A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems

Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta and Carlo Ghezzi
{cugola, dinitto, fuggetta, ghezzi}@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci, 32
20133 Milano (Italy).
Tel.: +39-2-23993638
Fax: +39-2-23993411

May 22, 1996

Abstract

Most modern business activities are carried out by a combination of computerized tools and human agents. Typical examples are engineering design activities, office procedures, and banking systems. All these *human-centered systems* are characterized by the interaction among people, and between people and computerized tools. This interaction defines a process, whose effectiveness is essential to ensure the quality of the delivered products and/or services. To support these systems, process-centered environments and workflow management systems have been recently developed. They can be collectively identified with the term *process technology*. This technology is based on the explicit definition of the process to be followed (the *process model*). The model specifies the kind of support that has to be provided to human agents.

An essential property that process technology must exhibit is the ability of tolerating, controlling, and supporting *deviations* and *inconsistencies* of the real world behaviors with respect to the process model. This is necessary to provide consistent and effective support to the human-centered system, still maintaining a high degree of flexibility and adaptability to the evolving needs, preferences, and expertise of the human agents.

This paper presents a formal framework to characterize the interaction between a human-centered system and its automated support. It does not aim at introducing a new language or system to describe processes. Rather, it aims at identifying the basic properties and features that make it possible to formally define the concepts of inconsistency and deviation. This formal framework can then be used to compare existing solutions and guide future research work. Finally, we argue that our approach provides a general and reference framework to characterize the interaction between a computerized system and the (parts of) the real world it is supposed to describe and possibly control.

Categories and subject descriptors: D2.6 [Software Engineering]: Programming Environments; H1.2 [Models and Principles]: User/Machine Systems - *Human Factors*; K6.3 [Management of computing and information systems]: Software Management - *Software Development, Software maintenance.*
General Terms: Human Factors.
Additional keywords and phrases: Human-centered systems, software processes, business processes, deviation, inconsistency, formal model.

1 Introduction

A human-centered system is one in which humans, supported by computerized aids, play a key role [10]. Typical examples of human-centered systems are engineering processes (such as software engineering or VLSI design), information systems, and, more generally, business processes. All these systems share common properties that can be summarized as follows:

- *Humans have a central role in performing the activities needed to accomplish the process goals.* They do so by interacting and cooperating among themselves and with computerized tools. For instance, in a software engineering process, programmers and designers cooperate according to defined rules and procedures to develop and deliver software products. They use several tools such as editors, compilers, and CASE tools to achieve these goals. Tools, however, cannot replace humans, but just support their work.

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

A human-centered system can thus be defined as a combination of humans, organizational structures, rules, procedures, and computerized tools. The goal and purpose of computerized tools is not to replace humans, but to facilitate their work and increase their effectiveness. Even in software development, where the role of automation is essential (how could we ever compile a program manually?), humans are still the key factor to ensure high quality results [21, 29].

In recent years, there has been an increasing interest in methods and technologies to model, improve, and support human-centered systems. Typical examples are WorkFlow Management Systems (WFMSs) [18] and Process-centered Software Engineering Environments (PSEEs) [16]. WFMSs have been developed to support information systems and, more generally, cooperative activities carried out in business processes. PSEEs aim at supporting the engineering process used to conceive, design, develop, and maintain a software product. These *process technologies* share many characteristics [5]. In particular, they are based on languages (Process Modeling Languages or PMLs) supporting explicit process description. The description of the process (the *process model*) specifies how people should interact and work, and also how and when the computerized tools used in the process should be used and/or automatically activated. The process model can then be *enacted* by a *process engine*, which guides and supports the human agents participating in the process, and automates the execution of a number of process steps. The customized support created using process technology for a given human-centered system is called the *process support system*. It acts as the "glue" that ties together the people operating in the process (e.g., software developers) and the domain-specific tools being used (e.g., a compiler or a test data generator).

One of the key issues emerging in building and using process technology is its inability to tolerate unexpected situations and deviations from the specified process model. Most processes, indeed, cannot be completely specified in advance and once for all. This means that the process model and the corresponding process support system must be able to evolve while the process is being executed, and, more generally, they must be able to take unforeseen situations into account.

In the PSEE domain, the solution that several environments adopt is to supply powerful mechanisms to support the modification of a process model [32]. A change in the process is applied to the process model before it is actually performed by human agents. But then a new problem arises: how do we recognize the need for a process model to be modified? Moreover, do we need to change the process model even to deal with temporary changes in the software process? Is it convenient to change the process model even when a minor deviation from the prescribed process occurs?

Similarly, in the WFMS domain [14], unforeseen activities and operations (*exceptions*) that human agents need to accomplish, and that are not specified in the workflow model, are managed by exploiting different kinds of techniques, such as exception handling procedures.

In general, the problem of dealing with deviations and with the resulting inconsistencies between the human-centered systems and the corresponding process support systems is a challenging topic for researchers and practitioners. Process support systems which enforce a strict approach, where human agents cannot perform an operation unless it has been anticipated and described in the process model, may be found to be too prescriptive and rigid. They impose a pre-defined behavior and do not allow for human creativity, unless the process model is explicitly modified. Eventually, they might be rejected by their intended users. On the other hand, tolerating unexpected situations and deviations from the expected behavior demands for powerful mechanisms to identify deviations, describe them, and eventually reconcile them with the baseline process model.

We have addressed these issues by designing and implementing two PSEEs: SENTINEL [12] and SPADE [6]. SENTINEL's main feature is its ability to recognize and describe temporary deviations from the process model. One of most important aspects of SPADE is that it supports dynamic changes of process models. Our work on PSEEs recognized the need for a more systematic definition of the relationship between the process support system and the human-centered system. For instance, it is essential to understand the precise meaning of terms like "deviation" and "inconsistency", since they are often used with different and even conflicting (informal) semantics. Although a few attempts have been reported to deal with these issues [13, 15, 4], there is no satisfactory characterization that can be used to properly define the problem and assess the available solutions.

This paper aims at providing a formal model to describe deviations and inconsistencies, and, more generally, the relationship between a human-centered system and its process support system. We define a model of the environment composed of these two systems. We study the behavior of such composite environment, by analyzing the way its subcomponents influence each other. The ultimate goal of the paper is not to introduce a new approach to process modeling. Rather, it aims at providing a general framework in which:

1. the concepts of deviation, inconsistency, and reconciliation are defined and formalized;
2. different existing systems and approaches can be analyzed according to their ability to deal with these issues.

To better appreciate both the limitations and the potential of the work presented in this paper, it is important to make the following two points:

a. A comprehensive analysis of the motivations and causes of deviations and inconsistencies demands for a detailed study of the cognitive and sociological factors that drive the human behavior when interacting with a computer. In this paper, we do not address the issues related to understanding "why" deviations and inconsistencies occur. Rather, we provide concepts to understand and characterize the state of a human-centered system and its process support system, when deviations and inconsistencies (for some reason) materialize. The goal of the paper is thus to describe "how" deviations and inconsistencies can be defined and observed, independent of the causes and motivations that originate them.

b. The concepts presented in the paper can be applied in a more general context to describe the relationship between a computerized system and (parts of) the real world. For instance, similar problems and issues are encountered when describing a digital control system and the physical device it is controlling. In this paper, we focus on the application of the framework to human-centered systems (and even more specifically to PSEEs), since this is the domain where our experience and research activities have been rooted.

1.1 An informal introduction to the proposed approach

Formal models are used in engineering to reason about complex problems [24, 33]. The main difficulty found in the definition of a formal model is the identification of the entities to be modeled.

This choice is influenced by the characteristics and properties to be studied. In this paper we use a formal model to understand the way human-centered systems and the corresponding process support systems interact. We will use the symbols $\mathcal{H}$ and $\mathcal{P}$ to indicate a human-centered system and its process support system, respectively. $\mathcal{H}$ and $\mathcal{P}$ are modeled by two state machines, H, and P, respectively, possibly with infinite states.

$\mathcal{H}$ and $\mathcal{P}$ constitute an *environment*, denoted by $\mathcal{E}$, whose behavior depends on the way $\mathcal{H}$ and $\mathcal{P}$ interact. The model of $\mathcal{E}$, E, is composed of H and P, and two relations R_s and R_t. R_s relates the states of P and H. In particular, a pair (s_p, s_h) is in R_s if s_p is a state of P, s_h is a state of H, and s_p "correctly describes" s_h. Similarly, R_t relates state transitions in P and H.

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

We distinguish between the concepts of *deviation* and *inconsistency*, even though they are related. Moreover, we distinguish between deviations and inconsistencies occurring within $\mathcal{H}$ and those due to a mismatch between $\mathcal{H}$ and $\mathcal{P}$.

Deviations occurring within the human-centered system $\mathcal{H}$ are called *domain-level deviations* (DDs). A domain-level deviation is an event that deviates from the expected behavior of $\mathcal{H}$. For instance, a DD occurs if a developer involved in a software process writes the source code of a module before writing its design, assuming that a process rule prescribes that source code creation of each module "must" be preceded by the definition of its design. Domain-level deviations might bring the human-centered system in a *domain-level inconsistent* state. For instance, in the example, the deviation brings the human-centered system in an inconsistent state, in which a source code file does not have a corresponding design file. Such deviations and inconsistencies may occur even when no process support system is used. Indeed, a process support system may (and should) describe even domain-level inconsistent states and deviations, in order to guarantee support to exceptional (but even possible) situations that should be tolerated.

