Tolerating Deviations in Process Support Systems Via Flexible Enactment of Process Models

Gianpaolo Cugola
cugola@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci 32
20133 Milano (Italy).
Tel.: +39-2-23993666
Fax: +39-2-23993411

Abstract— **Process Support Systems (PSSs) support business organizations in modeling, improving, and automating their business process. Thanks to their ability in enacting process models, they can be used to guide people in performing their daily work and to automate the repetitive tasks that do not require human intervention. Given these potential benefits, it is surprising to observe that PSSs are not widely adopted. This is especially true in case of highly flexible and human-intensive processes such as design processes in general and software processes in particular. This fact can be explained by observing that currently available PSSs do not fulfill some crucial needs of modern business organizations. One of their major drawbacks is that they do not offer adequate mechanisms to cope with unforeseen situations. They are good at supporting business processes if all proceeds as expected, but if an unexpected situation is met, which would require to deviate from the process model, they often become more an obstacle than a help.**

This paper deals with the problem of managing unforeseen situations that require deviations from the process model during enactment in the context of the PROSYT PSS. During process model enactment, PROSYT is capable of tolerating deviations from the process model by supporting users even when unexpected situations arise. Furthermore, it supports users in reconciling the process model with the process actually followed, if necessary.

Keywords: Process Support System, WorkFlow Management System, Process-centered Software Engineering Environment, Business Process, Process Modeling, Process Enactment, Deviation, Inconsistency, event-based components integration.

I. Introduction

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. More flexible and dynamic organizations follow looser, adaptive process, but some notion of company-wide process still exists.

Business processes are important because they affect the quality of the products and services that the organization delivers to its clients. Ideally, an effective business process produces high-quality products in a cost effective way. These were the motivations behind the development of *Process Support Systems* (PSSs). PSSs aim at supporting organizations in improving and automating their business processes. Usually, a PSS provides a *Process Description Language* (PDL) [1], [2], [3], 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. Typical examples of PSSs are WorkFlow Management Systems (WFMSs) [2], [3] and Process-centered Software Engineering Environments (PSEEs) [1], [4], [5]. They offer tools to create process models, analyze them, and improve them according to the results of the analysis.

Process modeling and improvement 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. To support these organizations, advanced PSSs offer the chance to *enact* process models. During process model enactment, the PSS uses the rules and constraints expressed in the model to *automate* the activities that can be carried out without the intervention of human agents, and to *guide* and *support* people in carrying out the activities that require their intervention.

In principle, a PSS supporting process model enactment should be one of the most useful tools for advanced business organizations. It is therefore surprising to observe that they are not widely adopted in practice. A possible explanation is that the cost of formalizing complex business processes is usually high and the economic benefits of adopting a PSS to guide and automate a business process

have never been clearly demonstrated, at least in the case of highly flexible and human-intensive processes such as design processes in general and software processes in particular. But this does not fully justify the difficulties encountered in introducing PSSs in highly dynamic business organization. Perhaps, the 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 (the same consideration have been made in several papers in the area of WFMSs, like [6], [7]).

In general, 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. As a consequence, PSSs should be able to support people even when an unforeseen situation is encountered, which would force them to deviate from the process model. Unfortunately, currently available PSSs do not offer the appropriate mechanisms to cope with such situations. They provide several facilities to support and automate a business process when all proceeds as expected, but if an unforeseen situation is encountered, which requires a change in the expected sequence of activities, they often become an obstacle.

The problem of deviations in PSSs has been formalized in [8]. The Software Engineering group at Politecnico di Milano faced these problems by designing and implementing two PSEEs: SPADE [9] and SENTINEL [10]. In SPADE the problem of supporting users in facing unforeseen situations is solved by providing features to change the process model on-the-fly. This approach has proven to be effective to cope with major deviations from the process model that are expected to occur again in the future, but the effort required to change the process model makes the approach unsuitable to coping with situations that require minor, temporary deviations from the process model. SENTINEL was developed to help users in these cases, by tolerating deviations from the process model. The paper describes a further step in this direction: *PROSYT* (PROcess Support sYstem capable of Tolerating deviations). This paper focuses on the aspects of PROSYT that have a direct impact on its ability of tolerating deviations from the process model during enactment and supporting users in reconciling the process model with the actual process. Other innovative PROSYT features, like its ability of supporting nomadic users, are out of the scope of this paper.

The paper is organized as follows. Section II describes the problem of deviations in PSSs. Section III gives an overview of the mechanisms provided by PROSYT to manage deviations from the process model during enactment. Section IV describes the PROSYT PDL, focusing on the features that have an impact on the ability of tolerating deviations. Section V describes the architectural choices made to increase PROSYT's ability in tracking users' actions. Section VI describes how PROSYT users may deviate from the process model during enactment and how the PSS supports users in reconciling the model with the process actually followed after a deviation has occurred. Section VII compares PROSYT with other PSSs described in the literature focusing on the problem of managing unforeseen situations. Finally, Section VIII draws some conclusions and shows some directions for future research initiatives.

II. The problem of deviations in PSSs

This section analyzes the problem of supporting deviations from the process model being enacted by providing a motivating example in Section II-A. This example is then used throughout the paper to explain the concepts presented. Section II-B describes possible approaches to face unforeseen situations during enactment, and offers a more precise definition of the concepts of deviation and inconsistency.

A. A running example

The process example described in this section is taken from our daily work: it is the process of writing technical documents in a team using LaTeX [11]. It represents a business activity (several organizations are involved in developing documents in a cooperative fashion), which involves a group 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.

Further properties of the example process are:

- The process involves several people. One of them plays the role of *project leader*. The project leader is in charge of selecting the *authors*, deciding the organization of the document into 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 the other sections and contains the title of the document and the names and affiliations of the authors.
- After organizing the team and structuring the document, 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). By default each author is the owner of the sections he or she is responsible for. Owners edit the sections they own until they believe their status is 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 document may be delivered.

During the process of writing a technical document 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 document produced and delivered by the authors is satisfactory. He or she may decide to deliver the document without performing the review phase.

As another example, imagine that the deadline for the delivery of the document is getting close while the editing phase has not been completed yet. Imagine also that all the sections have been delivered with the exception of the final section (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. A deviation from the process model is required to cope with the unexpected delay in completing the editing. The next section analyzes in more detail the problem of deviations in PSSs and the possible remedies.

