How To Deal With Deviations During Process Model Enactment

G. Cugola E. Di Nitto C. Ghezzi M. Mantione
Politecnico di Milano
Dip.to di Elettronica e Informazione
P.zza Leonardo da Vinci, 32
20133 Milano
e-mail: {dinitto, ghezzi}@elet.polimi.it

Abstract

A fundamental problem in software processes is how the intrinsic rigidity of a predefined (formal) model can be reconciled with the need for flexibility, change, and evolution. We therefore distinguish between software processes, as specified in a process description, and their actual performance by humans. Further, we claim that the two inevitably diverge, and thus it is necessary to provide means to reconcile them. We present a preliminary exploration into the problem. In particular, we illustrate how a temporal logic-based approach can be used to capture and tolerate some deviations from the process description during execution. We present a simple process language (LATIN), and its prototype environment (SENTINEL), in which these ideas are currently experimented.

1 Introduction

The need for quality improvement in software production has led to a research focus on *software processes*. In particular, considerable research activity has been (and still is) devoted to process modeling languages and automated support to software development, based on formal process models. The idea is that a Process-centered Software Engineering Environment (PSEE) can be built around a Process Modeling Language (PML). The PSEE should provide tools to validate PML process models and, most important, enact them. Enactment (i.e., interpretation of the process description by a machine) should ensure that the actual development process follows the rules and constraints that are specified by the model. That is, the model is not simply a passive documentation of what should be done, but rather becomes an active description of how things are carried out in reality.

A fundamental problem to face is how the intrinsic rigidity of a predefined process model can be reconciled with the need for flexibility, change, and evolution that is needed in practice during software development. A process model is needed, because it reduces the danger of ambiguity, it may improve communication among people, and may support automation. On the other hand, forcing people to follow a prespecified process may be inconvenient or even impossible.

This problem has been recognized by several researchers. A possible solution consists of providing ways to support dynamic process evolution. This means that, while the process is being enacted, it may be possible to change the process model or even its current instance. This requirement led to incorporating facilities to support meta-processes in software processes, such as introducing reflective features into process languages [3, 8]. This solution is adequate if one - say - wishes to support a kind of gradual evolution of the process as more feedback is provided by data gathered during enactment. The underlying philosophy, however, still considers the PSEE as a prescriptive agent: there is no way to enforce process behaviors that differ from what is currently embodied by the model, other than changing the model itself!

An alternative approach might allow the observed process to diverge from the modeled process. In such a case, however, it is necessary to provide support to handling the inconsistencies that arise between the process model and the observed process, otherwise the model becomes useless. This is exactly the focus of this paper. We would like to be able to formally specify a model that embodies what we consider to be the desirable process. If the observed process happens to follow the model, all remains consistent. The PSEE, however, should recognize and tolerate some deviations that may generate inconsistencies.

The abstract approach we explore in this paper is based on modeling the software process as a collection of state machines, whose states are described by a set of state variables. Legal state transitions are described via preconditions, and definition of safe states is specified via invariant assertions. Human interaction with the process model triggers certain state transitions. As long as the preconditions are satisfied for each state transition, the process is guaranteed to remain in a safe state. If, however, the precondition is violated due to some user interaction, the process is allowed to proceed as long as the invariant assertions that define safe states hold. This situation is called *tolerable deviation*. No further progress can occur, however, if an unsafe state is entered. In this case a reconciling process has to be carried out to fix divergence between process model and observed process. When a tolerable or intolerable deviation occurs, the data produced by the illegally fired transition may be incorrect. We say that data may be "polluted". The approach described here is able to recognize the data that might be polluted, and provides suitable reasoning facilities to support *pollution analysis*. At present no support is provided by the environment to automatically recover from an unsafe state. This is left to the user.

We view the present work as a preliminary exploration into the problem. We tried to provide a simple formal framework based on logic to deal with it, and we implemented a prototype environment (called SENTINEL) to verify feasibility of the approach. Future work includes both validating the approach and extending it to PMLs that are based on different paradigms (like SLANG[1][3, 11]).