Deviations occurring between the human-centered system $\mathcal{H}$ and the process support system $\mathcal{P}$ are called *environment-level deviations* (EDs). An environment-level deviation occurs when the event performed in $\mathcal{H}$ is not mirrored by an event in $\mathcal{P}$ that correctly describes it. This means that the behavior of the human-centered system is not properly reflected in the process support system. As a result of an environment-level deviation, the state of E is *environment-level inconsistent*, that is, the states of H an P do not belong to R_s. For instance, a design document exists, but the process support system is not aware of it (i.e., there is no information in the enacted process model that describes it).

1.2 A sample process

The example proposed in this section aims at clarifying the issue we are addressing. It will be used throughout the paper as a case study.

The example is taken from the software engineering domain. A software system is decomposed into different modules, each assigned to a developer. A developer is in charge of designing, coding, and testing the module. The developer is responsible for keeping the detailed design and the module source code consistent. All the artifacts related to a module (e.g., the design document and the source code) are reviewed by the developers who use it. The revision procedure is composed of the following steps:

- ◇ The developer releases a pre-release of the module she or he is in charge of and notifies all the reviewers (i.e., the developers who use it).
- ◇ Each reviewer produces a review document and stores it in a global workspace.
- ◇ The developer collects all the review documents, analyzes the feedback, and produces a new version of the artifact. If needed, the developer produces a document containing response comments to the reviewers.

The review procedure has a deadline. If the review is not produced within the specified deadline, the developer releases the artifacts without performing additional changes.

For the sake of simplicity, we suppose that the revision procedure is not reiterated; i.e., the new version of the module produced after the revision is supposed to be acceptable.

The software development process is supported by a process support system coordinating the developers' activities. The system controls the review procedure and keeps track of the artifacts and documents produced by the team members.

1.3 Structure of the paper

The rest of this paper is organized as follows:

- ◇ Section 2 introduces the formal models we use to describe $\mathcal{E}$ and its constituents $\mathcal{H}$ and $\mathcal{P}$.
- ◇ Section 3 proposes definitions and concepts to describe and characterize inconsistencies and deviations.
- ◇ Section 4 proposes some concepts to assess the quality of a process support system $\mathcal{P}$, by evaluating its capability of coherently modeling $\mathcal{H}$.
- ◇ Section 5 provides some examples where the framework is applied to existing PSEEs.
- ◇ Finally, Sections 6 and 7 compare this work with other efforts in the literature, and also draw some conclusions and indications for future research initiatives.

To formalize our framework, we adopted the Z formal specification language [30]. Note, however, that Z is only used as a linguistic vehicle we have adopted to express and present the concepts introduced in this paper. It is not intended to be used as a process description language to model deviations and inconsistencies.

2 The model of $\mathcal{E}$

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

2.1 H

The model H of a human-centered system $\mathcal{H}$ is defined as a Z schema that uses the following sets:

$$[HSTATE, HTRANSITION]$$

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

HTRANSITION is the domain of the state transitions of any model H. Function *HTrans* relates each state transition to its source and target state, respectively:

$$HTrans : HTRANSITION \rightarrow (HSTATE \times HSTATE)$$

Based on these definitions, the model H for a specific human-centered system $\mathcal{H}$ is described by the following schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{H} \\
\hline
S_h : \mathbb{P}\, HSTATE \\
T_h : \mathbb{P}\, HTRANSITION \\
s_h^i : HSTATE \\
\hline
\forall\, t : HTRANSITION \mid t \in T_h \bullet HTrans\ t \in (S_h \times S_h) \\
s_h^i \in S_h \\
\hline
\end{array}
$$

where:

- ◇ S_h is the set of H states.
- ◇ T_h is the set of H state transitions.
- ◇ s_h^i is initial state of H.

The schema predicate part states the following propositions:

1. Every transition has a source and target state through function *HTrans*.
2. The initial state of H belongs to S_h.

Example 1 *Let us consider the human-centered system $\mathcal{H}$ described in Section 1.2. A model H of this system is the state machine shown in Figure 1. We assume that this state machine correctly describes all the possible behaviors of $\mathcal{H}$.*

In Figure 1, `h1` *is the initial state of H. It represents the fact that a module* `M` *has not been released yet. The release of the initial version of the module is modeled by transition* `H_Create-Prerelease`*. When the module has been released (state* `h2`*), the review phase may start. Transitions* `H_StoreReviews` *and* `H_MailReviews` *model two possible ways of delivering the review documents to the module designer. The former is more formal: the review is stored in some global workspace and it is then delivered to the designer. The latter, represented by transition* `H_MailReviews`*, indicates that the review may be informally sent to the designer by e-mail. When the review phase terminates (state* `h3`*), the designer may create the final release of the module. This task may be accomplished in several ways. The designer may decide to modify the module design (transition* `H_ModDesign1`*) first, and then the module code (transition* `H_ModCode2`*), or vice versa. Moreover, based on the review results, she or he may decide to deliver the final release without making additional changes to the initial release (transition* `H_FinalRelease1`*). Along with the final release, the designer may deliver a document containing the author's comments on the review document. This document may be created before or after the creation of the final release. Whenever the review is not produced in time, H enters state* `h15` *through transition* `H_DeadlineFailed`*. In this state, the designer can release the module as it is, thus determining the execution of transition* `H_FinalRelease3`*.*

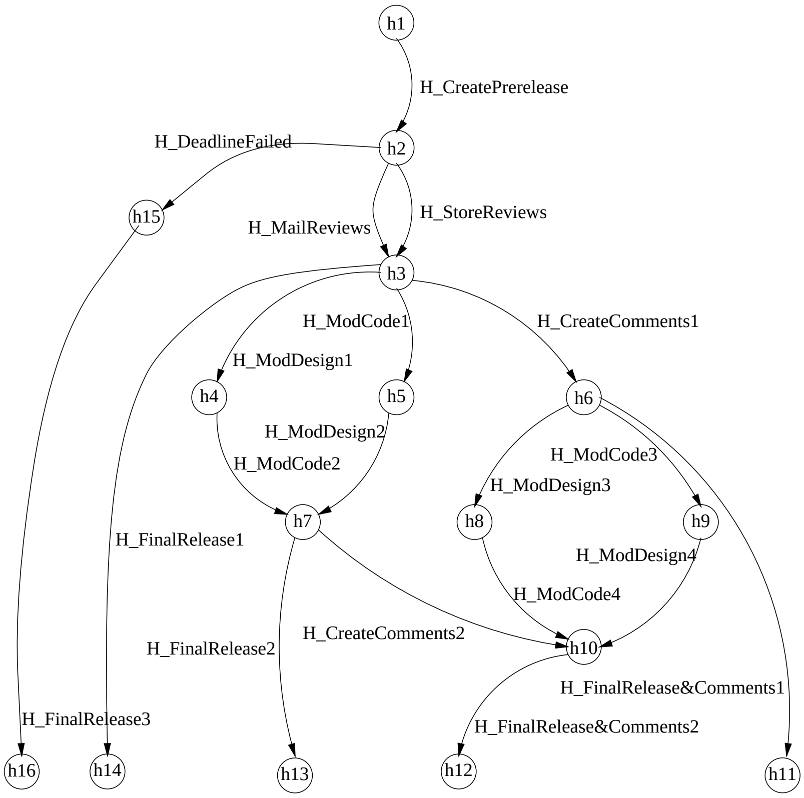

Figure 1: The model H for the example process.

Notice that in this example several (but not all) pairs of states are linked by a state transition. Some behaviors, in fact, never occur. For instance, the designer never writes the author's comments after modifying the module code and before modifying the module design.

Notice also that in H the review procedure may be performed in a way that is not necessarily compliant with the "official" steps described in Section 1.2. For instance, the review documents may be mailed to the developer without storing them in the global repository. The steps of Section 1.2 represent the "expected behavior" of the human-centered system, but there are also other situations that can occur even if they have not been anticipated in the official definition of the process.

In the remainder of the paper we will come back on this example to discuss it in more detail.

□

2.2 P

The Z schema defined for the model P of a process support system $\mathcal{P}$ is analogous to H's schema defined in Section 2.1:

$$[PSTATE, PTRANSITION]$$

$$PTrans : PTRANSITION \rightarrow (PSTATE \times PSTATE)$$

$$\begin{array}{|l}
\multicolumn{1}{l}{P} \\
\hline
S_p : \mathbb{P}\, PSTATE \\
T_p : \mathbb{P}\, PTRANSITION \\
s_p^i : PSTATE \\
\hline
\forall\, t : PTRANSITION \mid t \in T_p \bullet PTrans\ t \in (S_p \times S_p) \\
s_p^i \in S_p \\
\hline
\end{array}$$

Notice that P is not a "process model". Rather, it is a description of the "virtual machine" $\mathcal{P}$ which is supporting the human-centered system $\mathcal{H}$. In a process-centered environment, this virtual machine is obtained by enacting a process model [4]. Drawing an analogy with the traditional world of programming, the process model can be viewed as the program, $\mathcal{P}$ is the virtual machine executing the program, and P is a description of this virtual machine.