B. *Facing unforeseen situations, deviations, and inconsistencies*

As previously mentioned, most of the currently available PSSs 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 too rigid. It imposes a predefined behavior and does not allow users to cope with unforeseen situations. When an unforeseen situation arises, users have to exit the PSS to perform the required actions. This may result in an inconsistency between the state of the actual process and the state of the process as observed by the PSS, since the PSS cannot be aware of the actions performed by the users to cope with the unforeseen situation.

A possible solution to this problem, often adopted by workflow management systems [6], [7], [12], [13], [14], is to treat similar situations as *exceptions* and to provide exception handling techniques to cope with them. Unfortunately, this solution is not general enough. It allows users to cope only with a limited number of situations, captured by one of the exception handlers provided as part of the model. This explains why this situations are called *expected exceptions* in [7].

A different solution adopted by some PSEEs, including SPADE, is to supply mechanisms to support on-the-fly modification of the process model [15], [16]. By using these mechanisms, the project manager may change the process model during enactment, by explicitly modeling the unforeseen situation and the actions needed to cope with it. To support this solution, PSSs may incorporate meta-process facilities to describe and support the process of modifying an executing process. This can be obtained by introducing reflective features into PDLs [9], [17], [18].

Even this approach is often unacceptable, especially to support temporary, minor deviations. In fact, the activity of modifying the process model is time consuming and requires the project manager's intervention. Therefore, it is unreasonable to modify the process model to cope with unforeseen situations that are unlikely to occur again in the future and have limited impact on the overall process. In practice, these situations can be managed by performing the necessary actions out of the PSS control. Unfortunately, the problem with this naive approach is that these actions may result in an inconsistency between the state of the process as observed by the PSS and the actual state of the process, and this may invalidate any further support provided by the PSS.

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

A more precise (yet informal) characterization of the concepts of *process model*, *actual process*, and *observed process* is needed to understand how to support deviations in a PSS.

Process model. It is a model of the expected business process expressed 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 actual process, as it is performed in the real world. 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 action is performed).

Observed process. During process model enactment, a PSS has a partial view of the actual process. The PSS, in fact, is only aware of the actions that the users perform under its control. All other actions 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.

Using the terminology of [8], a *domain-level inconsistency* occurs when the actual process does not follow the process model. As an example, consider the process of writing a technical document described in Section II-A. If the project leader decides to deliver the document 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 document.

According to [8], an *environment-level inconsistency* arises when the observed process does not represent a correct view of the actual process. As an example, suppose that to deliver the document the project leader does not invoke the ad-hoc operation provided by the PSS, but simply prints the document and delivers it. If this happens,

the PSS cannot be aware of the fact that the document has been delivered and that the process has been completed. It would still expect 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 was modeled. The action of delivering the document without performing any review is an example of a 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 that is relevant for the business process out of the PSS control. The action described above of manually delivering the document by printing it is a typical example of an environment-level deviation.

A PSS is said to be *coherent* [8] if it is capable of tracking all the users' actions relevant for the process. This property reduces the chance that environment-level deviations may occur.

A *reconciling action* is an action whose goal is to reconcile the actual process with the process model (assuming they were inconsistent before executing the reconciling action). It is the opposite of a domain-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 the previous examples have shown, this is not always the case. When an unexpected situation occurs three different possibilities arise:

1. 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 that 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.
2. Before performing any action the process model is modified. A description of the actions that can cope with the unexpected situation is added to the model and the PSS is then 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 new process model. No deviations ever occur with respect to the new process model.
3. The process model is not modified, but the PSS offers the chance to deviate explicitly from the process model, by executing the actions necessary to cope with the unexpected situation under its control. Like in the previous case a domain-level deviation occurs, but here 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 that the deviation occurred and can analyze it and support its users in reconciling the actual and observed processes with the process model. The PSS *tolerates* the deviation.

In theory, case 2 seems to provide the safest and cleanest approach to cope with unforeseen situations. As mentioned before, however, it is often impractical, due to the effort required to modify the process model. More flexibility is often required in practice, especially in all cases where one needs to deviate temporarily. In such situations the approach of case 3 should be preferred. Unfortunately, most currently available PSSs (the SENTINEL prototype [10] is an exception) do not support this approach. The remainder of this paper shows how a PSS may offer this solution and discusses the advantages offered in terms of increased flexibility and usability of the PSS.

III. An overview of the PROSYT mechanisms to manage deviations

Supporting users in facing unforeseen situations, controlling that nothing bad happens, and tracking users' actions to help them during the reconciling activity are difficult and challenging tasks [19]. In [8] it has been shown that increasing the ability of a PSS to describe as many process situations as possible is mainly a *linguistic issue* while improving coherence is an *architectural issue*. Both these issues have to be pursued to reduce the chance that environment-level deviations might occur during process enactment. Moreover, there is a strong relationship among the mechanisms adopted at enactment-time to allow people to deviate from the process model, the PDL, and the effectiveness of the PSS in supporting people when unforeseen situations arise. These considerations have guided the design and implementation of the PROSYT PSS. More specifically, PROSYT offers the following main features:

- As for process modeling, most of the existing PDLs, can be classified as activity-based. Business processes are modeled in a top-down manner: 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. The artifacts produced during the process are described as inputs and outputs of activities.

Conversely, PROSYT adopts an *artifact-based* approach. Process models developed using the PROSYT PDL (called *PLAN*, the 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 under which operations are allowed to start. By using these expressions, the process designer may impose a partial ordering among the actions that characterize the business process if this is needed to constraint the expected sequences of process activities. Moreover, a PLAN model may include a set of invariants for each artifact. They are used to distinguish

between the acceptable and unacceptable sets of process states.

An artifact-based PDL presents advantages similar to the advantages provided by an object-oriented programming language. The latter helps software developers in identifying the basic operations that can be applied to data types. These operations depend only marginally on the specific program that is being developed. This increases the chance to reuse software modules and reduces the impact of changes. Similarly, an artifact-based PDL helps process modelers in identifying a set of basic operations that depends more on the type of artifact managed during the process than on the specific process adopted. The specifics of the business process adopted are embodied in the constraints that regulate the ordering of the operations. This distinction between a basic set of "general purpose" operations, which can be used both during standard operations and to cope with unforeseen situations; and a set of constraints that capture process specifics, constitutes the basis for the PROSYT approach to managing unexpected situations.

- As for process enactment, PROSYT users are not forced to satisfy the constraints stated in the PLAN process model. They can invoke operations on artifacts even if their constraints 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. This allows PROSYT users to cope with most unexpected situations. At the same time, as also underlined in Section VI-C, some situations are so complex to be solved that they necessarily require a modification of the process model. In particular, this is the case when users need to perform any operation that was not codified in the process model. An example of this kind of situation may be found in Section VI-C. Here it is important to observe that such situations occur very rarely in PROSYT because the artifact based approach to process modeling helps process designers in identifying the complete set of operations that may be invoked upon artifacts.