The paper is organized as follows. Section 2 provides some basic definitions and more motivations for the chosen formal approach. Section 3 introduces the language (called LATIN, LAnguage Tolerating INconsistencies) chosen to explore the issue of tolerating divergency between the formal process model and the observed process. Section 4 shows how deviations can be handled. This is done by recording histories of relevant events and encoding the formal process model in an underlying temporal logic, called PDTL (Process Description Temporal Logic). Section 5 shows an example of a simple process model. Section 6 presents the architecture of the SENTINEL environment. Finally, Section 7 discusses related work, open issues, on-going and future work.

2 Overview of the approach

Our reference model has been inspired by previous work on formal specification for secure and real-time systems [9, 7]. The enacted process is viewed as a set of state machines evolving through sequences of state transitions.

A formal *process model* is composed of two parts: *process constraints* (PCs) and *process description*. The process constraints state the critical requirements for the process; for example, a security policy that two programmers can never edit the same module at the same time. The process description defines a set of *taskTypes*.

The run-time representation of the process model is the *process model instance*, composed of a set of *task instances*[2], concurrently executed, that are dynamically instantiated from the corresponding taskTypes. Each taskType is modeled as a state machine. State transitions are atomic; they are characterized by a *precondition*, called ENTRY, and a *body*. The ENTRY defines a property that has to be satisfied before executing the corresponding transition. In the body, two different classes of operations can be performed: *actions*, and *value assignments*. Actions produce results into the external environment (a typical action might consist of launching of a tool). Value assignments modify the values of a task state variables.

We express the process constraints as a set of *invariants*. One of them (the *global invariant*) defines critical requirements for the whole process. The others are defined at taskType level: they define some correctness constraints on the execution of the task instances. The process constraints are satisfied at a certain time if and only if the conjunction of the global invariant, and of all of the task-level invariants is proved to hold at that time for all task instances.

A formal process model is *correct* if, in every reachable state of the corresponding task instances, the process constraints are satisfied. This can be proved inductively by showing that:

- The initial state satisfies the process constraints.
- For each transition, if its precondition and the process constraints are satisfied, then the process constraints are satisfied upon completion[3].

From now on, we only consider correct process models, i.e., we suppose that some (here unspecified) verification step can prove the formal process model to be correct.

Let P be a formal process model, that is provided to carry out some software development task. As we mentioned, in reality, not all observed processes can be forced to be instances of P. Observed processes that are instances of P are correct (since the process model is assumed to be correct). All the other observed processes may be classified in two classes: *feasible processes* and *unacceptable processes*.

An observed process is *feasible* if, in any reached state, the process constraints are verified, but some transition is executed under false precondition. An observed process is *unacceptable* if the process constraints are violated in some reached state. Our goal is to provide automatic support to correct and feasible processes, monitoring their execution in order to fix their state, as they become unacceptable.

Observed processes are assumed here to be supported by SENTINEL, which creates a process model instance from a process model and starts its enactment. SENTINEL enacts a process model instance by executing the transitions whose precondition is true. Moreover, it determines changes in the process model instance state through the actions defined in the body of the transition being executed. User interaction with the process model instance occurs through a special kind of transition: the *exported transition*. Exported transitions are defined as any other transition, i.e., they are characterized by a precondition and a body. They differ from normal transitions in the way they are executed by the PSEE:

- A normal transition is executed only if its ENTRY becomes true at the execution time[4].
- An exported transition can be executed only if the user requests it. If the corresponding ENTRY is not verified, the user may still force its execution. In such a case, we say that the transition *illegally fires*.

Deviation of observed processes from process model instances occurs if a user forces an exported transition to fire when its precondition is false. As a deviation occurs, it is not guaranteed that the state entered by the observed process satisfies the process constraints. Thus, process constraints satisfaction must be verified at each step of process enactment after occurrence of an illegally fired transition. If the process constraints holds in the new state entered by an illegally fired transition, then the observed process is feasible,

[1] The major software process research effort currently undertaken in our group.