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

Example 2 *In the previous section, we defined a model H for the example presented in Section 1.2. Figure 2 shows a model P for the corresponding process support system. In Section 2.3 we will formalize the relationships between H and P, and we will come back to this example to illustrate them. Here we can intuitively say that each state and transition in Figure 2 corresponds to some state and transition in Figure 1. For instance, transition* `P_CreatePrerelease` *describes, in the process support system model, the creation of the initial release of the module in the human-centered system model.* □

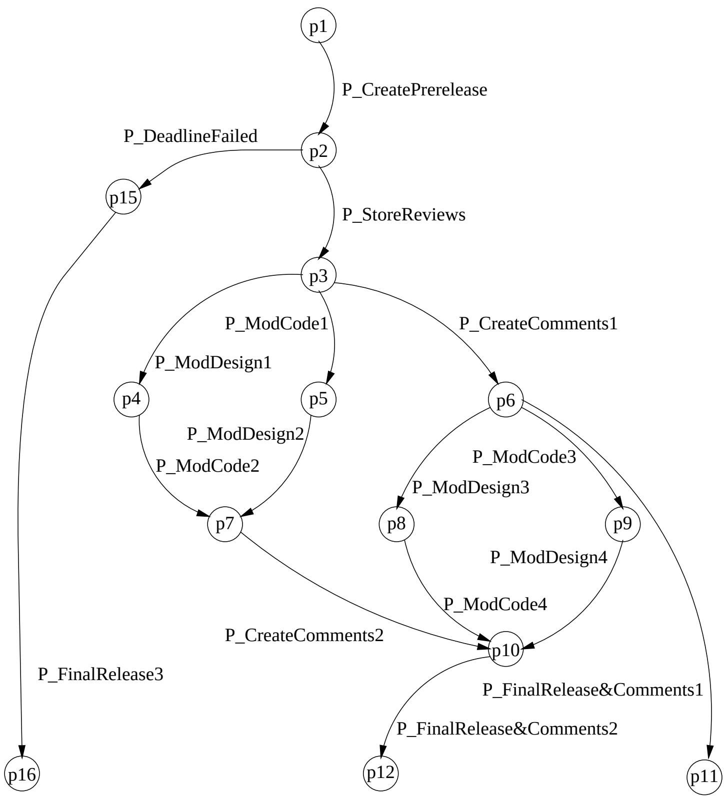

Figure 2: The model P for the example process.

2.3 E

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

An E state is defined as a pair of P and H states:

$$ESTATE == PSTATE \times HSTATE$$

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

$$ETRANSITION == PTRANSITION \times HTRANSITION$$

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

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

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

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

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

The following Z schema describes environment E:

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

$$
\begin{array}{|l}
\hline
\textit{E} \\
H \\
P \\
T_{ef} : \mathbb{P}(\mathit{ETRANSITION}) \\
s_e^i : \mathit{ESTATE} \\
R_s : \mathit{PSTATE} \leftrightarrow \mathit{HSTATE} \\
R_t : \mathit{PTRANSITION} \leftrightarrow \mathit{HTRANSITION} \\
\hline
T_{ef} \subseteq (T_p \times T_h) \\
s_e^i = (s_p^i, s_h^i) \\
s_e^i \in R_s \\
\mathrm{dom}\, R_s \subseteq S_p \\
\mathrm{ran}\, R_s \subseteq S_h \\
R_t \subseteq T_{ef} \\
\forall\, t_p : \mathit{PTRANSITION};\ t_h : \mathit{HTRANSITION} \bullet (t_p, t_h) \in R_t \Leftrightarrow \\
\quad \mathit{first}(\mathit{PTrans}\ t_p) \mapsto \mathit{first}(\mathit{HTrans}\ t_h) \in R_s \wedge \\
\quad \mathit{second}(\mathit{PTrans}\ t_p) \mapsto \mathit{second}(\mathit{HTrans}\ t_h) \in R_s \wedge \\
\quad (t_p, t_h) \in T_{ef} \\
\hline
\end{array}
$$

where:

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

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

For instance, consider a human-centered system that executes the process presented in Section 1.2. Assume that the only way to accomplish the pre-release of a module is to invoke an operation provided by the process support system, which copies the released files from some reserved directory into the reviewers' directory. In this case, the occurrence of transition `H_CreatePrerelease` (see Example 1), which models in H the delivery of the module, always causes the occurrence of transition `P_CreatePrerelease` (see Example 2) that models in P the same operation. Suppose, however, that in $\mathcal{H}$ the designer can make a copy of the module using some other mechanism which is not directly controlled by the process support system (for instance, by directly issuing the Unix `cp` command). This behavior can be represented by a new transition `H_CreatePrereleaseByCP` from `h1` to `h2`. If this operation is not detected and controlled by the process support system, its execution does not cause transition `P_CreatePrerelease` in P to occur. That is, in the first case the only feasible event is (`P_CreatePrerelease`, `H_CreatePrerelease`). In the second case, event (I_{p1}, `H_CreatePrereleaseByCP`) would also be feasible, where I_{p1} is the identity transition that keeps P in the initial state `p1`. In such a case, the module is pre-released, but the process support system has not detected this event.

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

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

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

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

$\square$

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

Example 3 *Referring to Figures 1 and 2, relation R_s is defined by the pairs:*

R_s = $\{$(p1, h1), (p2, h2), ... , (p12, h12), (p15, h15), (p16, h16)$\}$

H states h13 *and* h14 *do not have any corresponding P state through relation R_s. This means that the support system is not able to describe these states. As a result, whenever the human-centered system reaches state* h13 *or* h14*, it will be "inconsistent" with respect to the actual P state. Notice that an inconsistency between H and P may also occur when, given the current states of H and P, s_h and s_p (respectively), s_h is actually described by some other P state $\overline{s_p}$, such that $\overline{s_p} \neq s_p$.*

For the sake of simplicity, we have used similar names to represent every pair of H and P transitions belonging to R_t (e.g., (P_ModCode1, H_ModCode1)*). From Figures 1 and 2 it results that H transitions* H_FinalRelease1*,* H_FinalRelease2*, and* H_MailReviews *do not have corresponding transitions in P.*

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

(I_{p2},H_MailReviews)

(I_{p7},H_FinalRelease2)

(I_{p3},H_FinalRelease1)

(P_StoreReviews, I_{h3})

where I_{p2}, I_{p3}, I_{p7}, and I_{h3} denote identity transitions.

As a consequence of the definition of T_{ef}, $\mathcal{P}$ is unable to detect - say - that users in $\mathcal{H}$ exchange review documents by e-mail. Whenever this happens, the process support system cannot react by performing a state transition that brings it into state p3. $\square$

2.4 Event sequences and histories

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

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

$$
\begin{array}{l}
\textit{E\_SequencesAndHistories} \\
\hline
E \\
S_{te} : \mathbb{P}(\mathrm{seq}_1\ ETRANSITION) \\
PH_e : \mathbb{P}(\mathrm{seq}_1\ ETRANSITION) \\
S_{er} : \mathbb{P}\ ESTATE \\
S_{pr} : \mathbb{P}\ PSTATE \\
S_{hr} : \mathbb{P}\ HSTATE \\
T_{unr} : \mathbb{P}\ ETRANSITION \\
\hline
S_{te} = \{s_{te} : \mathrm{seq}_1\ ETRANSITION \mid (\forall\, i : \mathrm{dom}\, s_{te} \bullet s_{te}\ \ i \in T_{ef}) \wedge (\forall\, i : 2 \mathrel{..} \# s_{te} \bullet \\
\qquad (second \circ ETrans \circ s_{te})\ (i-1) = (first \circ ETrans \circ s_{te})\ i\} \\
PH_e = \{s_{te} : \mathrm{seq}_1\ ETRANSITION \mid s_{te} \in S_{te} \wedge (first \circ ETrans \circ s_{te})\ 1 = s_e^i\} \\
S_{er} = \{s_e : ESTATE \mid s_e = s_e^i \vee \\
\qquad (\exists\, h : \mathrm{seq}_1\ ETRANSITION \mid h \in PH_e \bullet (second \circ ETrans \circ h)\ \# h = s_e)\} \\
S_{pr} = \{s_p : PSTATE \mid (\exists\, s_h : HSTATE \bullet (s_p, s_h) \in S_{er})\} \\
S_{hr} = \{s_h : HSTATE \mid (\exists\, s_p : PSTATE \bullet (s_p, s_h) \in S_{er})\} \\
T_{unr} = \{t : ETRANSITION \mid t \in T_{ef} \wedge \\
\qquad (\forall\, s_{te} : PH_e \bullet \neg\, (\exists\, i : \mathrm{dom}\, s_{te} \bullet s_{te}\ \ i = t))\} \\
\hline
\end{array}
$$

where:

- ◇ S_{te} is the set of all the possible event sequences for E.[2]
- ◇ PH_e is the set of all the possible E histories, that is, all the possible event sequences having s_e^i as initial state.
- ◇ S_{er} is the set of E reachable states.
- ◇ Sets S_{hr} and S_{pr} are defined as the projection of S_{er} on S_h and S_p, respectively.
- ◇ T_{unr} contains all feasible events that do not belong to any history.