PROSYT may adopt different *deviation handling* and *consistency checking policies*. Such policies state the adopted level of enforcement (i.e., the classes of constraints that can be violated during enactment) and the actions that have to be performed when invariants are violated as a result of a deviation, respectively. Both these policies may change at enactment-time. The deviation handling policy may also vary from user to user. As an example, some deviations 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 that are forbidden to beginners.

- As for system architecture, in order to increase system coherence, PROSYT models are enacted in an open runtime 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. As explained in [20], the event-based communication paradigm increases dynamic reconfigurability of the system architecture and allows for an easy integration of existing tools, such as editors and other office-automation tools. This improves system coherence by increasing the ability of PROSYT to monitor and control the environment in which the process is carried out.

The next sections provide further details of the PROSYT approach to process modeling and enactment.

IV. Modeling process activities and constraints: PLAN, the Prosyt LANguage

Since, the focus of this paper is how to support deviations from a specified process model during enactment, here we deal only with the features of PLAN that are relevant to tolerating deviations and supporting users in facing unforeseen situations. In particular, this section describes the mechanisms adopted by PLAN to allow process designers to separate the description of process activities from the description of process constraints and shows how the latter are organized in different classes to support an accurate control of deviations.

As mentioned, PLAN is an artifact-based PDL. Artifacts are defined by a set of operations and a set of constraints. PLAN also provides operations and constraints that refer to a collection of artifacts through the notions of *folder types*, *repository types*, and *project types*. The overall structure of these types is similar: they describe the internal structure, the behavior, and the constraints that characterize their instances.

A detailed description of artifact types, folder types, repository types, and project types is provided next.

A. Artifact types

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

$$\langle attrs, states, exp_ops, auto_ops, methods, invariants \rangle$$

Attrs. It is the set of attributes that determine the *concrete state* of each instance of the artifact type. Each attribute is defined by using the same syntax adopted by Java for object attributes[1]. As an example, see the attributes `owner`, `manager`, and `edited` in Figure 1.

States. This is an ordered set of *logical state* definitions. Each logical state is defined by a pair: $\langle identifier, predicate \rangle$. 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 set of concrete states that compose the logical state. As an example, the description of the artifact type `Source` may include two attributes: `editTime` and `compileTime` and two logical states: `toBeCompiled` and `compiled` characterized by the two predicates "`editTime > compileTime`" and "`editTime <= compileTime`", respectively.

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,

[1] PLAN models are automatically translated in Java to be enacted.

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. If none of the predicates is satisfied, the logical state of the artifact is said to be *partially inconsistent*.

Exp_ops 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 defined by a 6-tuple:

$\langle name, formal_pars, agents, guard, precondition, body \rangle$

- The name and formal parameters have the usual semantics as - say - method names and parameters in Java. The type of formal parameters, however, is left unspecified in PLAN. 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 a list of users and/or groups that determines the users who are authorized to invoke the operation. If it is not specified, any user may invoke the operation. An example can be found in exported operation `changeOwner` in Figure 1.
- The guard is a sequence of *guard entries* which, in turn, are pairs: $\langle identifier, predicate \rangle$. 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. If the predicate is omitted the default value `true` is assumed. The operation may be executed if the logical state of the artifact is one of the logical states mentioned in the operation's guard and if the corresponding predicate is satisfied. If the guard is entirely omitted the operation can be executed in any state (as an example, see exported operation `show` in Figure 1).
- The precondition is a boolean expression on the values of the artifact's attributes and operation parameters. If it is omitted, the default value `true` is assumed. The difference between the guard and the precondition is that the latter describes a condition that does not depend on the logical state of the artifact. It must hold before the exported operation is executed, no matter what the state of the artifact is. As a result, usually preconditions describe more critical requirements for the operations than guards. As an example of precondition see operation `changeOwner` in Figure 3.
- Finally, the body is a sequence of Java statements that describe the actions performed when the exported operation is invoked. A set of predefined operations (i.e., methods) is provided to perform basic operations, such as launching a tool, creating a new artifact, and so on. The complete set of these operations can be found in [21].

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

Auto_ops. These operations 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 a case, the `initialize` operation is automatically called),
- invocation of exported operations,
- start of execution of exported operations, and
- end of execution of exported operations.

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`. The actions performed by automatic operations are described in Java. Several examples of automatic operations may be found in Figure 3.

Automatic 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 procedures are local to the artifact where they are defined. They are mainly used to simplify the implementation of the artifact's exported and automatic operations. They are also visible to the artifacts that inherit from the artifact where they are defined. They are similar to Java "protected" methods.

Invariants. They 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 are no predefined labels). By using labels, the process designer can define as many types of invariants as needed.

As I will describe in Section VI, at enactment-time different actions may be associated with the violation of different types of invariants. For example, violating invariants of type `weak` may result in sending a message 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 1 shows the PLAN description of the artifact type `TextDocument`. As mentioned in Section II-A, 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 document. During initialization of a new text document (operation `initialize`), the artifact's icon (used by the "projectBrowser", i.e., by the PROSYT user interface) is chosen and the value of the `manager` and `owner` attributes are initialized. Moreover, a file having the same name as the artifact with the extension "`tex`" is associated with the text document under the logical name "source". This name may be used to refer to

```
artifactType TextDocument {
attributes:
  String owner, manager;
  boolean edited=false;
states:
  readyToBeEdited : !edited;
  edited : edited;
exportedOperations:
  changeOwner(newOwner) to users manager groups ProjectLeaders;
  {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");
  }
}
```

Fig. 1. The artifact type `TextDocument`.

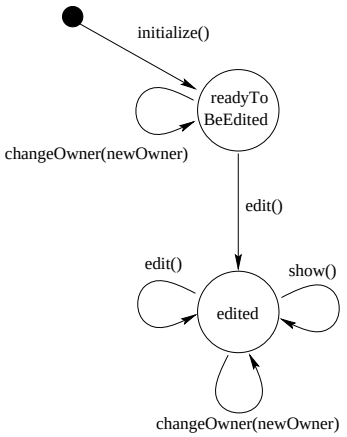

Fig. 2. The FSM which describes the behavior of text documents.

```
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="";}
}
```

Fig. 3. The artifact type `Section`, a sub-type of `TextDocument`.

