavors system programmatic self-reference for changing its own architectural components, including user interfaces, activity network structures, resource allocations, and distributed communications.

By taking advantage of these features, Endeavors users may increase state and transition completeness when needed. In particular, some changes (i.e., the changes related to the control and data flow) may be easily performed through a graphical tool (i.e., the network editor). This reduces the cost of changing the process model to cope with unforeseen situations, which is one of the most important drawbacks of PSSs that adopt the Endeavors approach to manage deviations.

Comparing Endeavors with PROSYT is hard. As already mentioned the two approaches to cope with unforeseen situations they adopt, namely, changing the model or relaxing the constraints stated by the model to allow users to perform deviations under the PSS's control, are not alternative. What can be said in general is that the PROSYT approach increases state and transition completeness with no cost for the users, while the Endeavors' approach requires some work. On the other hand, changing the process model is often necessary to cope with major deviations (e.g., sometimes new operations have to be added). The same consideration is valid for SPADE and OASIS, which adopt the same approach adopted by Endeavors to face unforeseen situations.

In conclusion, the PROSYT approach seems to be better to cope with minor deviations which have a low impact on the overall process or with situations that require an immediate answer and that are known to occur sporadically, while the SPADE, OASIS, and Endeavors approach is better suited to cope with major deviations which will probably occur again in the future. An advanced PSS should provide features to support both approaches.

5.5 PEACE

PEACE [5, 4] proposes a PDL to describe and reason about facts that are believed to be true during software process. The authors of PEACE focus their attention on the problem of inconsistency between the state of the PSS and the state of the process actually followed and try to solve the problem by using non-monotonic logical reasoning.

The PEACE PDL adopts a logic based, goal oriented approach for process modeling. A PEACE process model is organized as a set process model fragments (PMFs). Each PMF describes a process step with its associated goal. It is composed of a PMF specification and a set of PMF implementations. A PMF specification describes the process step associated to the PMF in terms of process step inputs, outputs, intrinsic role of the associated human agent who is responsible for the enactment of the step, incoming and outgoing events, and intrinsic precondition and postcondition.

PEACE distinguishes between complex and elementary PMF implementations. An elementary PMF implementation describes how a tool can be used to implement a PMF. More interesting is implementation of complex PMFs. A complex PMF implementation I describes how the process step associated with the PMF can be implemented as a composition of other PMFs $F_1, ..., F_n$ which represent the sub-steps of I. Each F_i specification is enriched in I by a contextual role, a contextual precondition, and a contextual postcondition. They express conditions that are specific to a particular project while their intrinsic counterpart express general conditions that have to be verified in all projects. The role (resp. precondition and postcondition) of a PMF is given by the conjunction of the intrinsic and contextual role (resp. precondition and postcondition).

A complex PMF implementation I describes also a set of contextual rules that establish logical ties between pre and postconditions of PMFs F_i. They are used to describe how the PMFs used in an implementation have to be used. As an example, suppose that $Post_1$ and $Post_2$ are the postconditions of F_1 and F_2, respectively. The contextual rule $Post_1 \Rightarrow Post_2$ can be used to describe the fact that in the particular project described by I, enaction of F_2 is not necessary if F_1 has already been enacted.

In PEACE, intrinsic and contextual roles, preconditions, and postconditions, are expressed using an autoepistemic, nonmonotonic modal logic based on the Moore's logic [68]. It features an operator to describe users beliefs. By taking advantage of the expressive power of this logic, PEACE can provide process support even in presence of uncertain and incomplete knowledge of the actual performing process [3]. In fact, during the enactment, beliefs may change over time and the runtime system is able to non-monotonically retract the derived propositions.

During enactment, a logical reasoning is performed to choose the process steps that have to be executed to satisfy a given goal. Given the knowledge about the process state owned by the process engine, several sequences of steps can be used to satisfy such goal. The user is given the responsibility of choosing one of these sequences. The chosen sequence is then executed. Users may add new beliefs or retract existing ones to describe new situations.

The PEACE approach to process modeling and enactment increases the environment flexibility, reducing the need for deviations. A PEACE process model may describe a wide range of process states and transitions and the right one may be chosen based on the actual beliefs of the system with respect to the environment. On the other hand, if a process step suitable to cope with a particular situation (i.e., a set of beliefs) has not been described, PEACE users do not have any way to face the situation. Conversely, in PROSYT, even if a process step (i.e, an operation or an allowed sequence of operations) defined to face a particular situation has not been described, the users can try to use (i.e., invoke) the existing operations, eventually violating the constraints embedded into the process model, to cope with the unexpected situation. Moreover, as mentioned in Section 4.1, the activity-based approach adopted by PEACE to describe processes does not help process designers in identifying the right granularity of process basic activities in order to support users even during deviations.

5.6 Inconsistencies and deviations support in WFMSs

Workflow systems are a special kind of PSSs. They are designed to assist groups of people in carrying out work procedures. They contain knowledge of how procedures are performed in the default case (they enact a *workflow model*). Since organizational changes and exceptions to the default case are inevitable, it is important that workflow systems allow their users to deviate from the workflow model.

5.6.1 Deviation support via on-the-fly modification of the process model

Several WFMSs support users in facing unforeseen situations by allowing them to change a process model on-the-fly (i.e., at enactment-time). Examples of such WFMSs are InConcert from InConcert, Inc. [79] and TeamWare Flow from TeamWare [80]. They offer tools to execute distributed processes involving multiple people, support the modeling and design of business processes, provide an open programming interface and a system integration environment to integrate existing off-the-shelf products. Most important, they allow users to modify processes on-the-fly so that exceptions and improved ways of working can be handled efficiently by the users, without having to wait for process modelers to fix the process.

In comparing all these systems with PROSYT we can make the same consideration made for SPADE, OASIS, and Endeavors. Changing the process and tolerating deviations are two complementary approaches to cope with unexpected situations that should be adopted together to obtain the most benefits. The former is better suited in the presence of major deviations which tend to become permanent while the latter is more suitable to cope with minor deviations which have a low impact on the overall process or with situations which require an immediate answer and that are known to occur sporadically.

The problem of *dynamic change* in workflow systems as the first step to enhance flexibility and usability of WFMS environments has been addressed by Ellis, Keddara, and Rozenberg in [37]. A dynamic change is a change of the workflow model performed during enactment. Performing a dynamic change is not a trivial task. The migration from the old workflow model to the new one has to be carefully managed in order to avoid inconsistencies in the data managed by the workflow system. In [37] the authors observe that not all the changes are possible at enactment-time. Given this assumption, they formalize the concept of dynamic change correctness in the context of the Information Control Net (ICN) model of workflow. They also present an algorithm that, given a dynamic change to an ICN model, is able to find a correct dynamic change equivalent to the first one.

5.6.2 Deviation support via exception handling techniques

Several WFMS [75] provide mechanisms to cope with unexpected situations via exception handling techniques. They provide facilities to describe the actions that have to be performed when an exception to the expected sequence of activities arises.

An example of WFMS that provide an explicit support to exceptions is WIDE. WIDE uses an activity-based PDL [20] to describe the business process. The focus is more on coordination and management of the people involved in the business activities than on automatic execution of these activities[1]. A WIDE model is composed of an organizational model and a process model. The organizational model describes the structure of the organization that carry out the process in terms of actors, groups, organization functions, and their relationships. The process model describes the documents accessed during the process, the tasks that compose the process, and their relationships. Exception handlers are composed by a pair $\langle condition, reaction \rangle$. The condition part specifies an event which triggers the reaction. Examples of events are violation of process constraints expressed in the model, updates of workflow variables, user rejecting a task which was assigned to him or her, agent not available, and user trying to jumping forward or backward in the task flow. The reaction part specifies the actions that have to be carried out when the exception arises.