3 Inconsistencies and deviations

Using the concepts introduced in Section 2, it is now possible to provide formal definitions for the terms *inconsistency* and *deviation*.

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

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

3.1 Domain-Level Inconsistencies and Deviations

In Section 1.1 inconsistencies and deviations occurring in the human-centered system have been called *domain-level inconsistencies* (DIs) and *domain-level deviations* (DDs), respectively. They can occur even when no process support system is used.

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

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

Whether a state is domain-level inconsistent depends on the human-centered system under consideration. Formally, this concept can be modeled by partitioning the state set S_h in two subsets, S_{h_i} and S_{h_c}, describing the set of domain-level inconsistent and consistent states of H, respectively. Similarly, the concept of domain-level deviation is modeled by partitioning set T_h in two subsets, T_{h_d} and T_{h_g}. T_{h_d} represents the set of domain-level deviations, T_{h_g} represents the set of the expected behaviors (i.e., the "good" transitions).

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

$$
\begin{array}{|l}
\hline
H_1 \\
H \\
S_{h_c}, S_{h_i} : \mathbb{P}\, HSTATE \\
T_{h_g}, T_{h_d} : \mathbb{P}\, HTRANSITION \\
\hline
s_h^i \in S_{h_c} \\
\langle S_{h_c}, S_{h_i} \rangle \text{ partition } S_h \\
\langle T_{h_g}, T_{h_d} \rangle \text{ partition } T_h \\
\forall\, t_h : T_h \bullet t_h \in T_{h_g} \Rightarrow (\mathit{first}\ t_h) \in S_{h_c} \land (\mathit{second}\ t_h) \in S_{h_c} \\
\hline
\end{array}
$$

From the schema it follows that:

- ◇ The initial state of H, s_h^i, is supposed to be domain-level consistent.
- ◇ Deviations and inconsistencies are related concepts. In particular, the last axiom of schema H_1 asserts that a necessary condition for a transition to be "good" (i.e., not to be a deviation) is that it starts and ends in a consistent state. Notice that this is just a necessary condition. In fact, transitions between two consistent states may still represent an undesired action (i.e., a deviation from the desired process).

In the Example 1, states `h4`, `h5`, `h8`, and `h9` belong to S_{h_i}. This is intuitively justified by the fact that they model $\mathcal{H}$ states in which a developer has modified the code of the module she or he is assigned to, but the corresponding design document has not been modified yet (or vice versa). Therefore, design and code are inconsistent, and this is not allowed in the application domain (see Section 1.2). From the axiom reported in schema H_1 it follows that all transitions starting or ending in an inconsistent state (e.g., `H_ModDesign1`, `H_ModCode2`, ...) are domain-level deviations. Moreover, transition `H_MailReviews` has to be considered a deviation too. In fact, according to the statements of Section 1.2, reviews should be delivered by storing them into a global workspace.

As a final remark, we observe that DIs have been simply defined as a subset of S_h. A richer model might go further by classifying inconsistencies according to their severity level. This can be formalized by partitioning S_{h_i} into equivalence classes defined by relation Sev: given two states h_1 and h_2, $h_1\ Sev\ h_2$, if both have the same severity level. It is easy to show that Sev can be defined as an equivalence relation. Again, Sev should be defined according to the requirements and properties of the application domain.

$\mathcal{P}$ does not necessarily describe only domain-level consistent states and "good" transitions, but it can also describe domain-level inconsistent states and deviations. This is true, for instance, for Example 3, where the domain-level inconsistent states `h4`, `h5`, `h8`, and `h9` are properly described in P by states `p4`, `p5`, `p8`, and `p9`, respectively, while domain-level deviations `H_ModCode1`, `H_ModDesign1`, `H_ModCode2`, `H_ModDesign2`, `H_ModCode3`, `H_ModDesign3`, `H_ModCode4`, and `H_ModDesign4` are modeled in P by the transitions having a similar name. Therefore, in general, the process support system $\mathcal{P}$ may be able to control the progress of the human-centered system $\mathcal{H}$ even when it deviates and reaches a domain-level inconsistent state (see also Section 5).

3.2 Environment-Level Inconsistencies and Deviations

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

An *incorrect event* is a feasible event of E, $t_e = (t_p, t_h)$, such that the transition t_p performed by the process support system does not correctly describe the corresponding transition t_h performed by the human-centered system. In other words, an incorrect event is a feasible event that does not belong to R_t.

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

Recalling the previous example and the definition of ED, one may notice that the E feasible events (I_{p2},`H_MailReviews`) and (I_{p7},`H_FinalRelease2`) are typical examples of environment-level deviations.

Domain-level and environment-level deviations are closely related concepts. If the process support system $\mathcal{P}$ does not describe a DD, the occurrence of this deviation results in an ED. For instance, in Example 3, the occurrence of DD `H_MailReviews`, results in ED (I_{p2},`H_MailReviews`). On the other hand, an ED may occur even if a DD does not occur. In this case, the process support system does not completely describe the expected behavior of $\mathcal{H}$. Again, in Example 3, H transition `H_FinalRelease2` is not a DD but its occurrence results in the ED (I_{p7},`H_FinalRelease2`) because transition I_{p7}, performed by the process support system, does not describe transition `H_FinalRelease2`, performed by the human-centered system. Enhancing *completeness*, that is, increasing the number of $\mathcal{H}$ transitions (both DDs and "good" transitions) described by $\mathcal{P}$, reduces the chance of an ED to occur (see Section 4).

When an ED occurs, H and P have to be reconciled in some way. We define a *reconciling sequence* as a sequence of E feasible events starting from an environment-level inconsistent state and terminating in a environment-level consistent state.

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

E_DeviationsAndRecSeqs

$E\_SequencesAndHistories$
$I_e : \mathbb{P}\, ETRANSITION$
$ED : \mathbb{P}\, ETRANSITION$
$RS : \mathbb{P}(\mathrm{seq}_1\, ETRANSITION)$

$$I_e = T_{ef} \setminus R_t$$

$$ED = \{t_e : T_{ef} \mid (first \circ ETrans)t_e \in R_s \land (second \circ ETrans)t_e \notin R_s\}$$

$$RS = \{s : \mathrm{seq}_1\, ETRANSITION \mid s \in S_{t_e} \land (first \circ ETrans \circ s)\ 1 \notin R_s \land (second \circ ETrans \circ s)\ \#s \in R_s\}$$

EIs can be modeled in E as follows:

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

This schema defines two additional sets:

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

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

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

In Example 3, the state (`p2`, `h3`), reachable from state (`p2`, `h2`) through event (I_{p2},`H_MailReviews`), is weakly inconsistent. In fact, event (`P_StoreMail`, I_{h3}) brings E in state (`p3`, `h3`), that belongs to R_s. Conversely, state (`p7`, `h13`), reachable from state (`p7`, `h7`), through event (I_{p7},`H_FinalRelease2`) is a strongly inconsistent state.

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

$$
\begin{array}{|l}
\hline
E_1 \\
\textit{E\_Inconsistencies} \\
\textit{E\_DeviationsAndRecSeqs} \\
\hline
\end{array}
$$

4 $\mathcal{E}$ completeness and coherence

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

In this section we define two properties of an environment $\mathcal{E}$, completeness and coherence, that are indicators of the ability of $\mathcal{P}$ to support a human-centered system $\mathcal{H}$.

4.1 Definition of completeness and coherence

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

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

$E\_StateComplete$

E_1

$\forall s_h : S_{hr} \bullet \exists s_p : S_p \bullet s_p \mapsto s_h \in R_s$

Similarly, E is *transition complete* iff every H transition t_h of some event that belongs to a history of E is described by at least one transition of P. The following schema defines a transition complete environment:

$E\_TransitionComplete$

E_1

$\forall t_h : T_h \mid (\exists h_e : PH_e \bullet \exists i : \mathrm{dom}\, h_e \bullet t_h = (second \circ h_e)i) \bullet$
$\quad (\exists t_p : T_p \bullet t_p \mapsto t_h \in R_t)$

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

If an environment E is built in such a way that all events of any possible history belong to R_t, EDs never occur. We define this environment to be *coherent*. In a coherent environment any transition occurring in H is always mirrored by the transition of P that correctly describes it.

A coherent environment is formally described by the following schema:

$E\_Coherent$

E_1

$\forall h_e : PH_e \bullet \forall i : \mathrm{dom}\, h_e \bullet h_e\ i \in R_t$

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

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

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

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

$$\neg (\exists s_p : S_p \bullet s_p \mapsto s_h \in R_s) \tag{1}$$

There are two possible cases:

i. $s_h = s_h^i$ or

ii. $s_h \neq s_h^i \wedge \exists s_h' : HSTATE, t_h : T_h \bullet s_h' \in S_{hr} \wedge s_h' = (first \circ HTrans)t_h \wedge s_h = (second \circ HTrans)t_h$.

If case (i) holds, from the definition of R_s, it follows that at least the initial state s_p^i of P, exists such that $s_p^i \mapsto s_h \in R_s$. This contradicts Formula 1.