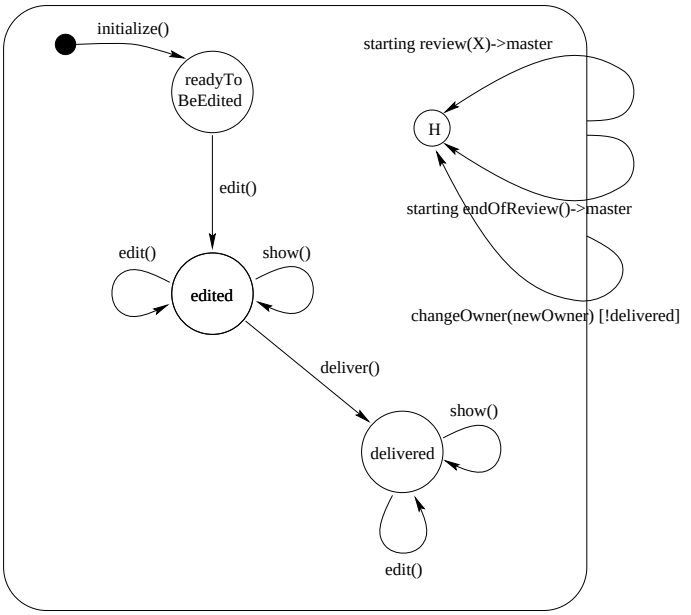

Fig. 4. The FSM that describes the behavior of sections.

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 body of these operations shows how tools can be invoked. Observe that the current version of the PROSYT prototype invokes tools directly by using their name. In the future we plan to add a mechanism to have an indirect binding between tools and the logical operation they perform. Through this mechanism it will be possible to call an "editor" which will be bound to different editors depending on the environment in which the user operates (e.g., notepad under Windows and Emacs under Unix).

The finite-state machine in Figure 2 formalizes the behavior of text documents, which is determined by the guards of its exported operations.

Before concluding this section it is important to mention the fact that PLAN is an object-oriented language that supports single inheritance. Artifact types may inherit from other artifact types and may add new attributes, new states, new operations (both exported and automatic), new methods, and new invariants, if necessary. Moreover, the sub-types of an artifact type T may redefine the states, operations, and invariants of T. A detailed description of the semantics of inheritance in PLAN is out of the scope of this paper. The interested reader may refer to [21].

As an example of inheritance in PLAN, Figure 3 shows the PLAN definition of the artifact type `Section`, a sub-type of the artifact type `TextDocument` shown in Figure 1. The finite-state machine that formalizes the behavior of this class of artifacts is shown in Figure 4.

B. Folder, repository, and project types

To describe activities and invariants that refer to a collection of artifacts PLAN provides the concepts of *repos-*

itory and *folder*. Each repository is an instance of some *repository type* and contains a set of *folders* organized in a tree structure. Each folder (instance of some *folder type*) is a container of artifacts and other folders. A folder must belong to some repository. Repositories are analogous to partitions of a file system, while folders are analogous to directories.

Attributes, states, exported operations, automatic operations, methods, and invariants may be associated with repository types and folder types using a syntax similar to that adopted for artifacts. Exported operations and invariants are used to describe business activities and invariants that 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 introspective 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>)` applied to a folder F returns true if all the artifacts belonging to F whose type is "`<artifact type>`" are in the logical state "`<state>`". The complete set of these predicates can be found in [21].

Finally PLAN provides the notion of a *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 operations. Groups are relevant to managing deviations because the level of enforcement may change from group to group.

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, and both the guard and the precondition hold, then the process constraints are satisfied upon completion[2].

Due to the complexity of the PLAN language, correctness cannot be checked mechanically. It is the responsibility of the process designer to check model correctness. If an incorrect process model is enacted, it may happen 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 it is possible to continue process enactment even if some invariants are violated.

Figure 5 uses PLAN to model the activity of writing a document described in Section II-A. It defines the project

[2]Both exported and automatic operations are atomic.

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

repositoryType DocumentRep {
attributes:
  boolean documentFolderCreated=false;
states:
  readyToCreateDocument : !documentFolderCreated;
  readyToStartEditingPhase : documentFolderCreated;
exportedOperations:
  createDocument(name) to groups ProjectLeaders;
  guard readyToCreateDocument;
  {
    createNewFolder("DocumentFolder", name);
    createNewFolder("PictureFolder", name+".pict");
    documentFolderCreated=true;
  }
automaticOperations:
  initialize()
  {enableGroup("ProjectLeaders"); enableGroup("Authors");}
}

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

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

Fig. 5. The DocumentWriting process model.

type that describes such process, together with the repository and folder types needed to model the process, are defined there. Figure 6 shows the PLAN definition of the artifact type `MasterDocument`; artifact types `Section` and `TextDocument` were defined in Figures 3 and 1, respectively.

The project type `DocumentWriting` defines two groups (`ProjectLeaders` and `Authors`), a repository of type `DocumentRep` that holds the document 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 document. The automatic operation `initialize`, called when a new instance of the repository type `DocumentRep` is created, uses the procedure `enableGroup` (which is part of the PROSYT API) to make the repository visible to the project leader and to all the authors.

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

```
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 ProjectLeaders;
  guard readyToPrint and revisor.equals("") && allSectionsDelivered();
  {revisor=revisorName;}
  endOfReview() to users revisor;
  guard readyToPrint;
  {revisor="";}
  deliver() to groups ProjectLeaders;
  guard readyToPrint and revisor.equals("") && allSectionsDelivered();
  {delivered=true;}
  addSection(sectionName, manager) to groups ProjectLeaders;
  {
    Object[] o=new Object[2]; o[0]=manager; o[1]=this;
    sections.put(createNewArtifact("Section", sectionName, o),
                 "not delivered");
  }
  edit() to groups ProjectLeaders 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;
  }
}
```

Fig. 6. The artifact type `MasterDocument`.

ument.

When the document folder is created a new artifact of type `MasterDocument` is added to the folder (see the automatic operation `initialize` of the folder type `DocumentFolder`). It represents the master document described in Section II-A. Several exported operations may be invoked upon the master document. They are used to automate the basic activities that compose the process of writing a document. In particular, it is possible to add new sections, edit the master document, show its content, compile the document (i.e., generate the postscript file of the entire document from the LaTeX sources), print the document, 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 that characterize the expected process defined in Section II-A.