The problem with such an explicit approach to handle unexpected situations (i.e., exceptions) is that, in some sense, it is appropriate for "expected" exceptions only. If an exception arises, which does not have a corresponding exception handling procedure, the WFMS cannot provide any help to its users. Conversely, the PROSYT approach does not require the process designer to provide a description of the exceptions which may occur and of the procedures to handle them in advance. Deviation handling is left to the users that may take benefit of the generic support given by the PSS to handle unexpected situations. This means that the PROSYT approach supports users even during "unexpected" exceptions.

[1] This choice results from the fact that WIDE was developed to support office procedures, which usually cannot be automated.

5.7 Managing standards compliance

Several standards have been proposed in software development as a means for transferring "good practice" in software engineering. Examples of such standards are ISO 12207 [58], IEEE 1074 [57], and PSS-05 [67]. They can be considered as very abstract process models. They set down the properties that both the process and the product must possess at given points during development.

In the last years, there has been intense interest in adopting such standards in industry as a consequence of product certification requirements. Adopting these standards results in a new problem for software industries. In fact, once a standard has been adopted, it is important to manage compliance of the actual software process with the standard. This problem is similar to the problem of supporting users in carrying out a business process. In both cases the consistency between a process model and the actual process followed by people has to be managed and guaranteed.

In [38] the problem of managing standards compliance in software development and particularly during the phases of software requirements analysis and specification is described and an approach to automate this activity is presented. This approach is based on the use of DOORS (Dynamic Object Oriented Requirements System), a generic document management system. The approach is defined as "tolerant". Developers are free to organize for themselves the way they reach the goals set by project management. They are provided with ways to assess where they are with respect to their duties to conform to the practices.

Each standard prescribes a set of *practices* which, in their turn, entail a set of *properties* on the artifacts produced during the process. In order to manage standard compliance, these properties are modeled as first-order logical formulas over the attributes of the documents put under DOORS control and over the relationships among these documents. This set of formulas may be viewed as the formalization of the normative model embedded into the standard. They play the same role of the process model in a standard PSS.

In order to control the level of compliance, for each practice, one or more *policies* may be specified. Each policy is triggered by an *event*. Events are opening a document, closing a document, updating an attribute of a document, or a combination of thereof. Combined events are specified using the FLEA language [22]. Each policy specifies a diagnostic function (chosen among a set of predefined functions) which determine the diagnosis provided to the user when the corresponding practice is violated. Depending on the *policy mode* (specified as part of each policy), different actions are carried out when a policy is triggered by an event:

- in "error mode", the check is immediately performed and the failure of the check prevents the action associated to the event from being completed. To fix the problem, the user is provided with the diagnosis generated by the diagnostic function defined by the policy;

- in "warning mode", the check is immediately performed and the failure of the check provides the user with the diagnosis, but the user is allowed to perform the action, thus deviating from the standard;

- in "guideline mode", the user is informed that it is advisable to perform the check, but it is not automatically performed.

Both practices and policies are specified at process model design time and cannot be changed at run-time (they are compiled in order to obtain the “process implementation”).

The approach proposed is similar to the approach adopted by PROSYT but several differences exist. First, PROSYT allows a more precise definition of the adopted process model: the structure of the artifacts may be specified together with the operations which may be executed upon them and with the invariants that characterize their consistency. Conversely, in the approach presented in [38], there is no a precise definition of the process that must be adopted. By describing the set of artifacts involved in the process with their attributes and with the properties that characterize the standard adopted, process designers may describe the set of “consistent states” only. This is the same expressive power given by PROSYT invariants. At the same time, the ability of describing not only states, but also operations is fundamental to characterize deviations from the model. The approach presented in [38] takes into consideration only the basic operations that are associated with the predefined events (opening and closing a document and changing the value of an attribute).

Second, policies are a means to specify both the consistency checking and the deviation handling policy adopted. They can be used to block actions which result in violating one or more practices and to determine the actions that have to be carried out when these practices have been violated. Conversely, PROSYT distinguishes between violating operation preconditions and moving outside the borderline determined by invariants.

Third, PROSYT policies may be changed at run-time and may differ from user to user while policies, as described in [38], are statically defined.

Finally, the approach adopted by PROSYT is more general-purpose. PROSYT may be used to manage standard compliance but it may also be used as a flexible PSS. Moreover, it is not limited to a particular process (the software requirements analysis process) but may be used to support and automate any kind of business process.

In conclusion, by applying the model given in Chapter 3, it is possible to say that the approach to manage standard compliance presented in [38] automatically increases state and transition completeness (if a policy mode different from the error mode is specified for at least one policy) thus increasing the ability of the environment of supporting its users even during deviations. At the same time, PROSYT is more flexible and allows process designers to better control the permitted deviations and the actions that have to be carried out when inconsistencies arise.

Chapter 6

Inconsistency issues in computer science

"As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality."
Albert Einstein (1879-1955)

The term "inconsistency" arises in various research areas of science, and computer science in particular. Logicians, for example, use the term "inconsistent" to denote a property of a theory. A set of first order logic formulas is said to be inconsistent if it does not exist any model that satisfies all of them. In classical logic it is not possible to derive any useful result from an inconsistent theory. Having recognized the importance of analyzing inconsistencies in a theory, several studies in non-classical logic deal with the issue of performing automatic reasoning in presence of inconsistency. Similarly, knowledge engineers try to detect and eliminate inconsistencies in their descriptions, and software engineers try to detect and handle inconsistencies affecting their specifications of software systems, and inconsistencies among specifications and implementations.

These simple examples show that the term "inconsistency" is used in several research areas but, in different contexts, it is implicitly associated with different meanings and this often results in misunderstandings among people having a different background. When, for example, a data base expert says that a data base is inconsistent, he or she uses word "inconsistent" with a meaning different from that used by a knowledge engineer speaking about inconsistency between the real world and the representation that a robot has of this world. The former uses the term "inconsistent" to refer to an internal property of a formal model (i.e., the set of tables included in the data base), the latter uses the same term to say that the robot owns a wrong representation of the real world.

Sometimes, the term "inconsistency" is used with different meanings even in the same research field and this further complicates the problem. For instance, software engineers use this term both to describe discrepancies between different specification documents and to describe differences between the software process view that the process support system has in a process-centered software engineering environment and the software process really executed.

This chapter discusses a wide range of works related to the concept of "inconsistency" and examines how the general problem of "inconsistency detecting and handling" takes place in various fields of science and how it is faced in different context, ranging from artificial intelligence to software engineering and data base studies. This survey does not pretend to be exhaustive. The range of works related to the concept of consistency management is too large to be covered entirely. This chapter describes some approaches focusing on the software engineering domain which is the domain of interest of the author. Some works related to other domains are simply mentioned.

In this chapter, the term inconsistency is used to refer to the general concept of a discrepancy between two models or between a model and the domain it describes [27]. This discrepancy makes the two models "incompatible". This means that it is not possible to find any domain that is described by both of them. This definition is similar to the definition of inconsistency given in logic but it is not limited to a single language (i.e., logics).

6.1 The term "inconsistency" in software engineering

In software engineering, the term inconsistency is used to describe a number of different phenomena. The next subsections show three different meanings given to this term in different areas.