[2] In the following, the term *task* will indicate a task instance, if no ambiguity arises from the context.

[3] Transitions are viewed as atomic entities. States entered while a transition is being executed are considered invisible.

[4] If more than one transition in a task has true precondition, one is chosen nondeterministically.

otherwise, the observed process is unacceptable. In the first case, the PSEE continues to support process enactment. In the second case, the values of some state variables are corrupted, and should be fixed in order to continue the enactment in a safe way. To fix corrupted data, illegal firings have to be recognized, and potentially polluted data have to be identified. This is done by recording the history of behavior of the process model interpreter, and by reasoning on this history through some predefined rules (see Section 4).

3 Definition of the LATIN language

A LATIN process model is composed of a *global part* and a set of *taskTypes*.

The global part of a process contains global types and variables visible in the whole process model, a global invariant, the declaration of the taskTypes that will be instantiated during the enactment, and the description of the *main task*. When the process enactment starts, an interpreter for the main task is created. All the other tasks are instantiated by the main task or, in turn, by previously instantiated tasks.

A *taskType* is composed of the following parts:

- A *header*, which defines the name of the taskType and its parameters. Parameters are used during task instantiation in order to assign an initial state to the task instance.
- An *import section*, which lists all variables imported from other taskTypes. An imported variable is a read-only variable for the instances of the importing taskTypes.
- A *declaration section*, which declares local types and variables.
- An *export section*, which lists the names of all the variables exported to other tasks.
- An *init section*, which lists all initial values (if any) to be assigned at instantiation time.
- A set of *transitions*.
- An *invariant*.

LATIN process data are represented by variables. All LATIN variables are typed. Both global and local types can be either *taskTypes* or *basic types*. A variable of type taskType contains a reference to a task instance, and can be used to access the values of the exported variables of the task instance by using the dot notation: `TaskTypeVar.ExpVar`. After the termination of the task instance to which it refers, a variable of taskType allows to access the final values assumed by the exported variables of the task instance. Basic types can be standard (INTEGER, REAL, STRING, BOOLEAN, ENUM) or user-defined (RECORD and SET). To access fields of records the usual dot notation is used.

Variables can be *state variables*, specifying the state of a task, or *quantified variables*. Quantification can only be applied to sets. Quantified variables do not need to be declared in the declaration section; they are local to a transition, and their type is determined by the type of the elements of the set to which quantification is applied.

Transitions are composed of the following parts:

- a header, which defines the name of the transition and, if needed, its parameters,
- an ENTRY clause, that is a LATIN well-formed formula, evaluated at the starting time of the transition,
- a body, composed of:
  - a declaration of local variables used in the transition,
  - an optional ACTION clause, containing tool invocations and new task instantiations,
  - an EXIT clause, containing a list of value assignments to the state variables. Conceptually, the EXIT clause is executed in a single step at the end of transition.

LATIN formulas used in the ENTRY clause are logic formulas, evaluated during enactment. Their truth value depends on the values held by the state variables. If some uninitialized variable appears in the ENTRY, the ENTRY is not evaluated. The formalism used for formulas in ENTRY clauses is a first-order logic augmented with the operators PAST, LASTSTART, LASTEND[5], with the following semantics:

- `PAST(f, t)` is true iff the well-formed formula `f` was true at time `t`.
- `LASTSTART(TransName, t)` is true iff the last start time of transition `TransName` occurred at time `t`.
- `LASTEND(TransName, t)` is true iff the last end time of transition named `TransName` occurred at time `t`.

Term `t` represents a value of the numeric set TIME, the temporal domain. TIME contains all time values (for simplicity, integers) up to the current instant. It is automatically updated during process enactment. It is possible to quantify over TIME. For example:

```
EXISTS t IN TIME (PAST(team.numberOfMembers>5, t))
```

is true iff at a time `t` in the past the field `numberOfMembers` of the state variable `team` was greater than 5. Formula:

```
LASTSTART(taskRef.TransName, 7)
```

is true iff, at time 7, transition `transName` of the task instance referred by `taskRef` fired, and there where no later starts of that transition.