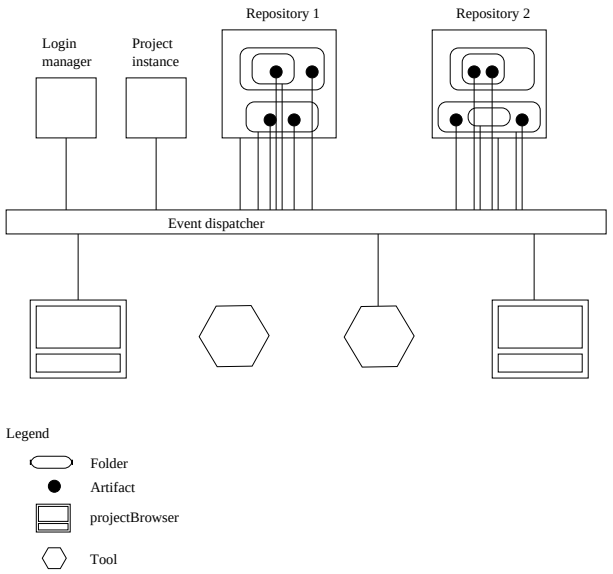

Fig. 7. The PROSYT run-time architecture.

V. Improving coherence: the PROSYT architecture

As mentioned in Section III, in order to avoid the occurrence of environment-level deviations, a PSS must adopt an architecture that maximizes system coherence. PROSYT achieves this goal by adopting an event-based architecture that facilitates tools integration. Users perform most of the actions relevant to the process through tools. Integrating these tools into the PSS increases the system ability of tracking user actions, thus reducing the chance that actions relevant for the process are performed out of the PSS control.

The run-time architecture of PROSYT is composed of a set of active, distributed components, which interact by delivering and receiving events. Events are particular types of messages. While conventional messages may be sent from a sender to one or more recipients, as specified by the sender, events do not include any information about their recipients. An event is generated by a component and automatically sent to all the components that declared an 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.

To support event dispatching, PROSYT takes advantage of the services provided by the JEDI distributed infrastructure [20]. The resulting PROSYT run-time architecture is described in Figure 7. It is composed of the following distributed, active components:

- A login manager, which is in charge of controlling users' login and logout. It manages all the information about the human agents that are involved in the process.
- 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 bound to a particular host during the deployment step.
- A set of folders. Each folder belongs to a repository. Folders may move from a repository to another during enactment. If a folder F moves from a repository α to a repository β then all the folders and artifacts contained in F 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 a host to another. This is obtained by exploiting the features provided by the JEDI infrastructure to support the implementation of mobile applications [22].

- 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.
- 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 logged-in user, which provide user interfaces 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 8 shows the projectBrowser in action during enactment of the `DocumentWriting` project.
- 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 (in Figure 7 integrated tools are modeled as boxes connected to the event dispatcher while non-integrated tools are not connected). In the former case tools can be fully controlled by PROSYT. In the latter, they can only be started and ended. The higher the number of fully integrated tools is, the better the system can track users' actions, reducing the occurrence of environment-level deviations and increasing system coherence.
- An event dispatcher, which is in charge of delivering events to the interested components. The event dispatcher is part of the JEDI infrastructure.

Each of the above components is active, that is, each of them has its own thread of control.

When a user invokes an exported operation E upon an object O (i.e., an artifact, a folder, a repository, or a project) 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.

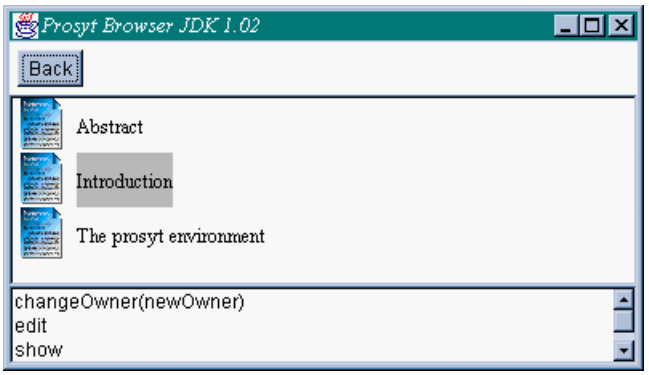

Fig. 8. The projectBrowser during enactment of the `DocumentWriting` project.

VI. Tolerating deviations: the PROSYT enactment mechanism

In order to support process deviations, PROSYT allows users to execute an exported operation even if the constraints that state the conditions for its correct execution (i.e., the constraint on the users allowed to invoke it, its guard, and its precondition) are violated.

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

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

A. Deviation handling and consistency checking policies

Four types of deviations are defined in PROSYT:

User deviation. An exported operation is executed by a user who is not allowed to do so.

Condition deviation. An exported operation E is executed upon an object whose current state is listed in E's guard, but whose corresponding predicate is not satisfied.

State deviation. An exported operation E is executed upon an object whose current state does not appear in E's guard.

Precondition deviation. An exported operation is executed when its precondition is not satisfied.

In general, a user deviation is less critical than a condition deviation, which is less critical than a state deviation,

which, in turn, is less critical than a precondition deviation. Obviously, this is not always true, and depends on the specific 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 of the state of the object (i.e., artifact, folder, repository, or project) they belong to, while guards depend on the object's state.

The *deviation handling policy* specifies the actions to perform when a user invokes an exported operation whose execution results in a deviation. For each deviation type and for each user, one among five different policies may be specified:

Abort The operation is not executed at all.

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

Ask user The user is informed that executing the operation results in a deviation and he or she is prompted to confirm the invocation.

Inform user and continue The user who invoked the operation is informed that executing the operation results in a deviation but the operation is executed anyway.

Continue The operation is executed without even informing the user that he or she is deviating from the model.

A deviation handling policy that adopts the "abort" policy for deviations of type T and user U means that U is not allowed to perform a deviation of type T from the prescribed process. As an example, suppose U invokes an operation O upon an artifact A. Suppose also that O's precondition is not satisfied and that policy "abort" is specified for user U and precondition deviations. When U invokes operation O through the projectBrowser tool, an event is generated to notify the artifact A that the operation O has been invoked, but artifact A simply ignores this event. In addition to that, the "inform user and abort" policy would notify the user that the operation cannot be executed. Conversely, the "ask user" policy would prompt the user to decide if he or she really wants to deviate. Finally, "inform user and continue" and "continue" policies would allow users to deviate from the process model without requiring any further confirmation.

If executing an exported operation results in violating more than one constraint (i.e., if the resulting deviation belongs to more than one type) the strongest policy is adopted. As an example suppose that an unauthorized user U executes an exported operation whose precondition is not verified. Suppose also that the deviation handling policy for U specifies the "abort" policy for precondition deviations and the "ask user" policy for user deviations. In such case the "abort" policy is adopted.