If case (ii) holds, from the transition completeness hypothesis, we deduce that there are two states $s_{p_1}, s_{p_2} : S_p$, and a P transition $t_p : T_p$ such that:

$$s_{p_1} = (first \circ PTrans)t_p \wedge s_{p_2} = (second \circ PTrans)t_p \wedge t_p \mapsto t_h \in R_t$$

Therefore, from axiom on R_t, we can deduce that $s_{p_2} \mapsto s_h \in R_s$. This contradicts Formula 1. □

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

Proof. Suppose that E is coherent. Then for each transition $t_h : T_h$, such that:

$$t_h = (second \circ h_e)\ i$$

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

$$t_p = (first \circ h_e)\ i$$

exists that correctly describes it. This proves that E is transition complete. $\square$

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

4.2 Metrics for completeness and coherence

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

$$
\begin{array}{|l}
\hline
E_2 \\
\hline
E_1 \\
S_{undes} : \mathbb{P}\ HSTATE \\
T_{undes} : \mathbb{P}\ HTRANSITION \\
PH_{incorr} : \mathbb{P}(\operatorname{seq}_1 ETRANSITION) \\
\hline
S_{undes} = \{s_h : S_{hr} \mid (\neg\ (\exists\ s_p : S_p \bullet s_p \mapsto s_h \in R_s))\} \\
T_{undes} = \{t_h : T_h \mid \\
\quad ((\exists\ h_e : PH_e \bullet \exists\ i : \operatorname{dom} h_e \bullet t_h = (second \circ h_e) i) \\
\qquad \wedge \\
\quad \neg\ (\exists\ t_p : T_p \bullet t_p \mapsto t_h \in R_t))\} \\
PH_{incorr} = \{h_e : PH_e \mid (\exists\ i : \operatorname{dom} h_e \bullet h_e\ i \notin R_t)\} \\
\hline
\end{array}
$$

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

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

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

$$
\begin{array}{|l}
\_ \geq_{sc} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall\ \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{sc} \tilde{E} \Leftrightarrow \\
\quad (\hat{E}.S_h = \tilde{E}.S_h \wedge \hat{E}.T_h = \tilde{E}.T_h \wedge \hat{E}.s_h^i = \tilde{E}.s_h^i \wedge \\
\quad (\exists\ f_{sc} : \hat{E}.S_{undes} \rightarrowtail \tilde{E}.S_{undes}))
\end{array}
$$

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

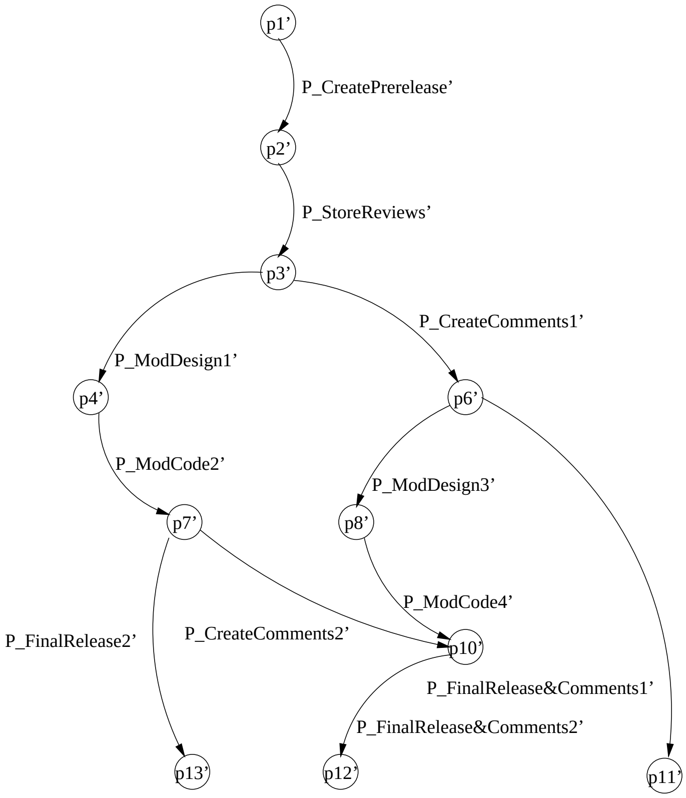

Figure 3: The process support system model $\tilde{P}$.

$$
\begin{array}{|l}
\_ \geq_{tc} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{tc} \tilde{E} \Leftrightarrow \\
\quad (\hat{E}.S_h = \tilde{E}.S_h \wedge \hat{E}.T_h = \tilde{E}.T_h \wedge \hat{E}.s_h^i = \tilde{E}.s_h^i \wedge \\
\quad (\exists f_{tc} : \hat{E}.T_{undes} \rightarrowtail \tilde{E}.T_{undes}))
\end{array}
$$

$$
\begin{array}{|l}
\_ \geq_{c} \_ : E_2 \leftrightarrow E_2 \\
\hline
\forall \hat{E}, \tilde{E} : E_2 \bullet \hat{E} \geq_{c} \tilde{E} \Leftrightarrow \\
\quad (\hat{E}.S_h = \tilde{E}.S_h \wedge \hat{E}.T_h = \tilde{E}.T_h \wedge \hat{E}.s_h^i = \tilde{E}.s_h^i \wedge \\
\quad (\exists f_{c} : \hat{E}.PH_{incorr} \rightarrowtail \tilde{E}.PH_{incorr}))
\end{array}
$$

Example 4 *Let $\hat{P}$ be the process support system described in Example 2 (see Figure 2) and supporting the human-centered system H of Example 1 (see Figure 1). We can compare the resulting environment $\hat{E}$ with the environment $\tilde{E}$, composed of H itself and of the model $\tilde{P}$ described by the state machine shown in Figure 3. Using the previous definitions, it can be shown that $\hat{E} \geq_{sc} \tilde{E}$ and $\hat{E} \geq_{tc} \tilde{E}$. In fact, it is easy to find a function f_1 from $\hat{E}.S_{undes}$ = {*`h13`*,* `h14`*} to $\tilde{E}.S_{undes}$ = {*`h5`*,* `h9`*,* `h14`*,* `h15`*,* `h16`*} and a function f_2 from $\hat{E}.T_{undes}$ = {*`H_MailReviews`*,* `H_FinalRelease1`*,* `H_FinalRelease2`*} to $\tilde{E}.T\_undes$ = {*`H_MailReviews`*,* `H_FinalRelease1`*,* `H_ModCode1`*,* `H_ModDesign1`*,* `H_ModCode2`*,* `H_ModDesign2`*,* `H_DeadlineFailed`*,* `H_FinalRelease3`*} that are injective but not surjective, as required by the formal definition of operators $\geq_{tc}$ and $\geq_{tc}$, respectively.* □

If the level of completeness and coherence of an environment is low, then EDs are likely to occur. In this case, the level of support provided to a human-centered system may be enhanced by replacing

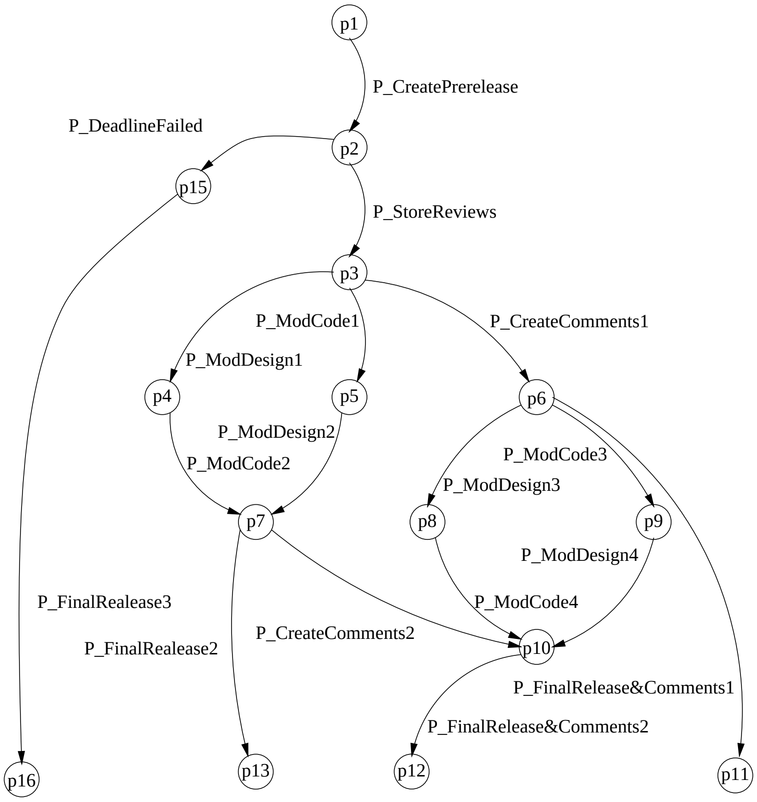

Figure 4: The process support system model $\overline{P}$.

the corresponding process support system with another system that provides a higher level of state completeness, transition completeness, and coherence. If the new P only adds states and transitions to the older one and does not eliminate any of the existing states and transitions, we say that the resulting environment (let it be $\hat{E}$) *extends* the older one (let it be $\tilde{E}$), and we write $\hat{E} \sqsupseteq \tilde{E}$. Formally:

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

Example 5 *Consider again the environment $\hat{E}$ composed of the process support environment $\hat{P}$ shown in Figure 2 and of the human-centered system H shown in Figure 1. This environment may be extended by replacing $\hat{P}$ with the new process support system, $\overline{P}$, shown in Figure 4. In fact, all the H states and transitions described in $\hat{P}$ have a correspondent state or transition in $\overline{P}$. Moreover, state* `h13` *and transition* `H_FinalRelease2` *are described in $\overline{P}$ by* `p13"` *and* `P_FinalRelease2"`*, respectively.* □

4.3 A classification of possible extensions

In the previous section we observed that if an environment $\mathcal{E}$ does not offer the level of completeness and coherence that is required by the application domain, it should be modified. In order to modify $\mathcal{E}$, we can change $\mathcal{P}$ to improve its ability to describe $\mathcal{H}$, or we can change $\mathcal{H}$ to reduce the set of reachable states (and/or transitions) that $\mathcal{P}$ does not represent. Finally, we can improve environment coherence by reducing the difference between T_{ef} and R_t in order to decrease the number of possible incorrect histories.