In the ACTION clause, a taskType instance can be created by using the `NEW` construct. Task instantiation is performed by spawning a new interpreter that will execute the task. All active tasks are executed concurrently.

In the ACTION, tool invocation is obtained by using the `CALL` construct. For example:

```
ACTION CALL(``vi'' + fileName);
```

it is interpreted by the PSEE as a request of executing the editor `vi` opening the file whose name is contained in variable `fileName`. The parameters of `CALL` are executable Unix shell commands[6].

[5] These operators are similar to those used in ASTRAL [7]. Their purpose is to allow preconditions to be stated as a condition on the past history.

[6] At present LATIN does not provide fine grained integration mechanism for tools. They will be provided in the future.

Exported transitions are distinguished by the keyword EXPORTED. Their structure is the same of the normal transitions, but they are triggered by events in the external environment, and can be forced to be executed even if their ENTRY is false. Moreover, they can have parameters whose values must be specified by the external environment at firing time.

Invariants are well-formed formulas, having the same syntax defined for the ENTRY. They are used to specify the process constraints. In order to determine formal process model correctness all the variables which appear in the invariants have to be initialized. Invariants may be considered similar to the law introduced by Minsky in [10]. Like Minsky's law, invariants have always to hold. In our case invariants truth is verified and not enforced.

To guarantee transition atomicity, global variables are managed using the usual locking mechanism. When a transition accesses a global variable to modify its value, the variable cannot be accessed by the other transitions.

4 Enactment and pollution analysis

During enactment, the relevant events that occur in the process are stored in a knowledge base. The knowledge base is accessed by process interpreters in order to evaluate preconditions and invariants. It is also accessed in order to perform pollution analysis, i.e., to identify polluted data, illegally fired transitions, and illegally started tasks (i.e., task instances created by illegally fired transitions). Pollution analysis is performed starting from the sequence of relevant events and using a set of *pollution rules*, that defines all the possible cases for which the events are potentially incorrect, being inconsistent with the process description.

Pollution analysis is based on a logic reasoning mechanism. For this reason, the relevant events, the pollution rules, and the relevant information concerning the process model are expressed as formulas of a first-order temporal typed logic called PDTL. In PDTL a temporal domain exists, called $TIME$, which is a finite numeric set. $TIME$ includes all integer values in the range 0 (associated with the first event occurrence in the system), Now (representing the present time). The PDTL formula:

$$AT(\phi, t)$$

where ϕ is a PDTL wff, is true if and only if ϕ is true at time t.

From analysis of pollution data, the process modeler may decide how to reconcile process model and observed process. In this paper we do not deal with reconciling mechanisms. This issue will be a matter of future work. Instead, we will concentrate on the definition of the relevant events for the process, and of pollution rules.

4.1 Relevant events

LATIN preconditions and invariants are temporal formulas, in which temporal operators PAST, LASTSTART, and LASTEND can be used (see Section 3). Therefore, to enact LATIN process models, we need to store in a knowledge base all the events that can be used in these formulas. The relevant events are:

1. the starting time of each transition firing,
2. the ending time of each transition firing,
3. the start time of each task instance, along with a reference to the transition that instantiated the task,
4. the termination time of each task instance,
5. all value assignments for each state variable, along with the time of the assignment.

Relevant events occurrences are represented in PDTL by the following formulas:

Re1. AT(Start(tr), t)

Re2. AT(End(tr), t)

Re3. AT(StartedTask(task, tr), t)

Re4. AT(EndedTask(task), t)

Re5. AT(Assigned(var, tr, value), t)

where tr, $task$, and var are transition, task, and variable identifiers respectively, t is a temporal value and $Start$, End, $StartedTask$, $EndedTask$ and $Assigned$ are PDTL predicates, whose meaning is intuitive.

Relevant events are used also when deviations make the process unacceptable, and repairing actions have to be performed. In this case, from their occurrence it is possible to understand what was wrong. In the following sections we discuss about how identify a deviation, and its generated pollution.