6.1.1 Internally inconsistent specifications

A first, and perhaps most widely accepted, use of term inconsistency in software engineering is to indicate that a specification is contradictory (i.e. it is *internally inconsistent*), meaning that it includes conflicting statements. As an example consider the following specification of an abstract data type describing a stack:

type Stack[Item]
  uses Item, Boolean,
  syntax
    1 NEWSTACK: $\rightarrow$ Stack,
    2 PUSH: Stack $\times$ Item $\rightarrow$ Stack,
    3 ISEMPTY: Stack $\rightarrow$ Boolean,
    4 POP: Stack $\rightarrow$ Stack,
    5 TOP: Stack $\rightarrow$ Item,
  semantics
    for all stk **in** Stack; elm **in** Item;
    6 ISEMPTY(NEWSTACK)=**true**,
    7 ISEMPTY(PUSH(elm,stk))=**false**,
    8 POP(NEWSTACK)=**error**,
    9 POP(PUSH(elm,stk))=stk,
    10 TOP(NEWSTACK)=**error**,
    11 TOP(PUSH(elm,stk))=elm,
end Stack.

This specification is internally inconsistent. In fact, function `PUSH` is defined from `Stack`$\times$`Item` to `Stack` in line 2 but its semantics is given in line 7,9 and 11 as if it was defined from `Item`$\times$`Stack` to `Stack`.

Similar inconsistencies of syntactic nature are very common and quite simple to detect. The chance to produce internally inconsistent specifications may be reduced by using specification schemes that developers may instantiate to produce their specifications and/or developing tools to help users in writing correct specifications, like syntax-directed editors and automatic consistency checker. For instances, an automatic type checker for abstract data types could easily find inconsistencies like the previous one.

6.1.2 Inconsistencies between different specifications

The term inconsistency is also used by software engineers to describe the fact that different specifications (or, to be more generic, different descriptions or models) are incompatible because, for example, they describe the same component but one of them asserts fact ϕ while another assert a fact that is in contradiction with ϕ. For instance, suppose that the interface of a module `M` exports a type `T` and a procedure `P(in par1: T; out par2: T)` but its implementation defines the type `T` and a procedure `P(in par1: T; in par2: integer; out par3: T)`. Then the two descriptions (i.e., the `M`'s interface and its implementation) are inconsistent.

Usually these situations arise when different developers work together on a large project, developing different, but partially overlapping, parts of the same software system. This kind of inconsistencies has been formalized in Chapter 3 with the concept of domain-level inconsistency. In fact, the state in which two documents are inconsistent usually violates the constraints stated in the process model and, consequently, it is a domain-level inconsistent state.

While the traditional approach to deal with such kind of inconsistencies is to prevent them, and to employ means to restore consistency as soon as the inconsistency is detected, different researchers claimed that it is often undesirable to enforce consistency at any stage, because this could reduce developers' freedom, especially during the initial phases of system design.

In [41] and [43], the problem of managing inconsistencies between different descriptions during large software development is described. The papers describe an approach that allows software development to proceed even in presence of inconsistencies between different *views* of the system provided by different developers. Any developer has his or her own *viewpoint* that is a loosely-coupled, locally managed partial specification also encapsulating knowledge about the representation language used to specify the system, and information about the development process. This additional information supports the formalization of infra-viewpoint and inter-viewpoint consistency rules. This rules formalize consistency constraints inside single viewpoints or between two or more viewpoints describing different subsystems (possibly written in different languages) and give the process manager the chance to specify in a formal way automatic procedures to handle inconsistencies.

The problem of acting in the presence of inconsistency has been faced in [70]. The paper describes a framework for managing inconsistencies that affect software specifications. The idea is that a specification may contain a number of inconsistencies which can be detected in various ways (e.e., using logic-based consistency checking). A detailed analysis of these inconsistencies may suggest a set of actions that can be used to deal with each inconsistency. The choice of which action to perform can be determined by analyzing the consequences that each of these alternative actions has on the original specification. The paper also suggests a particular kind of impact analysis based on so-called "derivable information". A specification's derivable information is any piece of information that can be obtained through some inference process from that specification. By examining the relationship between the information derivable from the original specification and the information derivable from each possible consequent specification obtained by applying the available actions to the original specification, it is possible to "measure" the impact of different actions.

K. Narayanaswamy and N. Goldman in [69] address the problem of consistency checking and maintenance in repositories of PSEEs, which contain software process artifacts. Multiple access to these artifacts may result in inconsistencies between different versions of the same document produced by different developers. Database techniques to preserve consistency, like locking methods, are too restrictive for cooperative software development environments that are concerned with long transactions on large documents. The authors propose an alternative approach based on negotiation of *proposed change* (PC) to documents that supports "lazy" consistency during development. Any developer makes the necessary modifications on a local copy of the documents and, after finishing modifications, submits a change proposal to the process support system. This system automatically manages PCs and controls the process of collecting and merging PCs in order to obtain a new version of the document. This approach explicitly distinguishes between the time when the system is expected to be in a consistent state, and the time when it can be in an inconsistent state, thereby increasing system flexibility.

PEACE, pollution markers (that will be described later), and the proposed change technique by Narayanaswamy and Goldman are all significant examples of mechanisms that can be used to enrich a modeling language, in order to ease the representation and management of domain-level inconsistencies, to increase the flexibility of the process support system, and to prevent both environment-level deviations and environment-level inconsistencies.

S. Horwitz and T. Reps in [54] describe a language-independent program representation (the *program dependence graph*) and discuss how it can provide the basis for powerful programming tools that address problems such as understanding the differences between several versions of a program. These tools can be very useful to understand the sources of inconsistencies between different versions of the same program.

In [56] A. Hunter and B. Nuseibeh present an adaptation of classical logic, QC logic, that allows reasoning even from an inconsistent base of facts. In traditional logic, reasoning in presence of inconsistency is impossible because from an inconsistent premise it is possible to derive any formula (this phenomena is known as *trivialization*). QC logic, instead, prohibit all trivial derivations, still allowing all resolvants of the assumptions to be derived. The authors describe also an extension to their logic that allows logical analysis of inconsistent specification, qualifying inferences from inconsistent specifications, and identifying possible sources for an inconsistency. They apply their approach to multiperspective software specification [41] in order to increase the ability of the process support system in reasoning in presence of inconsistencies.

6.1.3 Inconsistencies between a computerized system and the domain it supports

The third common use of the term inconsistency in software engineering is to denote that a computerized system does not necessarily have a correct view of the domain it is supposed to support. This concept has been addressed in Chapter 3, where it was defined as "environment-level inconsistency".

In [39] the authors address the problem of inconsistencies between the expected behavior of a system, captured by the requirements specification, and the actual behavior of the system. This problem may result because:

- the human agents in the environment may behave in ways that were not anticipated, or were infeasible at requirements definition time;
- the post-deployment evolution of the environment can make initial assumptions that are no longer valid.

The paper describes a dynamic approach to cope with this problem based on runtime detection and automatic response to violations. The approach requires that alternative system designs are explicitly represented as system parameters and/or alternative refinement of the design tree. In particular, two kinds of parameters can be automatically extracted from a requirements specification expressed by using the goal-oriented approach described in [31, 32]: monitored parameters (used to observe environment behavior) and control parameters (used to change system behavior in response to changes in the environment).