More specifically:

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

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

The next section will provides some examples of these different strategies.

5 Examples

The previous sections defined the concepts of deviations and inconsistencies that can be used to precisely characterize the problem of tolerating and controlling deviations and inconsistencies in human-centered systems. In this section, these concepts are applied to describe and compare the characteristics of three PSEEs, namely, SPADE [6], SENTINEL [12], and Provence [9].

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

SENTINEL, like PEACE [2], assumes that humans can deviate from the specified process model, and relies on its flexibility to represent deviations in its internal state.

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

Sections from 5.1 to 5.3 provide a description of these three PSEEs, exploiting the framework presented in this paper. Section 5.4 briefly draws some lessons derived from the comparison of the above PSEEs, highlighting the relationship between the completeness and coherence properties, and the architecture and features of these PSEEs.

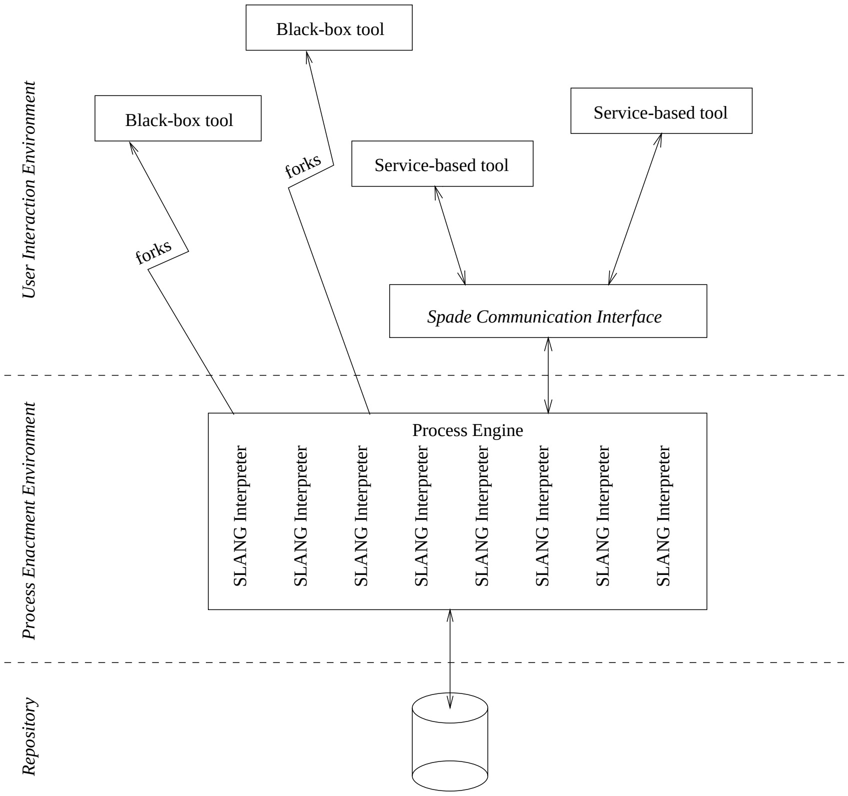

Figure 5: The SPADE Architecture.

5.1 SPADE

SPADE [7] is a software engineering environment that supports software process analysis, design, enactment and evolution. SPADE is based on a process modeling language, called SLANG, which, in turn, is based on high-level Petri nets. In SLANG process artifacts, including process models, are modeled as tokens of a Petri net and are implemented as objects in an object-oriented database. Transitions of the Petri net are associated with a *guard* and an *action*. The guard is a precondition that has to be satisfied by a tuple in order to be selected to enable the firing of the transition. The action describes how the firing of the transition generates new token values in the output places as a function of the values of the token removed from the input places. Suitable constructs are also provided to describe the interaction with software engineering tools and process agents in a uniform style.

5.1.1 The SPADE environment

The SPADE architecture is structured in three different layers (Figure 5): the Repository, the Process Enactment Environment (PEE), which includes the Process Engine, and the User Interaction Environment (UIE), which includes the SPADE Communication Interface (SCI).

The Repository stores both process model fragments and process artifacts. A process artifact can be any document or information created during the software development process. The Repository is implemented on top of the object-oriented database management system O_2 [27].

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

The UIE is the SPADE front-end to its users. It is composed of the tools used by the process agents involved in the development activities and controlled by the PEE. Black-box tools are seen by the PEE as a single function that receives some input and produces some output. SPADE controls only the invocation and the termination of these tools. Often, these tools use their own repository for data (typically the file system). This means that all the process data stored in the O_2 database have to be explicitly converted and mirrored in the file system to make it possible for tools to operate on them. Service-based tools offer an interface through which it is possible to invoke all the individual services they provide. Moreover, each tool can invoke external services offered by other tools and notify the PEE of specific events or tool state changes. SCI manages the communication between service-based tools and the PEE, through the SPADE Communication Protocol. SCI is connected to all service-based tools and to the Process Engine executing the different SLANG Interpreters; it is able to redirect messages from the UIE to the enacting active copies.

SPADE offers support to process evolution through the reflectivity features of SLANG, that is, process models may be modified as any data by other processes (metaprocesses). A detailed presentation of this issue may be found in [6].

5.1.2 Completeness and coherence in SPADE

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

1. All operations performed in the software process must be *explicitly* described in the process model. Whenever we want to enable the PSEE to accomplish some new operation, we need to introduce a specific process model fragment that implements it.

   The process model can deal with DIs and DDs. This can be accomplished by explicitly introducing new process model fragments describing such situations. Referring to the process example introduced in Section 1.2, suppose that the source code and the design specification of a module are represented in the process model as SLANG tokens. The human-centered system state in which the source code is not aligned with the design specification may be described by adding a flag to the token describing the source code, which signals the inconsistency. Moreover, the presence of this flag may cause the firing of some transitions devoted to fix the inconsistency.

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

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

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

Clearly, when these principles are violated, then EDs occur. As an example of ED in SPADE, consider the SLANG process fragment shown in Figure 6, which describes activity `AddRFC` that adds a request for changing a module (RFC) to a document repository. Suppose that only the project manager is allowed to add new change requests. In the SLANG fragment of Figure 6, a guard is associated with transition `StartActivity` to guarantee that both the requests to create a new RFC and to add it to the repository are performed by the project manager. Suppose that a different agent wants to add a new RFC, and that she or he issues this command through a tool that has been integrated in SPADE. The command is reified in place `NewRFCRequest` as a token,

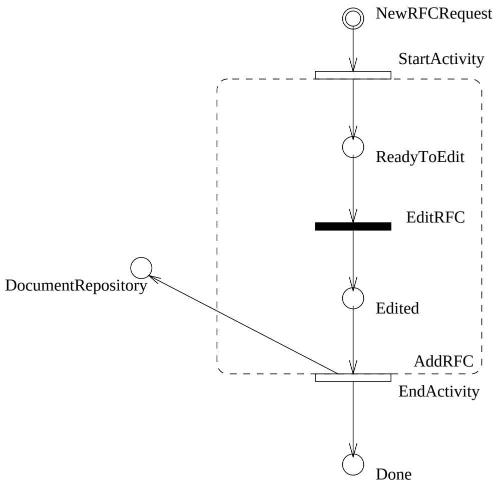

Figure 6: SLANG activity AddRFC.

but it does not cause the firing of transition `StartActivity`, since its guard evaluates to false. That is, by using SPADEShell or any other tool that has been integrated in SPADE, this agent will never be allowed to accomplish the operation.

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

Even without introducing this new process fragment, a file describing the new RFC can be created outside the control of SPADE if the agent is allowed to access a standard Unix shell. In such a case, the state of SPADE would not change, while the state of the human-centered system (i.e., the software process and, specifically, its file system) would be different because there is a new RFC. Therefore, an EI can still be introduced because of the inability of the system to guarantee coherence (e.g., there is a way to bypass SPADE). To eliminate this EI, we should preclude the possibility for the agent to execute any operation which creates and edits files outside the control of SPADE. This choice would reduce the set of feasible events that do not belong to R_t (see Section 4.3).