PROSYT allows the deviation handling policy to be changed for each user at enactment-time by using an ad-hoc tool. This powerful feature offers the process manager an opportunity to adopt the deviation handling policy dynamically, choosing the level of enforcement that is most appropriate for each situation.

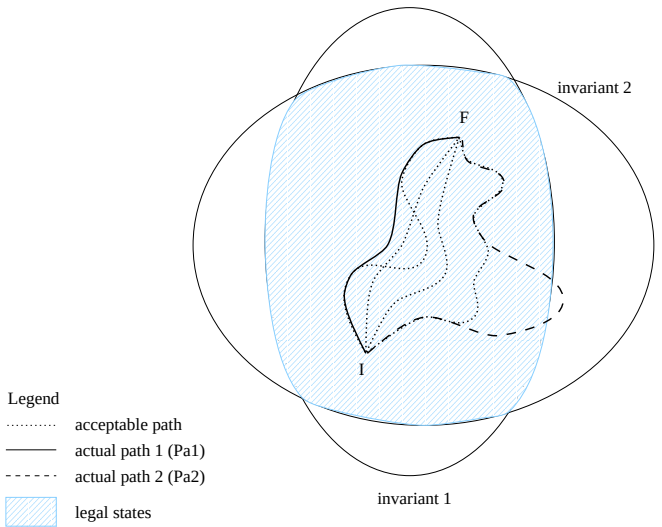

Fig. 9. A graphical description of the constraint stated by a PLAN model.

PROSYT provides also a feature to control that the state of the system does not become unacceptable as a result of deviating from the process model. This feature is based on the use of invariant predicates and on the possibility of specifying different *consistency checking policies*.

By specifying a set of invariants for artifacts, folders, repositories, and projects, the process designer defines the boundaries which distinguish the set of states of the system considered to be acceptable from those considered to be unacceptable. Intuitively, this fact can be described as in Figure 9[3]. The figure describes the space of process states. A business process can be viewed as a path from an initial state I to a final state F. A PLAN model describes a *state space* and a set of *acceptable paths* in such space from I to F. Such paths are described by the set of basic actions (i.e., the exported operations) defined in the process model, and by the constraints (agents, guards, preconditions) that characterize them, which specify the correct execution sequences. Each invariant defines a subset of process states, which contains all the states that satisfy the invariant. The intersection of all such subsets, corresponding to all the invariants defined for the PLAN model, is said to be the set of *legal process states*. As we said, all acceptable paths of a correct PLAN model are guaranteed to consist only of legal process states.

When a deviation occurs, the path that describes the actual process diverges from any of the acceptable paths described by the model. Depending on the severity of the deviation, the path may also cross the boundary of the set of legal process states. This is exemplified by path $Pa2$ in Figure 9. This suggests an approach to controlling deviations based on the detection of the invariant that is violated during the deviation. PROSYT adopts this approach by giving

[3]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. For a formal description of these concepts see [8]

the process manager the ability of defining and changing the consistency checking policy at enactment-time[4]. It determines the action that the system has to perform when an invariant is violated. For each type of invariant one of the following predefined policies may be chosen, which correspond to different levels of severity of the violation:

Stop enactment. The process is stopped and the process manager can start the reconciling activity. This is a safe option but severely constrains the freedom 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 exported 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 a portion of the process is frozen until the consistency is restored; the remaining part can continue. To solve inconsistency, the process manager may start the reconciling activity on the isolated portion of the process.

Inform process manager and continue. A message is sent to the process manager to inform her or him that an invariant has been violated. The message includes information regarding the occurred deviation. Process enactment continues.

Continue. Violation of the invariant is ignored. In general, adoption of this action is very dangerous: the process manager does not have any means 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, this may occur when the process manager decides that an invariant is too restrictive but he or she does not want to change the process model.

Notice that PROSYT 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, to make this behavior possible, the enactment of the artifact must continue even if an invariant has been violated (i.e., a consistency checking policy, other than "stop enactment" and "stop object enactment", has to be chosen). This behavior constitutes the basis of a strategy to restore consistency, as described in the next section.

B. The reconciling activity

The reconciling activity is responsible for restoring consistency between the actual and observed processes (which are supposed to be consistent) and the process model.

The approach adopted by PROSYT to support users during the reconciling activity moves from the consideration that reconciliation is an activity that requires human intervention. It depends on a number of conditions that are not known to the PSS, such as 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 actual process. The PSS can only help the process manager in analyzing the occurred deviation to choose the best strategy to achieve reconciliation.

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

1. modify the process model 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 state to a consistent one.

Obviously a mixed approach may be also chosen.

As mentioned PROSYT adopts the second strategy. When an invariant is violated the process manager has several choices:

- If either the "inform process manager and continue" or the "continue" consistency checking policies have been adopted, the enactment continue until, hopefully, the natural evolution of the process restores consistency.
- If the violation of the invariant stops the 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.
- Finally, the process manager may use the tools provided by PROSYT to access the artifacts, folders, repositories, and project objects, changing the values of their attributes in order to restore consistency. These actions may be combined with actions that 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, PROSYT may display a history of the operations called by users upon artifacts from the first deviating action until the current time instant. 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 this information the process manager may get a clear view of the occurred deviation and may decide which approach is best to reconcile consistency.

C. Dealing with unexpected situations in PROSYT: two examples

As an example of an unexpected situation that may occur during the example process of Section II-A imagine that an author A other than the project leader, needs to start the process of writing a document. To perform this action, the author A must invoke the exported op-

[4]The consistency checking policy may be changed through the same tool used to modify the deviation handling policy

eration `createDocument` upon the document repository. In principle, authors are not allowed to invoke this operation. PROSYT, however, allows this operation to be executed, assuming that a deviation handling policy other than the `abort` and `inform user and abort` is chosen for user deviations, for user A. This deviation is recorded by PROSYT and the information regarding it is presented to the process manager, if necessary.

As a second example, to start the review phase when some section has not been delivered yet, the project leader must invoke the operation `review` on the master document even if the predicate `allSectionsDelivered` is false. If the deviation handling policy adopted for the project leader permits it, reviewing phase can be started any how. On the other hand, a strict consistency checking policy (e.g., the `stop enactment` policy) would prevent the reviewer from delivering the document until the editing phase is concluded. In the example, executing the operation `deliver` on 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.