The problem of describing inconsistencies that could arise in complex software systems, like a programming environment, has been addressed by T. M. Hagensen and B. B. Kristensen in [53]. In this work, the authors suggest the need of viewing software systems such as programming environments, from a consistency perspective. They introduce a general framework for discussing and describing consistency, which is based on the concepts of *description* and description *interpretation*. Consistency of descriptions is defined as a relation between their interpretations. Given a description D of some domain $\mathcal{D}$ its interpretation is defined as the set of elements of $\mathcal{D}$ modeled by the D. Using these definitions they formalize the concept of internally consistent description and various types of consistency relations between different descriptions. The authors use their framework to model a programming environment using a graphical notation to describe different kind of descriptions involved in such environment, like text files, structured files, and executable files, and to model the operations allowed on these descriptions. Using this model the authors make some consideration about the various features provided by different programming environments which have an impact on the consistency relations. The paper includes also a preliminary proposal for a taxonomy of the techniques for consistency handling in programming environments. This work is similar to the work presented in Chapter 3 but it is not formal and it focuses on domain-level inconsistencies only, leaving out environment-level inconsistencies.

The general relationship between a computerized system and the application domain where it is supposed to be used has been recently examined by Jackson in [59] and [60]. Jackson discussed the relationship between these two entities, and provide important insights to better clarify the relationship between "the world and the machine". Our work provides formal foundations to discuss this general problem. The framework has been applied in a more restricted domain (business processes and, more specifically, process-centered software engineering environments) only because this is the context where this work originated.

Inconsistencies in PSEEs

A special case in which a computerized system may not have a correct view of the domain occur in PSEEs when there is a discrepancy between a software process model and the process actually performed.

Several PSEEs try to solve this problem by adopting different approaches. Most of them were described in Chapter 5. This section describes other works related with the problem of inconsistency formalization and management in PSEEs.

This topic has been discussed by Dowson and Fernström [35], who introduced an informal model that has often been referenced in the literature. This model is based on the identification of three conceptual layers: the *process definition domain* (i.e., the process models), the *process enactment domain* (i.e., the enactment of process models), and the *process performance domain* (i.e., the actual process performance). Comparing this model with the framework described in Chapter 3 we may observe that the process definition domain corresponds to what has been called the process model. The process enactment domain is what has been called the process support system. Finally, the process performance domain is the business process actually carried out. With respect to Dowson and Fernström's work, our approach provides a more formal characterization of the problem. In particular, our description of (in)consistency, completeness, and coherence formalizes the relationships between process performance (i.e., business process) and process enactment (i.e., process support system).

The importance of formalizing this relationships has been identified in [40]. In this paper, Fernström extends and partially formalizes the framework introduced in [35]. He discusses the importance of relating process enactment and process performance models and the need for a formal description of this relationship. The paper focuses on formally describing state synchronization between process enactment and process performance during system progress.

The problem of detecting and representing environment-level inconsistencies and deviations that may arise during the software development process, has been also addressed by Cook and Wolf in [23]. In this paper, the authors develop a *process validation technique* for detecting and characterizing any differences between a formal model of the software process being executed and the actual execution of the process itself. The enacted process model and the process that is actually executed are seen as streams of events, and metrics to measure the distance between these two streams are provided. These metrics are all based on string distance concepts and range from simple exact matching to a non-linear distance measure in terms of event insertions and deletions. The authors apply their metrics to the ISPW6/7 example, showing how it is possible to measure the correspondence between a model of this process expressed as a Petri net and different examples of process execution.

A first informal definition of some of the concepts presented in Chapter 3 was given in [8]. This paper identifies three basic mechanisms for dealing with "on the fly" deviations. The *bypass mechanism* allows the users to perform operations that are not allowed by the process model; the *triggering mechanism* catches the events that produced deviation; the *reconciling mechanism* supports the reconciling of the actual process with the process model. In our environment, the bypass mechanism is supported by allowing users to invoke the exported operations even if the conditions that regulate their execution do not hold. The triggering mechanism is the one that catches deviations, identifies violation of invariants, and stops enactment if required. The reconciling mechanism supports the users in reconciling the actual process with the process model by providing them with all the necessary information about the deviations that occurred.

6.2 Database systems and information technology

Databases are probably the most common kind of computer applications. Databases manipulate large amounts of data and a wide range of semantic integrity constraint can be stated and enforced in order to control input and modification of these data.

Databases may become inconsistent when only partial modification of data is accomplished and this partial modification do not satisfy integrity constraints. Typically this happens when, due to application or hardware failures, atomic transactions are interrupted before they have completed their work. Many protocols have been studied and implemented to correctly perform databases update. Typical examples are (i) two-phase commit protocols, used to ensure that transactions are executed atomically; and (ii) timestamp ordering of operations, used to ensure serializable execution of concurrent transactions.

Applying the framework presented in this thesis to this kind of inconsistencies, which can be assumed to be handled by the database management system, is not interesting. Conversely, this section focuses on another kind of database inconsistency that results from the need of adding "special" data to an existing database, which violate the integrity constraints. This happens because almost all such constraints have exceptions, i.e., data that have to be added to the database but violate one or more constraints. This situation differs from previous because here integrity violation is explicitly required by users. This situation may be classified as a domain-level inconsistency since it regards the domain of the process and it usually violates the constraints stated by the business process model.

In [19] A. Borgida describes an approach to face this problem in which database constraints are viewed as normal conditions that may occasionally conflict with the actual state of the world. For this reason he focuses his attention on allowing violations of such constraints to occur and persist. Borgida's solution is based on the introduction of an exception handling mechanism. The occurrence of a violation is signaled by the creation of a special object belonging to a special class of data, which contains the exceptional data. These *violation objects* have properties identifying the constraint that was violated. Exception handlers are special transactions, automatically called when violations occur and manipulating the corresponding violation objects. Every kind of violation has specified its own handler. Special commands allow exceptions to persist, adding the exceptional data to the database. This approach lets users manipulate exceptional data in an explicit manner, via suitable exception handling transactions.

In [6] R. Balzer presents a technique to automatically relax integrity constraints formulated as logical sentences in order to allow their violation from exceptional data and to mark these data with *pollution markers*. A pollution marker is an atomic sentence indicating the violation of specific constraint by a particular datum. In this approach, relaxing integrity constraints is an automatic operation and users do not need to provide any special operation to detect and handle exceptional data.

D. Gabbay and A. Hunter in [47] assert that inconsistencies in databases are not necessarily "bad". They argue that dealing with inconsistencies is not necessarily done by restoring consistency but can be handled by supplying rules telling one how to act when inconsistencies arise. They distinguish between the object-level language, used to describe the data contained in the database and the integrity constraints stated on these data, and the meta-level language used to describe the actions to be taken in order to handle inconsistent data. They describe a meta-level language based on first-order linear temporal logic, with `since` and `until` operators that allows them to formally describe how to act in presence of inconsistencies. They also describe an approach (based on a previous work by Gabbay [46]) to execute their temporal specifications. Taking advantage of this approach, they show how to create automatic tools that interpret a specification given in their meta-level language and perform the actions needed to handle inconsistencies.

In [21] the authors describe an approach to automatically generate transactions that restore database consistency after an inconsistency has been detected. The traditional response of a DBMS to an incorrect transaction (i.e., a transaction that violates some database constraint) is to rollback the transaction. In this paper an alternative approach is described, in which the database system autonomously reacts to inconsistencies by triggering a set of automatically generated repair actions. This interesting approach to integrity maintenance is useful to allow temporary deviations from database consistency and to support reconciliation.

Strong and Miller in [76] examine the problem of inconsistencies and exceptions in computer-based information processes. According to the authors, exceptions are cases that cannot be correctly processed by computer system without manual intervention. The authors try to derive a set of managerial recommendations for treating exceptions, by examining an information process used by an existing electronic system factory. They conclude that exceptions are intrinsic to organizational processes and cannot be considered as errors to be eliminated. According to this perspective, it is necessary to provide support and guidance to humans in order to enable them to solve exceptions. It is, however, impossible to replace humans in solving these process exceptions.