4.2 Deviations and inconsistency

When an observed process becomes unacceptable, the defined knowledge base does not satisfy the process constraints. This means that some tolerated deviations produced incorrect data. These incorrect data have to be identified and isolated, and have to be fixed in a proper way.

If no deviations between observed process and process model is allowed to occur, the following formula has to hold:

$$\forall tr \in transitionId(Start(tr) \Rightarrow ENTRY(tr)) \qquad (1)$$

where $transitionId$ is the domain of transition identifiers, and $ENTRY(tr)$ is a PDTL predicate whose value is true iff the ENTRY of transition tr is true. Formula (1) formalizes the fact that, given the correctness of the process model, an observed process is correct when transition firing occurs only if the associated ENTRY is true.

Feasible processes, as we said, can violate constraint (1). A knowledge base containing both (1) and relevant event occurrences of feasible processes would be inconsistent. Formally, inconsistency of a knowledge base implies that it is possible to derive both a fact A, and its negation $\neg A$. This makes the knowledge base completely useless. Two different approaches are commonly used to deal with inconsistency. The first simply forbids inconsistency to happen by avoiding constraint violations. The second eliminates constraints. Both solutions are unacceptable in our case. In fact, we want to tolerate deviations and, at the same time, we want to identify the transitions that produced deviations (i.e., we cannot eliminate constraints).

Our solution has been inspired by the approach proposed by Balzer [2]. Balzer relaxes a constraint Q by substituting it for a new formula, stating that a new condition P, called

Pollution Marker, holds if and only if Q does not hold. The new formula has the form:

$$P(x_1, \cdots, x_n) \Leftrightarrow \neg Q(x_1, \cdots, x_n) \qquad (2)$$

Given that constraint Q instantiates some variables $x_1, \cdots, x_n$, the same variable instances appear in condition P.

The advantage of this approach is that the new constraint is never violated. Moreover the truth of Pollution Marker P indicates the original constraint violation and identifies data $(x_1, \cdots, x_n)$ that violated it. In this way it is possible to apply access restrictions to polluted data, and identify methods for inconsistency resolution.

Our extension of Balzer's proposal allows the process to continue execution as long as process constraints are not violated, but, at the same time, takes care of the problem of pollution propagation. This solution is technically presented in the next section.

4.3 Pollution Rules

Pollution is propagated if polluted variables are used to assign a value to other variables, or if they affect the execution of a transition, when they are used in the evaluation of the corresponding ENTRY. The idea is that all of the variables whose value is affected by a deviation are potentially incorrect, and, therefore, the operations performed using these variables are unreliable. If a polluted variable A is used to assign a value to another variable B, then B becomes polluted too. If a polluted variable is used in the evaluation of an ENTRY, the truth value of the ENTRY is unreliable. If the corresponding transition is executed, it is considered to fire illegally, and all of the variables whose value is changed by it become polluted.

Pollution can also be propagated to an entire task. In fact, a task instantiated during an illegally fired transition, is illegally started. All its transitions will fire illegally and all of the variables they will affect will be polluted.

When the value of a polluted variable is modified during a legal execution of a transition, and no polluted variables are used in the value assignment, then the variable returns to a "clean" state. If its present "clean" value is used, no pollution propagation occurs. However, if its previous value is used, it still causes pollution propagation[7].

By following the Balzer's approach (i.e., no pollution propagation is performed), according to the schema of Formula (2), Formula (1) would be transformed into:

$$\forall t \in TIME, \forall tr \in transitionId(AT(Illegal(tr), t) \Leftrightarrow \neg AT(Start(tr) \Rightarrow ENTRY(tr)))$$

Since we take pollution propagation into account, we obtain more complicated formulas, that we call *pollution rules*:

Pr1.