Observe that some unexpected situations are so complex to be managed that the PROSYT approach is not adequate. They would require an on-the-fly modification of the process model to be efficiently solved. As an example imagine that a user needs to delete a section S because it was erroneously created. The PLAN model of our example does not provide any `delete` exported operation and thus the user cannot delete Section S under the PROSYT control. The best way to cope with this situation would be to add the delete operation to the PLAN model on-the-fly.

VII. Related work

The problem of tolerating deviations and inconsistencies in PSSs has not been studied satisfactorily in the past. [23] and [24] provide two different attempts to define a framework for dealing with process evolution in the context of PSEE. In particular, [23] provides a first distinction among *process definition*, *process performance*, and *process definition enactment*. Process definition corresponds to what we called process model, process performance corresponds to the actual process, and process definition enactment corresponds to the observed process. The possibility for the process performance to deviate from the process definition is recognized, but mechanisms for dealing with this problem are not discussed.

[25] and [26] deal with the inconsistencies caused by the uncertain and incomplete knowledge in the observed processes. A goal-oriented language PEACE is defined, which formalizes parts of a process model using an autoepistemic logic, which allows users' beliefs to be modeled, and supports the reasoning about the differences between the user beliefs and the actual process. 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 transition 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 have no 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 possibly face a particular situation has not been described, the users can try to use (i.e., invoke) the existing operations, violating the constraints embedded into the process model, to cope with the unexpected situation.

In the Provence PSEE [27], the process model is not used to automate the process. Provence uses the model to verify that the actual process follows the specified behavior. Provence is based on an event-action monitor called Yeast [28], [29], which provides mechanisms to detect significant events occurring in the actual process and to trigger specific actions that have been associated with those events. Provence monitors the execution of the actual 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 PSS uses the information provided by Yeast to define the actual state of the process and to determine whether it is consistent with the process model. The Provence approach to identifying deviations and to increasing coherence is interesting, but the environment does not offer any explicit support to its users in facing unexpected situations.

SPADE [30] is a Process-centered Software Engineering Environment (PSEE) that supports software process analysis, design, enactment, and evolution. SPADE provides an activity-based PDL, called SLANG, which is based on high-level Petri nets. As the majority of existing PSEEs, SPADE assumes that the humans involved in the development process do not change the way they work unless they change the process model (using the process evolution mechanisms offered by the PSEE). SPADE offers an extended support to process evolution through the reflectivity features of the SLANG PDL [9]. Process models may be modified as any data by other processes (meta-processes). As mentioned, this approach is safe and clean but is not suitable to coping with temporary deviations due to the effort required to modify the process model.

OASIS [17] provides an object-oriented, reflective framework for the definition, customization, and evolution of software process meta-models and of the software process models that are their instances. In developing this framework, the authors start from the consideration that every process modeling approach relies on some specific set of abstractions that define a process meta-model. They observe that the ability to provide a uniform model of 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 defining 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 the kernel classes using an object-oriented definition toolkit. Classes at one level are 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) to be modeled as any other process. This is a very powerful approach to manage process model evolution and can be used to cope with unexpected situations that require major changes on the process model. At the same time, as observed in Section II-B, the drawback of this approach is that modifying the process model is a time consuming activity. 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 more suitable to coping with minor deviations.

Similar considerations are valid also for Endeavors. Endeavors [18] is an open, distributed, extensible process support system infrastructure, which allows the object-oriented definition and specialization of *activities*, *artifacts*, and *resources* associated with a software development process. Endeavors' *activity networks* define the interrelationships 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.

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 run-time. 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.

The SENTINEL PSEE [10] was the result of a preliminary exploration into the problem of improving the usability of PSSs by tolerating deviations from the process description. PROSYT was designed and developed by taking advantage of the experience gained by implementing and using SENTINEL. The SENTINEL environment provides an activity-based process modeling language called LATIN. Processes are modeled in LATIN as collections of task types. Each task type describes a process activity as a state 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. 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, we say that the transition *fires illegally*. By forcing exported transitions to fire illegally, users are able to deviate from the process model in order to cope with unexpected situations. SENTINEL records the 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. The enactment is suspended only if one of the invariants that define the critical requirements of the process is violated.

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 that are most 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 four main advantages over the SENTINEL client-server architecture:

1. existing tools may be easily integrated into the environment and this helps in achieving higher system coherence, which is a fundamental quality for PSSs.
2. 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;
3. enacting process fragment can be moved from host to host to reduce network traffic and to support nomadic users;
4. components can be added or removed at run-time, and they can be moved from host to host. That is, the architecture of the running system can be changed dynamically

to follow the requirements of the process to be supported;
• 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.

Finkelstein et al. [31] deal with inconsistency in the domain of multi-perspective software specifications. A software specification is captured by means of a set of *ViewPoints*. Each ViewPoint is the specification of part of the system, seen from a particular domain. ViewPoints are related among them by *inter-ViewPoint* relations, that "...specify dependencies and mappings between system components". Moreover, each viewpoint holds a definition of the process that has to be carried out to complete the specification and a definition of the process that has to be followed when one of the inter-ViewPoint relations is violated. These *meta-level rules* are formalized in a temporal logic and can be invoked to perform the repairing actions. Their use implies the knowledge of the history of the data in the ViewPoints, and of the previous repairing actions on the inconsistent data. PROSYT does not provide explicit meta-level rules. We assume that the activity of reconciling cannot be defined in advance and requires the human intervention. PROSYT provides the users with all the necessary information to analyze the deviation occurred and to modify the process state and the process model as needed.

Several workflow management systems support users in facing unforeseen situations by allowing them to change the enacting process model on-the-fly. Examples of such WFMSs are InConcert [32] and TeamWare Flow [33]. 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 enacting 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 designers 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. The former is more suitable 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 that require an immediate answer and that are known to occur sporadically.

Several workflow management systems [2], [3] provide mechanisms to cope with unexpected situations via exception handling techniques [12], [13], [14]. 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 a WFMS that provides an explicit support to exceptions is WIDE. WIDE uses an activity-based PDL [34] 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. A WIDE model is composed of an organizational model and a process model. The organizational model describes the structure of the organization that carries 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 jump forward or backward in the task flow. The reaction part specifies the actions that have to be carried out when the exception is identified.

A similar approach is taken by OPERA [35] to handle several forms of deviations ranging from user deviations resulting from the need of solving unexpected situations to program and system failures in the underlying environment.