Using the framework introduced in this paper, we can describe this situation more precisely. Figure 7 shows the E model describing activity `AddRFC`. Transition `PMAddRFC` describes activity `AddRFC` performed by the process manager, while transition `UAddRFC` describes the same activity performed by a different user.

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

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

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

Coherence SPADE provides powerful mechanisms for controlling the tools integrated in SPADE and used by software developers. Operations supported by tools integrated in SPADE are detected by the Process Engine, and used to update the SPADE internal state. Coherence is therefore supported by *exploiting the control integration facilities* offered by the PSEE.

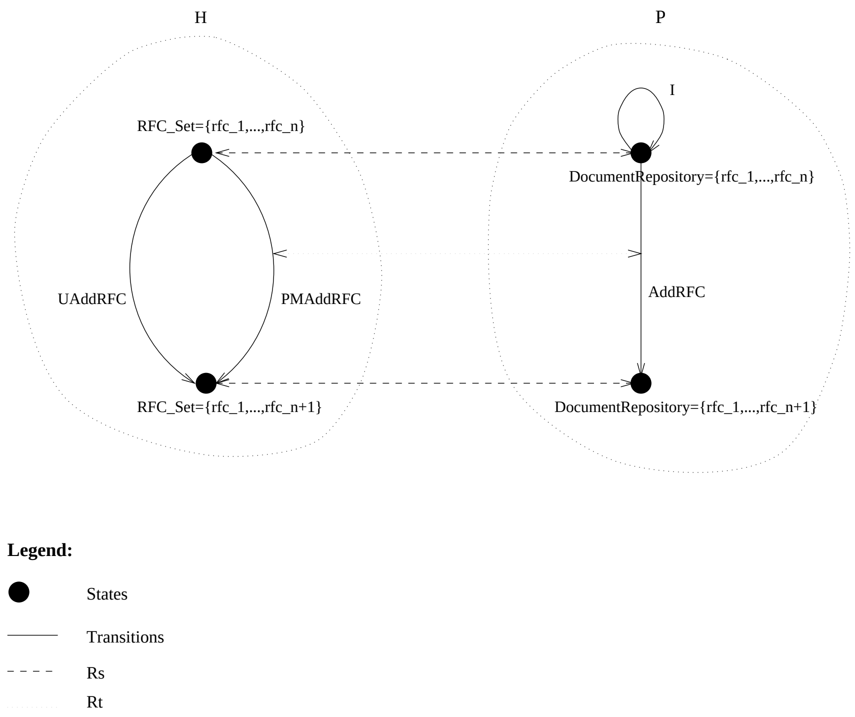

Figure 7: The model of the environment composed of SPADE enacting activity `AddRFC` and the corresponding human-centered system.

> Clearly, if the user can access tools that are only partially integrated within SPADE, EIs can occur. In general, the difference between T_{ef} and R_t is reduced by prohibiting those actions that are not controlled by SPADE.

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

5.2 SENTINEL

The goal of the SENTINEL (Software-engineering Environment to Tolerate INconsistencies Employing Logic) project [12] is to provide mechanisms to avoid EDs by automatically enhancing state and transition completeness. SENTINEL does not force users to rigidly follow the process model. Rather, it allows them to somehow deviate from the stated process model.

5.2.1 The SENTINEL environment

The SENTINEL environment is based on a process modeling language called LATIN (LAnguage to Tolerate INconsistencies). Software processes are modeled in LATIN as collections of task types.[3] Each task type describes a software 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. In the body, two different classes of operations can be performed: *actions*, and *value assignments* (EXIT clause). Actions produce results in the human-centered system (e.g., invoking a tool). Value assignments modify the values of the state variables of the task.

[3]LATIN has been defined extending ASTRAL [19], a formalism for specifying real time systems.

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

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

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

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

5.2.2 Completeness and coherence in SENTINEL

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

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

The following example shows SENTINEL's ability to automatically tolerate DDs even if they are not explicitly described in the process model, by increasing state and transition completeness. Consider the activity `AddRFC` described in the example of Section 5.1.2. This activity can be modeled in LATIN by the transition shown in Figure 8. Figure 9 shows the P model implemented by SENTINEL enacting this specification fragment. By comparing this figure with Figure 7 we observe that transition `UAddRFC` (which does not have a corresponding transition in SPADE) is described in SENTINEL by transition `AddRFC(U, rfc_n+1)`, and that state `RFC_Set = {rfc_1, ..., rfc_n+1}` has two states of SENTINEL describing it.

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

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

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

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

```
EXPORTED_TRANSITION AddRFC( u: User; RFCName: STRING )
ENTRY
        NOT EXISTS RFC IN DocumentRepository (RFC.Name=RFCName) AND u.Role=PM
LOCAL
        rfc: RequestForChange;
ACTION
        Call(``vi '' + RFCName + ``.rfc'');
EXIT
        rfc.Name=RFCName; rfc.File=RFCName + ``.rfc'';
        Add(DocumentRepository,rfc);
```

Figure 8: LATIN transition AddRFC.

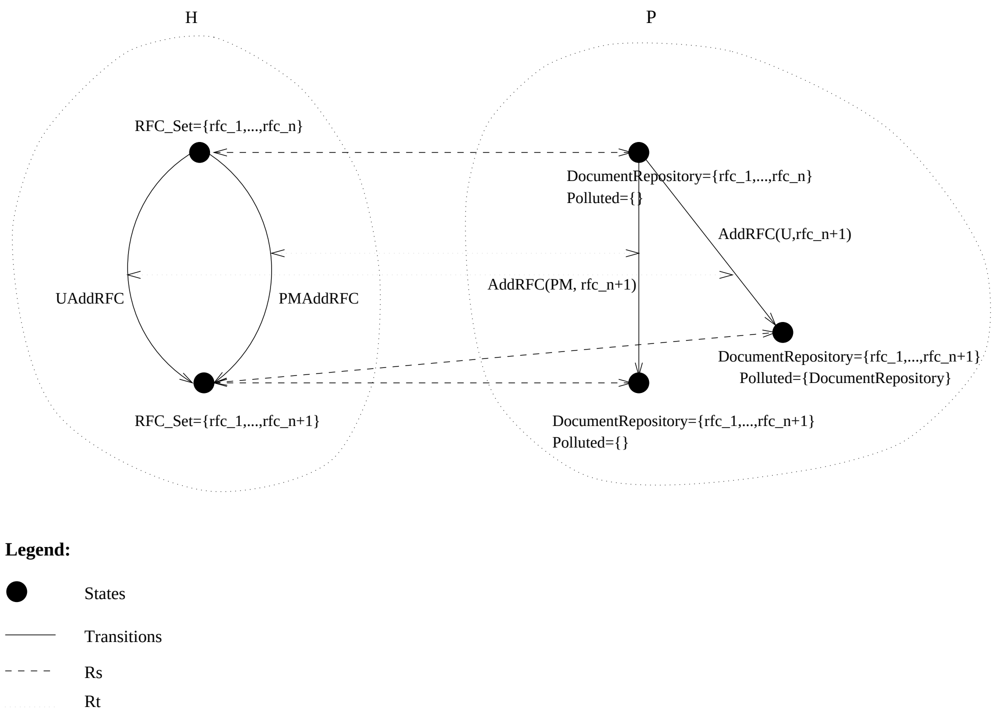

Figure 9: The model of the environment composed of SENTINEL enacting transition AddRFC and the corresponding human-centered system.

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

5.3 Provence

Provence [9] is a PSEE that explicitly separates the enactment of the software process model from the actual execution of the software process. Provence is based on an event monitor that detects the occurrence of events in the human-centered system and notifies the process support system. The PSEE uses the information provided by the event monitor to define the actual state of the process support system and to determine whether software process execution is consistent with the process model.

5.3.1 The Provence environment

Provence is based on an event-action monitor called Yeast [25, 28], which provides mechanisms to detect significant events occurring in the human-centered system and to trigger specific actions that have been associated with those events. Provence monitors the execution of the software process by translating high-level process tasks into low-level event-action specifications, involving file-based events and tool invocation events that are relevant to the process. The occurrence of a sequence of low-level events indicates the occurrence of a particular task in the process. Yeast is responsible for notifying the process model interpreter whenever an event occurs. The process model supplied to Provence describes the tasks that must be performed and the dependencies among these tasks. Process modeling is accomplished using MARVEL [8].

Thus, in Provence there is a component (Yeast) which is in charge of observing all the events occurring in the human-centered system, and mapping them to specific process steps. This monitor is implemented by trapping the invocations of operating system services. It is thus possible to observe even low-level events occurring in the human-centered system (e.g., file system operations). As an example, if a file named `main.c` is opened, written, and finally closed, using any editing tool, the monitor can reasonably deduce that the modification of a source code has occurred.

5.3.2 Completeness and coherence in Provence

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

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

Therefore, the Provence approach can be summarized as follows:

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

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

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

Figure 10: A preliminary comparison of SPADE, SENTINEL, and Provence.