$$\begin{aligned}
&\forall t \in TIME, \forall tr \in transitionId \\
&(AT(Illegal(tr), t) \Leftrightarrow \\
&(AT(Start(tr), t) \wedge \\
&(\neg AT(ENTRY(tr), t) \vee (AT(ENTRY(tr), t) \wedge \\
&(\exists x \in varId(IsUsedIn(ENTRY(tr), x)) \wedge \\
&AT(Polluted(x), t))) \vee \\
&\exists tId \in taskId(IsTransitionOf(tr, tId) \wedge \\
&IllegallyStarted(tId)))))
\end{aligned}$$

Pr2.

$$\begin{aligned}
&\forall t \in TIME, \forall var \in varId \\
&(AT(Polluted(var), t) \Leftrightarrow \\
&\exists tr \in transitionId(LastAssigned(var, tr, t) \wedge \\
&(\exists t_1 \in TIME, t_1 \leq t \\
&(LastFiring(tr, t, t_1) \wedge AT(Illegal(tr), t_1)) \vee \\
&\exists var2 \in varId(UsedToAssign(var, var2, tr) \wedge \\
&AT(Polluted(var2), t))))))
\end{aligned}$$

Pr3.

$$\begin{aligned}
&\forall t \in TIME, \forall task \in taskId \\
&(AT(IllegallyStarted(task), t) \Leftrightarrow \\
&\exists tr \in transitionId, \exists t_1 \in TIME, t_1 \leq t \\
&(AT(StartedTask(task, tr), t_1) \wedge \\
&(\exists t_2 \in TIME, t_2 \leq t_1 \\
&(LastFiring(tr, t_1, t_2) \wedge AT(Illegal(tr), t_2)))))
\end{aligned}$$

Formula (Pr1) specifies all the conditions for illegal firing of the transitions. Predicate $IsUsedIn(ENTRY(tr), x)$ is true if variable x appears in the ENTRY of transition tr. Predicate $IsTransitionOf(tr, tId)$ is true if transition tr is a transition of the task instance tId. Together with predicate $ENTRY(tr)$, they represent the static process description, and are automatically obtained from the LATIN process description[8].

Predicate $Polluted(x)$, defined by Formula (Pr2), is true if state variable x is polluted. $LastAssigned(var, tr, t)$ is a derived predicate, that is true if the variable var has been assigned for the last time by transition tr at time t. $LastFiring(tr, t, t_1)$ is true if tr started its last execution before t at time t_1. $UsedToAssign(var, var2, tr)$ is a predicate derived from the static information on the process model that is true if $var2$ is used to assign a value to var, in the EXIT part of the transition tr.

In order to improve the granularity of pollution analysis, the fields of LATIN state variables of type RECORD are considered as independent PDTL variables. Thus, according to Formula (Pr2), single record fields can become polluted, and can propagate pollution.

Concerning state variables of type SET, illegal transitions and pollution propagation do not cause the marking of the whole set as polluted, but rather the marking of the operation of adding and deleting elements from the set. In PDTL, two predicates: $AddPolluted(s, v)$ and $DelPolluted(s, v)$ model the fact that the insertion (deletion) of the element v into (from) the set s is polluted. Formulas describing such predicates are similar to Formula (Pr2).

Predicate $IllegallyStarted(tId)$, defined by Formula (Pr3), specifies the conditions under which a task is illegally

[7] We remind that preconditions and invariants may use previous values of a variable (see Section 3).

[8] Note at this point that the PDTL knowledge base represents not only the relevant event occurrences and the pollution rules, but also static process model information.

started. $StartedTask(task, tr)$ is true if task instance $task$ was created by the execution of the transition tr.