6.3 Logics, artificial intelligence, and robotics

As mentioned earlier, different AI studies face the problem of inconsistency management. In particular, they try to solve the problem of inferring useful conclusions from a set of inconsistent logical formulae and they deal with the problem of detecting contradictions in a knowledge base, identifying their source, and modifying the knowledge base to remove inconsistencies.

Multivalued [1, 36] and nonmonotonic logics [65, 52, 34] can be used to describe potentially inconsistent situations and to analyze them. They allow reasoning even in presence of uncertain information and can be used to identify potential sources of inconsistency and to remove them. The main advantage of these logics is that they can automatically retract the derived consequences of a set of formulas if new premises are added, which contradict previous results.

Most AI systems, and particularly autonomous agent developed by researchers in robotics, maintain a model of their internal state and of the external environment. During their evolution, they update this knowledge base adding new inferences and new premises as soon as they become available. For these systems, it is very important to detect inconsistencies in the knowledge base and identify and remove their source. The subsystems performing this task are usually called *belief revision* or *truth maintenance systems* [45, 48].

In [66] and [49] two different approaches to the problem of belief revision are described. [66] describes a belief revision system based on a logical language specifically conceived for this purpose. The most important feature of this language is that it is not possible to derive trivial facts from a set of inconsistent premises. Conversely, [49] addresses the problem of belief revision without making any assumption on the language used to represent these believes. Authors formally describe concepts of theory *expansion* (adding new formulas and their logical consequences), *contraction* (removing formulas and their consequences), and *revision* (adding new formulas that are inconsistent with the theory and reorganizing the theory itself; removing some formulas and their logical consequences, to obtain a consistent theory) and prove that from any contraction function it is possible to derive a revision function, and vice versa. Finally, they describe how it is possible to obtain a contraction function given a partial ordering on formulas of the theory satisfying some postulates.

Chapter 7

Conclusions and future work

"The secret to creativity is knowing how to hide your sources."
Albert Einstein (1879-1955)

7.1 Summary of the work presented

An effective and reproducible business process is fundamental for mature business organizations. Process support systems (PSSs) offer tools to model, improve, and automate business processes. Therefore, in principle, they should constitute one of the most useful tools for modern business organizations. Given these premises, and considering that PSS technology is now available, both as commercial products and as research prototypes, it is surprising to observe that such technology is rarely adopted. One of the major problems of currently available PSSs is their lack of flexibility and, in particular, their inability in supporting their users when unexpected situations arise during process enactment. Currently available PSSs force their users to follow the sequence of activities specified in the process model, thus preventing them from performing the actions required to cope with unexpected situations. Deviations from the process model are not allowed.

The main consequence of this approach is that users are forced to exit from the PSS and relinquish control of the PSS in order to perform the actions required to solve the unexpected situation. This behavior leads to an inconsistency between the PSS and the process actually followed. As a consequence, it prevents the PSS from continuing to provide support to the process.

In this thesis a framework to formalize the concepts of deviation and inconsistency was proposed. This framework formalizes the relationships between a business process, its process model, and its process support system (which enacts the model to guide and automate the process). Understanding this relationship is an essential step towards designing effective PSSs capable of overcoming the limitations of existing PSS.

The framework is based on formal models of the business process $\mathcal{BP}$ and of its process support system $\mathcal{PSS}$. These models are used to define in a precise way the concepts of inconsistency, deviation, completeness, and coherence. A distinction between domain-level inconsistencies (DIs) and environment-level inconsistencies (EIs) is formalized. DIs are related to the application domain being studied, and refer to the intrinsic characteristics of the process that is supported. They can occur even when no process technology is used. EIs refer to the relationship between a business process and its process support system. They are generated by the inability of the process support system to track the progress of the business process correctly. Finally, we argue that the ability of a process support system to tolerate and control EIs and the deviations that cause them can be expressed by the degree of completeness and coherence that it is able to ensure.

The framework has been applied to assess and compare the features of a number of existing PSSs. This assessment allowed us to argue that completeness is mainly related to the features and characteristics of the language used to model processes, while coherence is more strictly related to the architecture of the process support system.

In the second part of the research described in this thesis, the results of this work have been used to guide the development of PROSYT, a PSS capable of tolerating deviations from the process model during enactment. PROSYT features effective mechanisms to help users in facing unexpected situations, continuing process support and automation even during deviations. After a deviation has occurred, it provides its users with all the information needed to reconcile the process model and the process actually followed. The PROSYT approach to process modeling and enactment allows deviations to be tolerated as long as they do not affect the correctness of the system.

One of the key features of PROSYT is its ability of modifying the level of enforcement adopted (i.e., the deviation handling policy) and the actions that have to be performed when the overall system correctness is violated (i.e., the consistency handling policy) at enactment-time to adapt them to the different situations that may arise during process enactment. The process manager is also allowed to choose the deviation handling policy on a per-user basis; i.e., different users may be allowed to perform different kind of deviations.

7.2 Future work

The effectiveness of the approach adopted by PROSYT in facing unexpected situations has been proven on a number of in-house case studies. At this stage, PROSYT has not yet been used to support complex business processes. As a consequence, future work will be devoted in trying to use PROSYT to support real, complex business processes. This experience will allow us to asses the environment.

Another area which is worth of future investigation is the topic of supporting reconciliation. The support offered by PROSYT during the reconciling activity is very general and appears to be powerful enough to help process manager during this crucial activity, but more experience is needed in this field. One goal of this research is to find the mechanisms required to automatically identify a set of possible remedies to the deviations occurred, suggesting them to the process manager in order to automate the reconciling activity when possible.

Finally, the possibility of providing a graphical version of the PLAN PDL in order to simplify the activity of process modeling, is currently being evaluated.

Appendix A

The set of PLAN predefined methods

This appendix describes the set of predefined methods which compose the PROSYT API. By using these methods, the process designer may implement advanced processes. There are ad-hoc methods to invoke external tools, create new artifacts, query the state of an artifact belonging to a folder, make a folder or a repository visible to a given user or a group of users, and so on. They can be used in the body of exported operations, automatic operations, and other methods. They can also be used in guard and preconditions.

For each method a brief description is provided together with the signature, a detailed description of the parameters, and a reference to other methods related to the method being described.

A.1 Predefined methods in artifact types

`void associateFile(String logicalName, String fileName)`

Description

In PROSYT each artifact is characterized by two types of information. Information directly managed by PROSYT, which is kept in the artifact's attributes, and information that is not directly managed by PROSYT but is managed by external tools that creates and modifies it (e.g., an editor or a compiler). This information is kept in external files. As an example, an artifact `module` describing a software module could be characterized by a name and a version number, described as attributes, and could be associated to a file containing the source code of the module and to a set of files containing the module object code for different platforms.

Method `associateFile` binds a file containing information managed by external tools to an artifact under the given logical name. The logical name is used to refer to the file in method `callTool`.

Parameters

`logicalName`
: The logicalName given to the file (e.g., “source” or “object”).

`fileName`
: The name of the file.

See also

`dissociateFile`
`callTool`
`callUserTool`

`void callTool(String tool, String[] args)`

Description

Calls a tool on the “server” machine, that is, on the host that runs the repository which holds this artifact. The tool is run in the same directory that contains the files associated to the artifact.

Parameters

`tool`
: The full path of the tool (i.e., executable).