5.4 Evaluation

The informal application of the proposed framework to the above PSEEs makes it possible to draw some preliminary conclusions (summarized in Figure 10).

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

2. Coherence can be improved by enriching the architecture of the process support system with mechanisms that trigger all the events occurring in the human-centered system and map them onto the process model being enacted. SPADE operates at the tool control-integration level, by monitoring the messages being exchanged. Provence operates at the operating system level, by monitoring the OS services being invoked.

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

6 Related Work

As anticipated in Section 1, the roots of this work are in the research activity carried out by the authors in the SPADE and SENTINEL projects. In particular, we started working on the issue of tolerating and controlling inconsistencies in the SENTINEL project [12]. In parallel, the work carried out within the SPADE project to support dynamic process model changes originated a first informal discussion of some of the issues presented in this paper [4].

The topic addressed in this paper has also been discussed by Dowson and Fernström [13], who introduced an informal model that has often been referenced. This model is based on the identification of three conceptual layers: the process definition domain (i.e., the process definitions), the process enactment domain (i.e., the enactment of process definitions), and the process performance domain (i.e., the actual process performance). Roughly speaking, the process definition domain corresponds to what we simply call the process model. The process enactment domain is what we call the process support system. Finally, the process performance domain is the human-centered system. With respect to Dowson and Fernström's work, our approach provides a more formal characterization of the problem. In particular, our description of (in)consistency, completeness and coherence formalizes the relationships between process performance (i.e., human-centered system) and process enactment (i.e., process support system).

In [15] Fernström extends and partially formalizes the framework introduced in [13]. He discusses the importance of relating process enactment and process performance models and the need for a formal description of this relationship. He also focuses on formally describing state synchronization between process enactment and process performance during system progress.

The problem of detecting and representing environment-level inconsistencies and deviations that may arise during the software development process, has been addressed by Cook and Wolf in [11]. In this paper, the authors develop a *process validation technique* for detecting and characterizing any differences between a formal model of the software process being executed and the actual execution of the process itself. The enacted process model and the process that is actually executed are seen as streams of events, and metrics to measure the distance between these two streams are provided. These metrics are all based on string distance concepts and range from simple exact matching to a non-linear distance measure in terms of event insertions and deletions. The authors apply their metrics to the ISPW6/7 example, showing how it is possible to measure the correspondence between a model of this process expressed as a Petri net and different examples of process execution.

As for the general issue of tolerating and supporting inconsistencies, pioneering work was described by Balzer [3]. This seminal work argues for the need to tolerate (domain-level) inconsistencies and proposes pollution markers as a mechanism for describing and controlling them. The author presents a technique to automatically relax integrity constraints formulated as logical sentences in order to allow their violation by exceptional data and to automatically mark these data with *pollution markers*. A pollution marker is an atomic sentence indicating the violation of a specific constraint by a particular datum.

PEACE [2] proposes a specific language to describe and reason about facts that are believed to be true during software process. The authors of PEACE, focus their attention on the problem of inconsistency between process enactment and process performance states (they use the Dowson and Fernström definitions) and try to solve the problem by using non-monotonic logical reasoning. By following their approach, process support can be provided even in presence of uncertain and incomplete knowledge of the actual performing process [1]. In fact, during the enactment, believes may change over time and the runtime system is able to non-monotonically retract the derived propositions. This augments the environment flexibility, reducing the need for deviations. Actually, PEACE, as well as SENTINEL, allows one to avoid the occurrence of EDs and EIs, by enriching the level of completeness through the modeling of incomplete and uncertain information.

K. Narayanaswamy and N. Goldman in [26] address the problem of consistency checking and maintenance in the repositories of PSEEs, containing software process artifacts. Multiple access to these documents may result in inconsistencies between different versions of the same document produced by different developers. Database techniques to preserve consistency, like locking methods, are too restrictive for cooperative software development environments which are concerned with long transactions on large documents. The authors propose an alternative approach based on negotiation of *proposed change* (PC) to documents that supports "lazy" consistency during development. Any developer makes the necessary modifications on a local copy of the documents and, after finishing modifications, submits a change proposal to the process support system. This system automatically manages PCs and controls the process of collecting and merging PCs in order to obtain a new version of the document. This approach explicitly distinguishes between the time when the system is expected to be in a consistent state, and the time when it can be in an inconsistent state, thereby increasing system flexibility.

PEACE, pollution markers, and the proposed change technique by Narayanaswamy and Goldman are all significant examples of mechanisms that can be used to enrich a modeling language, in order to ease the representation and management of DIs, to increase the flexibility of the process support system, and to prevent both EDs and EIs.

Another approach to describe domain-level inconsistencies originates in the viewpoint work carried out at Imperial College [17]. The viewpoint technique aims at representing and integrating different views of a human-centered systems. Each developer in the viewpoint framework has his/her own *viewpoint* that is a partial specification of the system being developed. Each viewpoint is managed locally and is loosely coupled with the viewpoints owned by the other developers. It encapsulates knowledge about both the used representation language, and the performed development process. This additional information supports the formalization of intra-viewpoint and inter-viewpoint consistency rules. These rules formalize consistency constraints inside single viewpoints or among viewpoints describing different subsystems (possibly specified using different languages) and give the process manager the chance to specify automatic procedures to handle inconsistencies.

The problem of describing inconsistencies that could arise in complex software systems, like a programming environment, has been addressed by T. M. Hagensen and B. B. Kristensen in [20]. In this work, the authors suggest the need to view software systems such as programming environments, from a consistency perspective. They introduce a general framework for discussing and describing consistency, which is based on the concepts of *description* and description *interpretation*. Consistency of descriptions is defined as a relation between their interpretations. Using these definitions they formalize the concept of internally consistent description and various types of consistency relations between different descriptions. The authors use their framework to model a programming environment using a graphical notation to describe different kind of descriptions involved in such an environment, like text files, structured files and executable files, and to model the operations allowed on these descriptions. Using their model, the authors present also a preliminary proposal for a taxonomy of the techniques for consistency handling in programming environments.

The problem of dealing with domain-level inconsistencies, has also been addressed in the WFMS domain. In [14], Ellis, Keddara and Rozenberg address the problem of *dynamic change* in workflow systems as the first step to enhance flexibility and usability of WFMS environments. Workflow systems are designed to assist groups of people in carrying out work procedure. They contain knowledge of how procedures are performed in the default case (they enact a *workflow model*). Since organizational changes and exceptions to the default case are inevitable, it is important that workflow systems support changes to the workflow model even during enactment (*dynamic changes*). [14] formalizes the concept of dynamic change correctness in the context of the Information Control Net (ICN) model of workflow. It also presents an algorithm that, given a dynamic change to an ICN model, is able to find a correct dynamic change equivalent to the first one.

Strong and Miller in [31] examine the problem of inconsistencies and exceptions in computer-based information processes and WFMS. According to the authors, exceptions are cases that cannot be correctly processed by computer system without manual intervention. The authors try to derive a set of managerial recommendations for treating exceptions, by examining an information process used by a really existing electronic system factory. They conclude that exceptions are intrinsic to organizational processes and cannot be considered as errors to be eliminated. According to this perspective, it is necessary to provide support and guidance to humans in order to enable them to solve the occurred exceptions. It is, however, impossible to replace humans in solving these process exceptions.

The work presented in this paper is also related to a more general theme: the relationship between a computerized system and the application domain where it is supposed to be used. This topic has been recently addressed by Jackson in [22] and [23]. Jackson discussed the relationship between these two entities, and provide important insights to better clarify the relationship between "the world and the machine". Our work provides formal foundations to discuss this general problem. We have applied the framework in a more restricted domain (human-centered systems and, more specifically, process-centered software engineering environments) only because this is the context where this work originated.

7 Conclusions

The issue of supporting human-centered systems is relevant and has a strong impact on industrial practice. The goal is to define the proper mechanisms and environment architectures that support an efficient and controlled execution of human-centered processes. Within this context, it is extremely important to understand how to manage the relationship between the process, and the computerized environment that models and supports it. In the software engineering domain, the issue is how to build PSEEs that are able to effectively support software engineers. In the WFMS domain, the general issue is to identify useful means to assist and guide groups of agents that cooperate to achieve a common goal in their daily business activity.

In this paper, we propose a framework to formalize the relationship between a human-centered system, i.e., an existing process, and its process support system, i.e., the technology that supports the process. We argue that understanding this relationship is an essential step towards designing effective process support systems.

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

We have applied this framework to assess the features of three existing PSEEs. This assessment allows us to argue that completeness is mainly related to the features and characteristics of the language used to model processes, while coherence is more strictly related to the architecture of the process support system.

We believe that the results of this work can be used both to assess and compare existing systems, and to clarify and guide further research work. Moreover, it can be used in a more general context to study the relationship between a computerized system and the real world it is interacting with.

References

[1] 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.

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

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

[4] 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.

[5] S. Bandinelli, E. Di Nitto, and A. Fuggetta. Supporting cooperation in software development. Technical Report 32-95, Politecnico di Milano, 1995. Submitted for publication.

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

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

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

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

[10] R. Conradi, C. Fernström, and A. Fuggetta. Concepts for evolving software processes. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

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

[12] 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.

[13] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] 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.

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

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

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

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

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