For the sake of simplicity, Formulas (Pr1) and (Pr2), as presented here, are incomplete. They become more complex when we take into account the fact that, in an ENTRY, quantifications on state variables of type SET and time-dependent predicates are permitted. For example:

1. A precondition may contain quantification, as in:
```
ENTRY
  EXISTS programmer IN team
        (programmer.qualification = senior AND
         programmer.availTime = 0.20)
```
2. A precondition may also contain a universal quantification, as in:
```
ENTRY
  FORALL programmer IN team
         (programmer.salary < 4000)
```
3. Finally, as an example of a precondition with PAST operator, consider the following case:
```
ENTRY
  EXISTS t IN TIME (PAST(x > 3, t))
```

We informally discuss here how Formulas (Pr1) and (Pr2) may be changed to deal with the more general case. In example 1 the transition fires illegally not only if no element of the set `team` satisfies the condition, but also if all the elements for which the ENTRY is true are potentially incorrect; that is, the predicate $AddPolluted(team, element)$ is true for all of them. It can also be the case that the transition fires correctly, but the element selected by the quantifier is polluted and it is used to assign a value to some variable in the EXIT of the transition. This causes pollution propagation.

In case 2, the transition fires illegally if precondition is not satisfied or if it is satisfied but at least one element which does not satisfy condition `programmer.salary < 4000)` was illegally deleted from set `team` (predicate $DelPolluted(team, element)$ true). This is because if the deleted $element$ were in the set $team$, the condition 2 would not be satisfied.

In case 3, the transition fires illegally if there is no time t such that the condition $x < 3 \wedge \neg Polluted(x)$ is true.

Pollution analysis is performed by reasoning on the knowledge base composed of formulas (Re1), ..., (Re5), formulas (Pr1), (Pr2), (Pr3), and formulas expressing the static information on the process model. The theorem prover which evaluates the effect of pollution propagation is based on a decidable procedure, i.e, pollution analysis is performed mechanically. The main advantage of this approach is that no pollution rule is hard coded in some algorithmic way. Instead, they are expressed as logic formulas, and can be modified to express different policies to evaluate pollution.

5 An Example

In a software factory, the following configuration management policy is pursued. A repository contains the released modules of a project. Each released module has a version number and an ancestor (the module from which it has been derived). Modules can be added or extracted from the repository. Each time a module is checked out, the last version of the module in the repository is supplied. If version `V` of a module is checked out from the repository and is modified, a subsequent check in will add to the repository a new version, having `V` as ancestor. Initially, the version number of a new module is 1, and it has no ancestors.

In LATIN each module is modeled by a RECORD of type `module`:

```
module IS RECORD OF
  name: STRING;
  version: INTEGER;
  ancestor: INTEGER;
  text: STRING;
  busy: BOOLEAN;
END RECORD;
```

Field `busy` is true if the module has been checked out. Its use will be clarified in the following. `ancestor` represents the version number of the ancestor module.

The repository is modeled as a SET containing elements of type module:

```
releasedModules IS SET OF module;
```

Suppose one wishes to enforce a (simplistic) configuration management policy stating that it is not possible to enter two different modifications of the same module in the repository. This constraint represents the process constraints for our example, and can be formalized in LATIN by the following invariant:

```
FORALL m1, m2 IN releasedModules
  (m1 <> m2 <=> m1.ancestor <> m2.ancestor)
```

that is, different modules in `releasedModules` must have different ancestors.

The operations for checking a module in and out are modeled by two state transitions. These transitions are both exported in order make them visible to the users. Figure 1 shows the taskType `modifyMod`, in which transitions `check-in` and `check-out` are defined. The taskType can contain other transitions that, for example, allow the users to edit a module which was checked out from the repository. For simplicity, we decide to ignore these transitions.

Let us look at transition `check-out`. The parameter `m` is supposed to be a module name, which is supplied by the user when the transition is executed. The precondition holds if a module named `m` exists in the repository, whose last version has not been checked out. In the EXIT of the transition, the last version of the module is deleted from the repository, and assigned to variable `MyVersion` (a state variable of taskType `modifyMod`). Finally, the module is reinserted into the repository with the field `busy` equal to TRUE. In such a way, further attempts to access the module will be incorrect. Field `busy` becomes false again when the module is checked into the repository.