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 (the same consideration is made in [7]). 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 is "implicit". It does not require the process designer to provide a description of the exceptions that may occur, nor the procedures to handle them in advance. Deviation handling is left to the users who may benefit of the PSS ability of tolerating deviations during enactment. Users may change the order in which operations have to be executed. This, together with the fact that the artifact based approach results in process models composed of simpler and more general operations than the operations identified using an activity based approach, seems to be an effective mechanism to deal even with "unexpected" exceptions.

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 [36], IEEE 1074 [37], and PSS-05 [38]. 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. Recently, there has been much interest in adopting such standards in industry for certification purposes. Adopting these standards results in a new kind of problems: once a standard has been adopted, it is important to manage compliance of the actual software process with the standard. [39] describes the problem of managing standards compliance

in software development (in particular, during the phases of software requirements analysis and specification) and presents an approach to automate this activity. This approach is based on the use of DOORS (Dynamic Object Oriented Requirements System), a generic document management system. To implement standards compliance policies, the facility provided by DOORS to attach user-defined actions to the occurrence of special events has been used. In particular, standards are translated into a set of rules and deviations from these rules are trapped by DOORS and signalled through events. Actions are associated to the occurrence of such events in order to describe the activities that have to be carried out when a violation to the standard arises.

This approach is similar to the approach adopted by PROSYT, but there are many differences too. First, PROSYT allows a more precise definition of the process model: the structure of the artifacts may be specified together with the operations that may be executed upon them and with the invariants that characterize system consistency. Conversely, in the approach presented in [39], there is no a precise definition of the process that must be adopted. Secondly, by adopting the approach described in [39] it is not possible to distinguish between violating operation preconditions and moving outside the borderline determined by invariants. Thirdly, PROSYT policies may be changed at run-time and may differ from user to user, while the behavior of DOORS when a deviation from the standard is identified is defined statically. Finally, the PROSYT environment is more generic. It 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, such as the software requirements analysis process.

VIII. Conclusions and future work

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. One of the major problems of current PSSs is their lack of flexibility and, in particular, their inability to support 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 escape from the PSS control 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 paper we described PROSYT, a PSS capable of tolerating deviations from the process model during enactment. It features effective mechanisms to help users in facing unexpected situations; it continues to handle process support and automation even during deviations, and supplies all the information needed to reconcile the process model and the process actually followed. Deviations can be tolerated as long as they do not affect the correctness of the system.

One of the key features of PROSYT is its ability to modify the level of enforcement adopted and 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 level of enforcement adopted on a per-user basis; i.e., different users may be allowed to perform different kinds of deviations.

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 real business processes. Future work will be devoted in trying to experiment PROSYT in an industrial environment. This experience will allow us to assess the environment.

Another area which is worth investigating in future 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 managers 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, and suggest them to the process manager to (partially) automate the reconciling activity.

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

ACKNOWLEDGEMENTS

The author wishes to thank Carlo Ghezzi for his important contribution to the accomplishment of the work described in this paper. I wish also to thank Elisabetta Di Nitto and Alfonso Fuggetta who worked with me on the topic of deviation handling in PSSs and who give me useful suggestion to design PROSYT. I wish also thank all the people involved in the Promoter subgroup on inconsistency management: Wolfgang Emmerich, Anthony Finkelstein, Michael Goedicke, Tony Hunter, Jeff Kramer, Axel van Lamsweerde, Carlo Montangero, Bashar Nuseibeh, Cristophe Ponsard, Alessandra Russo, George Spanoudakis, and David Till. The discussion we had during our meetings provided me with a lot of ideas that I exploited in developing PROSYT. Finally, I wish to thank the anonymous referees for their careful reading and the constructive criticisms.

References

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

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

[3] H. Stark and L. Lachal, *Ovum Evaluates: Workflow*. Ovum ltd., September 1995.

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

[5] V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing process-centered environments," *ACM Transactions on Software Engineering and Methodology*, vol. 6, July 1997.

[6] G. Alonso and C. Mohan, "Workflow management systems: The next generation of distributed procesing tools," in *Advanced Transaction Models and Architectures* (Jajodia and L. Kerschberg, eds.), ch. 1, pp. 35–62, Kluwer Academic Publishers, 1997.

[7] J. Eder and W. Liebhart, "The workflow activity model WAMO," in *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS)*, (Vienna, Austria), May 1995.

[8] 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)*, vol. 5, July 1996.

[9] S. Bandinelli, A. Fuggetta, and C. Ghezzi, "Process model evolution in the SPADE environment," *IEEE Transactions on Software Engineering*, vol. 19, December 1993.

[10] 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.

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

[12] H. T. Saastamoinen and V. V. Savolainen, "Exception handling in office information systems," in *Proceedings of the Third International Conference on Dynamic Modeling of Information Systems*, (Noordwijkerhout), pp. 345–363, 1992.

[13] H. T. Saastamoinen, "Rules and exceptions," in *In Information Modeling and Knowledge Bases IV: Concepts, Methods and Systems* (H. Kangassalo, H. Jaakkola, K. Hori, and T. Kitashi, eds.), pp. 271–286, Amsterdam: IOS Press, 1993.

[14] *Proceedings of the EDBT Workshop on Workflow Management Systems*, (Valencia, Spain), March 27-28 1998.

[15] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proceedings of COOCS 95*, August 1995.

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

[17] 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)*, (Reston, Virginia), pp. 21–32, IEEE Computer Society Press, October, 10-11 1994.

[18] G. A. Bolcer and R. 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] 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.

[20] 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.

[21] G. Cugola, *Inconsistencies and Deviations in Process Support Systems*. PhD thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, Feb 1998.

[22] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna, "Analyzing Mobile Code Languages," in *Mobile Object Systems: Towards the Programmable Internet* (J. Vitek and C. Tschudin, eds.), LNCS 1222, pp. 93–111, Springer, April 1997.

[23] 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.

[24] 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.

[25] S. Arbaoui and F. Oquendo, "Peace: Goal-oriented logic-based formalism for process modelling," in *Software Process Modelling and Technology* (A. Finkelstein, J. Kramer, and B. Nuseibeh, eds.), Research Studies Press, J. Wiley, 1994.

[26] 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.

[27] N. 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.

[28] B. Krishnamurthy and D. 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.

[29] 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.

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

[31] 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), Springer-Verlag, September 1993.

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

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

[34] 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*, (Goldcoast, Australia), pp. 341–354, 1995.

[35] G. Alonso and C. Hagen, "Flexible exception handling in the opera process support system," in *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS 98)*, (Amsterdam, The Netherlands), May 1998.

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

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

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

[39] 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), IEEE CS Press, 1997. electronic publication.