`args`
: An array of strings passed to the tool as command lines parameters. If a string in the form `"<name>"` is used then it is automatically substituted with the name of the file associated to the artifact under the name `name` before calling the tool.

See also

`callUserTool`
`associateFile`

`void callUserTool(String tool, String[] args)`

Description

Calls a tool on the “client” machine, that is, on the host that runs the project browser of the user that invoked the exported operation which resulted in calling this method. If this method is called as a result of an automatic operation (which is not called by a specific user) it has no effect.

The files associated to the artifact are automatically moved from the “server” to the client machine.

Parameters

`tool`
: The full path of the tool (i.e., executable) on the “client” machine.

`args`
: An array of strings passed to the tool as command lines parameters. If a string in the form `"<name>"` is used then it is automatically substituted with the name of the file associated to the artifact under the name `name` before calling the tool.

See also
`callTool`
`associateFile`

```
ArtifactType createNewArtifact(String artifactType,
    String name)
```

Description
Creates a new artifact whose type is `artifactType` and whose name is `name`. The new artifact belongs to the same folder of the artifact which created it. By default the new artifact has a standard icon which can be changed by invoking the method `setIconFileName`. It returns a reference to the artifact created that may be used in automatic operations to identify the destination of a request of executing an exported operation.

Parameters
`artifactType`
The type of the new artifact.

`name`
The name of the new artifact. It can be changed at run time by using the method `setName` and can be accessed through the method `getName`.

Return value
A reference to the new artifact.

See also
`createNewArtifact(String, String, Object[])`
`getName`
`setName`
`setIconFileName`

```
ArtifactType createNewArtifact(String artifactType,
    String name, Object[] args)
```

Description
Creates a new artifact whose type is `artifactType` and whose name is `name` and passes **a deep copy** of the array of objects `args` to the `initialize` automatic operation of the new artifact. The new artifact belongs to the same folder of the artifact which created it. By default the new artifact has a standard icon which can be changed by invoking the method `setIconFileName`. This operation returns a reference to the artifact created that may be used in automatic operations to identify the destination of a request of executing an exported operation.

Parameters
`artifactType`
The type of the new artifact.

`name`
The name of the new artifact. It can be changed at run time by using the method `setName` and can be accessed through the method `getName`.

`args`
The set of arguments passed to the `initialize` automatic operation of the new artifact. **They are passed by copy (deep)**.

Return value
A reference to the new artifact.

See also
`createNewArtifact(String, String)`
`getName`
`setName`
`setIconFileName`

`void delete()`

Description
Deletes an artifact together with all the files associated to the artifact that are not associated to other artifacts.

See also
`createNewArtifact`

`void dissociateFile(String logicalName`

Description
Breaks the binding between an artifact and the file that was previously associated to the artifact under the logical name `logicalName`.

Parameters
`logicalName`
The logical name of file that have to be dissociated from the artifact.

See also
`associateFile`

`String getIconFileName()`

Description
Gets the name of the file which contains the icon used to show this artifact in the project browser.

Return value
The name of the icon file.

See also
`setIconFileName`

String getName()

Description

Gets the name of the artifact.

Return value

The name of the artifact.

See also

setName

void moveTo(FolderType folder)

Description

Moves the artifact to the given folder. If the folder is on a different repository on a different host the files associated to the artifact are moved also.

Parameters

folder

A reference to a folder (i.e., a value returned by a createNewFolder operation.

void sendEvent(String event)

Description

Sends the specified event. An event is described by a string in the form "<name>(<arg_1>,...,<arg_n>)". This event triggers the automatic operation: <name1>(<par_1>,...,<par_n>) if <name1> equals <name> and either <par_i> equals <arg_i>, or <par_i> is a *free parameter* (described in PLAN by using a single capital letter). As a result of the triggering, all the exported operation free parameters, if any, are bound to the corresponding arguments of the triggering event. As an example, the event represented by the string: "delivered(module1, 11-6-1997)" triggers the exported operation:
delivered(X,Y)
As a result of the match between the event and the exported operation signature, the free parameter X is bound to the string "module1" and the free parameter Y is bound to the string "11-6-1997". These parameters can be used in the body of the exported operation.

Parameters

event

The string representing the event to be sent.

void setIconFileName(String iconFileName)

Description

Sets the icon used to represent the artifact in the projectBrowser tool.

Parameters

`iconFileName`
The name of the file which contains the icon. Any gif image with a dimension of 32 by 32 pixels may be used. In the current implementation of the projectBrowser, icons are searched first locally (i.e., in a special directory of the client machine) and then remotely through the http protocol. The base URL used to access icons remotely may be specified in the projectBrowser preferences.

`void setName(String name)`

Description
Sets the name of the artifact.

Parameters
`name`
The name of the artifact.

See also
`getName`

A.2 Predefined methods in folder types

Folder types provides most of the methods provided by artifact types together with some other specific methods. In particular, in the body of folder types' operations the following methods, described in the previous section, can be used: `createNewArtifact`, `delete`, `getIconFileName`, `getName`, `moveTo`, `sendEvent`, `setIconFileName`, `setName`.

Several additional methods are defined to be used specifically in folder types' operations. They are:

`boolean allArtifacts(String state)`

Description
Returns true if all the artifact that belong to this folder are in the specified state.

Parameters
`state`
A string corresponding to one of the admissible states of artifacts.

Return value
True if all the artifact which belong to this folder are in the specified state.

See also
`allArtifacts(String, String)`
`allFolders`
`state`

`boolean allArtifacts(String artifactType, String state)`

Description

Returns true if all the artifact that belong to this folder and whose type is `artifactType` are in the specified state.

Parameters

`artifactType`

The type of the artifacts whose state is queried.

`state`

A string corresponding to one of the admissible states of artifacts.

Return value

True if all the artifact that belong to this folder and whose type is `artifactType` are in the specified state.

See also

`allArtifacts(String)`
`allFolders`
`state`

`boolean allFolders(String state)`

Description

Returns true if all the direct subfolders of this folder are in the specified state.

Parameters

`state`

A string corresponding to one of the admissible states of folders.

Return value

True if all the the direct subfolders of this folder are in the specified state.

See also

`allFolders(String, String)`
`allArtifacts`
`state`

`boolean allFolders(String folderType, String state)`

Description

Returns true if all the direct subfolders of this folder whose type is `folderType` are in the specified state.

Parameters

`folderType`

The type of the folders whose state is queried.

`state`

A string corresponding to one of the admissible states of folders.

Return value

True if all the direct subfolders of this folder whose type is `folderType` are in the specified state.

See also

`allFolders(String)`
`allArtifacts`
`state`

`FolderType createNewFolder(String folderType, String name)`

Description

Creates a new folder whose type is `folderType` and whose name is `name`. The new folder is a direct subfolder of the folder that created it. By default the new folder has a standard icon which can be changed by invoking the method `setIconFileName`. It returns a reference to the folder created that may be used in automatic operations to identify the destination of a request of executing an exported operation.

Parameters

`folderType`
The type of the new folder.

`name`
The name of the new folder. It can be changed at run time by using the method `setName` and can be accessed through the method `getName`.

Return value

A reference to the new folder.

See also

`createNewFolder(String, String, Object[])`
`getName`
`setName`
`setIconFileName`

`FolderType createNewFolder(String folderType, String name, Object[] args)`

Description

Creates a new folder whose type is `folderType` and whose name is `name` and passes **a deep copy** of the array of objects `args` to the automatic operation `initialize` of the new folder. The new folder is a direct subfolder of the folder that created it. By default the new folder has a standard icon which can be changed by invoking the method `setIconFileName`. This operation returns a reference to the folder created that may be used in automatic operations to identify the destination of a request of executing an exported operation.

Parameters

`folderType`
The type of the new folder.

`name`
The name of the new folder. It can be changed at run time by using the method `setName` and can be accessed through the method `getName`.

`args`
The set of arguments passed to the `initialize` automatic operation of the new folder. **They are passed by copy (deep)**.

Return value
A reference to the new folder.

See also
`createNewFolder(String, String)`
`getName`
`setName`
`setIconFileName`

`void enableGroup(String group)`

Description
Makes the folder (i.e., its content) visible to the users that belong to the given group. Users enabled are allowed to open the folder and to see the artifacts and subfolders included.

Parameters
`group`
A string representing the group of people which has to be allowed to see the folder. It is one of the groups defined in the project type.

See also
`enableUser`

`void enableUser(String userlogin)`

Description
Makes the folder (i.e., its content) visible to the specified user. He or she is allowed to open the folder and to see the artifacts and subfolders included.

Parameters
`userLogin`
The login of the user as specified in the `addUser` operation.

See also
`enableGroup`
`addUser`

`void moveTo(RepositoryType repository)`

Description

Moves the folder, together with the artifacts and all the subfolders it contains, to the given repository (i.e., at the first level of the hierarchy of folders included into the given repository).

Parameters

`repository`

A reference to a repository (i.e., one of the global variables defined into the `repositories` section of the project type.

`boolean state(ArtifactType artifact, String state)`

Description

Returns true if the specified artifact is in state `state`.

Parameters

`artifact`

A reference to the artifact whose state is queried (i.e., a value returned by a `createNewArtifact` operation).

`state`

A string corresponding to one of the admissible states of artifacts.

Return value

True if the specified artifact is in state `state`.

See also

`state(FolderType, String)`
`allFolders`
`allArtifacts`

`boolean state(FolderType folder, String state)`

Description

Returns true if the specified folder is in state `state`.

Parameters

`folder`

A reference to the folder whose state is queried (i.e., a value returned by a `createNewFolder` operation).

`state`

A string corresponding to one of the admissible states of folders.

Return value

True if the specified folder is in state `state`.

See also

`state(String)`
`allFolders`
`allArtifacts`

A.3 Predefined methods in repository types

Repository types provides all the methods provided by artifact types excluding methods: `allArtifacts`, `createNewArtifact`, `delete`, `getName`, `moveTo`, `setName`, and `state(ArtifactType, String)`. These methods have been excluded either because a repository may directly include folders only (and not artifacts), or because a repository is a static entity that cannot be moved and whose name is choosen statically.

No additional methods are defined to be used specifically in repository types' operations.

A.4 Predefined methods in project types

Project types provides methods `getIconFileName`, `sendEvent`, and `setIconFileName` with the same sintax and semantics of the corresponding methods provided by repository types, together with the following methods which are specific of project types:

`void addUser(String login, String passwd)`

Description

Adds a new user to the project.

Parameters

`login`

The user login.

`passwd`

The user password.

See also

`addUserToGroup`
`removeUser`

`void addUserToGroup(String login, String group)`

Description

Adds a user to the specified group.

Parameters

`login`

The user login.

`passwd`

The group in which the user has to be added.

See also

`removeUserFromGroup`

`void removeUser(String login)`

Description
Removes a user from the project.

Parameters
`login`
The user login.

See also
`addUser`

`void removeUserFromGroup(String login, String group)`

Description
Removes a user from the specified group.

Parameters
`login`
The user login.

`group`
The group from which the user has to be removed.

See also
`addUserToGroup`

Bibliography

[1] Robert Ackermann. *Introduction to Many-Valued Logics*. Routledge & Kegan, London, 1967.

[2] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing process-centered environments. *ACM Transactions on Software Engineering and Methodology*, 6(1), July 1997.

[3] S. Arbaoui and F. Oquendo. Managing inconsistencies between process enactment and process performance states. In *Proceedings of the 8th International Software Process Workshop*, Wadern (Germany), March 1993.

[4] S. Arbaoui and F. Oquendo. Software process performance support in PEACE. In *Proceedings of the 6th International Conference on Software Engineering and its Applications*, Paris (France), November 1993.

[5] S. Arbaoui and F. Oquendo. Peace: Goal-oriented logic-based formalism for process modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press, J. Wiley, 1994.

[6] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, Austin (Texas - USA), May 1991.

[7] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The architecture of the SPADE-1 process-centered SEE. In *Proceedings of the 3rd European Workshop on Software Process Technology*, LNCS 772, Villard de Lans (Grenoble), France, February 1994.

[8] S. Bandinelli, E. Di Nitto, and A. Fuggetta. Policies and mechanisms to support process evolution in PSEEs. In *Proceedings of the 3rd International Conference on the Software Process*, Reston (Virginia - USA), October 1994.

[9] S. Bandinelli, E. Di Nitto, and A. Fuggetta. Supporting cooperation in the spade-1 environment. Technical Report 2, December 1996.

[10] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Process model evolution in the SPADE environment. *IEEE Transactions on Software Engineering*, 19(12), December 1993.

[11] S. Bandinelli, A. Fuggetta, C. Ghezzi, and S. Grigolli. Process Enactment in SPADE. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992. Springer-Verlag.

[12] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: an environment for Software Process Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

[13] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.

[14] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Angelo Morzenti. A Multi-Paradigm Petri Net Based Approach to Process Description. In *Proceedings of the 7th. International Software Process Workshop*, Yountville, California (USA), October 1991.

[15] N. S. Barghouti and S. Arbaoui. Process support and workflow management with improvise. In *Proceeding of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*, Athens, GA, May 1996.

[16] N.S. Barghouti and G.E. Kaiser. Scaling up rule-based software development environments. In *Proceedings of the 3rd European Software Engineering Conference*, LNCS 550, Milano (Italy), October 1991. Springer-Verlag.

[17] N.S. Barghouti and B. Krishnamurthy. Using event contexts and matching constraints to monitor software processes. In *Proceedings of 17th International Conference on Software Engineering*, Seattle (Washington - USA), April 1995.

[18] Gregory Alan Bolcer and Richard N. Taylor. Endeavors: A process system integration infrastructure. In *Proceedings of the Fourth International Conference on Software Process (ICSP4)*, Brighton, UK, December 2-6 1996.

[19] A. Borgida. Language features for flexible handling of exceptions in information systems. *ACM Transactions On Database Systems*, 10(4), Dec 1985.

[20] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *Proceeding of the International Conference on Object-Oriented and Entity-Relationship*, pages pp. 341–354, Goldcoast, Australia, 1995.

[21] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems*, 20(3), 1995.

[22] D. Cohen, S. M. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *Proceedings of the 19th International Conference on Software Engineering(ICSE97)*, Boston, MA, May 1997. IEEE Computer Society Press.

[23] J. E. Cook and A. L. Wolf. Toward metrics for process validation. In *Proceedings of the 3rd International Conference on the Software Process*, Reston (Virginia - USA), October 1994.

[24] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 19th International Conference on Software Engineering (ICSE98)*, 1998.

[25] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A framework for formalizing inconsistencies in human-centered systems. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 5(3), July 1996.

[26] G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione. How to deal with deviations during process model enactment. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle (Washington - USA), April 1995.

[27] G. Cugola and C. Ghezzi. Inconsistencies in software development: Towards a reference framework. In *Proceedings of the ICSE'97 Workshop on "Living with Inconsistency"*, Boston, MA, May 1997.

[28] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. A Characterization of Mobility and State Distribution in Mobile Code Languages. In M. Mühlaüser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96*, pages 309–318. dpunkt, July 1996.

[29] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 93–111. Springer, April 1997.

[30] Gianpaolo Cugola. Tolerating deviations in process support systems via flexible enactment of process models. Technical Report 97.68, Dipartimento di Elettronica e Informazione - Politecnico di Milano, P.za Leonardo da Vinci 32 - 20133 Milano (Italy), December 1997.

[31] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, volume 20, pages pp. 3–50. 1993.

[32] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: An environment for goal-driven requirements engineering. In *Proceedings of the 19th International Conference on Software Engineering (ICSE97)*, Boston, MA, May 1997. IEEE Computer Society Press.

[33] E. Di Nitto and A. Fuggetta. Open issues in managing inconsistencies in human-centered systems. In *Proceedings of the ICSE'97 Workshop on "Living with Inconsistency"*, Boston, MA, May 1997.

[34] F.M. Donini, M. Lenzerini, D. Nardi, F. Pirri, and M. Schaerf. Nonmonotonic reasoning. *Artificial Intelligence Review*, 4:pp. 163–210, 1990.

[35] M. Dowson and C. Fernström. Towards requirements for enactment mechanisms. In *Proceedings of the 3rd European Workshop on Software Process Technology*, LNCS 772, Villard de Lans (Grenoble) France, February 1994.

[36] .J. Micheal Dunn and G. Epstein, editors. *Modern Uses of Multiple-Valued Logic*. Reidel, Dordrecht, 1977. Invited Papers of 5th ISMVL Symposium 1975 with Bibliography by R. G. Wolf.

[37] C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In *Proceedings of COOCS 95*, August 1995.

[38] W. Emmerich, A. Finkelstein, C. Montangero, and R. Stevens. Standards compliant software development. In *Proceedings of the ICSE'97 Workshop on "Living with Inconsistency"*, Boston, MA, 1997. IEEE CS Press. electronic publication.

[39] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. *IEEE Transactions on Software Engineering*, 1998. Submitted.

[40] C. Fernström. State models and protocols in process-centred environment. In *Proceedings of the 8th International Software Process Workshop*, Wadern (Germany), March 1993.

[41] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *Fourth European Software Engineering Conference (ESEC'93)*, Garmish, Germany, September 1993. Springer-Verlag.

[42] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

[43] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, March 1992.

[44] A. Fuggetta and C. Ghezzi. State of the art and open issues in process-centered software engineering environments. *Journal of Systems & Software*, 26(1), July 1994.

[45] A. Fuhrmann and M. Morreau. *On the Logic of Theory Change*. Number 465 in Lecture Notes in Artificial Intelligence. Springer Verlag, 1991.

[46] D. Gabbay. Declarative past and imperative future: Executable temporal logic for interactive systems. In *Colloquium on Temporal Logic in Specifications*, volume 398 of *Lecture Notes on Computer Science*. Springer-Verlag, 1989.

[47] D. Gabbay and A. Hunter. Making inconsistency respectable: Part 2 - meta-level handling of inconsistent data. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertanty*, LNCS 747. Springer-Verlag, 1993.

[48] P. Gärdenfors. *Belief Revision*. Cambridge Press, 1992.

[49] P. Gärdenfors and D. Makinson. Revisions of knowledge systems using epistemic entrenchment. In *Proceedings of the Second Conference on Theoretical Aspects of Reasoning About Knowledge*. Morgan Kaufmann, March 1988.

[50] D. Georgakopoulos, H. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3, 1995.

[51] C. Ghezzi and R.A. Kemmerer. ASTRAL: an assertion language for specifying real-time systems. In *Proceedings of the 3th European Software Engineering Conference*, LNCS 550, Milano (Italy), October 1991.

[52] M. Ginsberg. *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann, San Mateo, CA, 1987.

[53] T. M. Hagensen and B. B. Kristensen. Consistency in software system development: framework, models, techniques and tools. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, volume 17 of *Software Engineering Notes*. ACM, Tyson's Corner (Virginia - USA), December 1992.

[54] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering (ICSE14)*. IEEE Computer Society Press, 1992.

[55] W. S. Humphrey. *A Discipline for Software Engineering*. SEI Series in Software Engineering. Addison Wesley, 1995.

[56] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. Technical Report 95/15, Imperial College, London, UK, 1995.

[57] IEEE Computer Society. *IEEE Standard for Developing Software Life Cycle Processes (1074-1995)*, 1995.

[58] ISO/IEC. *International Standard, Information Technology Software Life Cycle Process. ISO12207*, 1995.

[59] M. Jackson. *Software Requirements & Specifications*. Addison-Wesley, 1995.

[60] M. Jackson. The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington (USA), April 1995. Keynote speak.

[61] P. Jamart and A. van Lamsweerde. A reflective approach to process model customization, enactment, and evolution. In *Proceedings of the Third International Conference on the Software Process (ICSP3)*, pages pp. 21–32, Reston, Virginia, October, 10-11 1994. IEEE Computer Society Press.

[62] A. Kaposi and M. Myers. *Systems, Models and Measures*. Springer-Verlag, 1994.

[63] B. Krishnamurthy and D.S. Rosenblum. An event-action model of computer-supported cooperative work: design and implementation. In *Proceedings of the International Workshop on Computer Supported Cooperative Work*, 1991.

[64] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986.

[65] V. Marek and M. Truszczynski. *Nonmonotonic Logic - Context-Dependent Reasonings*. Springer Verlag, 1993.

[66] Joao P. Martins and Stuart C. Shapiro. A model for belief revision. In *Artificial Intelligence*, volume 35, pages pp. 25–79. 1988.

[67] C. Mazza, J. Fairclough, B. Melton, D. De Pablo, A. Scheffer, and R. Stevens. *Software Engineering Standards*. Prentice Hall, 1994.

[68] R. Moore. Autoepistemic logic. In P. Smets, A. Mamdani, D. Dubois, and H. Prade, editors, *Non-Standard Logics for Automated Reasoning*. Academic Press, London, 1988.

[69] K. Narayanaswamy and N. Goldman. Lazy consistency: a basis for cooperative software development. In *Proceedings of the International Conference on Computer-Supported Cooperative Work*, Toronto (Canada), October 1992.

[70] B. Nuseibeh and A. Russo. On the consequence of acting in the presence of inconsistency. Unpublished manuscript, December 1997.

[71] O2. *The O_2 User Manual*. O_2 Technology, 1992.

[72] D. S. Rosenblum and B. Krishnamurthy. An event-based model of software configuration management. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991.

[73] I. Sommerville and T. Rodden. Human, social and organizational influences on the software process. In A. Fuggetta and A. Wolf, editors, *Software Process*, volume 4 of *Trends in Software*. J. Wiley, 1996.

[74] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[75] Heather Stark and Laurent Lachal. *Ovum Evaluates: Workflow*. Ovum ltd., September 1995.

[76] M. Strong and M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems*, 13(2), April 1995.

[77] Special issue on process evolution. *IEEE Transaction on Software Engineering*, December 1993.

[78] C. Tully. The software process and the modeling of complex systems. In *Proceedings of the 4th European Workshop on Software Process Technology*, LNCS 913, Noordwijkerhout, The Netherlands, April 1995. Springer-Verlag.

[79] Inconcert, inc home page. http://www.inconcert.com.

[80] Teamware home page. http://www.teamw.com.