The precondition of `check-in` is always true, that is, this transition is always enabled. In the EXIT clause, if the module to be inserted is a new version of a non-empty series of revisions, the module version from which the current revision has been produced is extracted, its field `busy` is set to FALSE, and then it is reinserted in the repository. Module `MyVersion` is then added to the repository as a new version. Note that the `EXTRACT` operation requires three parameters:

```
TASK_TYPE modifyMod;

VARIABLE
  MyVersion: module;

INITIAL
  MyVersion.ancestor BECOMES 0;
  MyVersion.version BECOMES 1;

EXPORTED_TRANSITION check-out (m: STRING)
ENTRY
  EXISTS mod IN releasedModules
   (NOT EXISTS mod2 IN releasedModules
     (mod.version < mod2.version) AND
      mod.name = m AND NOT mod.busy)
EXIT
  Del(releasedModules, mod);
  MyVersion BECOMES mod;
  MyVersion.busy BECOMES TRUE;
  Add(releasedModules, MyVersion);

EXPORTED_TRANSITION check-in
ENTRY TRUE

LOCAL_VARIABLE
  temp: module;
EXIT
  EXTRACT (releasedModules, temp,
           temp.name = MyVersion.name AND
           temp.version = MyVersion.version);
  IF temp <> Nil THEN
    temp.busy = FALSE;
    Add(releasedModules, temp);
  ENDIF
  MyVersion.ancestor BECOMES MyVersion.version;
  MyVersion.version BECOMES MyVersion.version + 1;
  MyVersion.busy BECOMES FALSE;
  Add(releasedModules, MyVersion);

...
```

Figure 1: The taskType `modifyMod`.

the name of a set, the name of a variable, and a condition. As result, an element of the set verifying the condition is assigned to the variable. If no element in the set satisfies the condition, a conventional value `Nil` is assigned to the variable.

It easy to verify that the process description satisfies the formal model. To do so, it introduces certain restrictions. E.g., it does not allow two different users to check out the same version of a module, work on separate revisions, and, later, check in the version obtained by merging the modifications. In our environment, we can tolerate this deviation as it does not cause process constraints violation.

Imagine for example the following scenario. Two task instances (`task1` and `task2`) of taskType `modifyMod` are created. `task1` supports user Jack; `task2` supports user John. Jack checks out module "`userInterface`", and starts working on it; i.e., he fires transition `check-out`, locks the last version of the module, and assigns it to the variable `MyVersion`. Suppose that the version of the checked out module is 2.

Meanwhile, the project manager decides that module "`userInterface`" is critical for the project, and decides to assign another programmer, John, to work on a modification of a different part of the module. Starting his work, John requires the execution of transition `check-out` of task instance `task2`, with "`userInterface`" as a parameter. The precondition of this transition is false (since `busy` is TRUE). John, however, still forces the execution of the transition, and starts modifying its private copy of the module. It is easy to prove that this violation of the precondition does not cause any violation of the invariant (i.e., the process constraints are satisfied).

John and Jack both know that they are working concurrently. The project manager asks Jack to merge the two versions of the module in order to submit the resulting version to the repository.

As a new version is obtained as a result of merging, Jack fires transition `check-in` of the task instance `task1`. This transition fires correctly, being its precondition true. The invariant also continues to be true.

Should we apply pollution analysis at this stage, we would infer that the `check-out` transition of task `task2` illegally fired, and that variable `MyVersion` of the same task, and version 2 of module "`userInterface`" can cause pollution propagation. This is formally represented by the following formulas:

1. $AT(Illegal(task2.check-out), t1)$
2. $AT(DelPolluted(global.releasedModule,$ $\{name = userInterface, version = 2,$ $ancestor = 1, text = ..., busy = TRUE\}), t2)$
3. $AT(Polluted(task2.MyVersion), t2)$
4. $AT(AddPolluted(global.releasedModule,$ $\{name = userInterface, version = 2,$ $ancestor = 1, text = ..., busy = TRUE\}), t2)$
5. $AT(Polluted(task1.check-in.temp), t3)$
6. $AT(AddPolluted(global.releasedModule,$ $\{name = userInterface, version = 2,$ $ancestor = 1, text = ..., busy = FALSE\}), t3)$

From this situation, if a `check-out`/`check-in` sequence of transitions is executed on module ``userInterface'', all

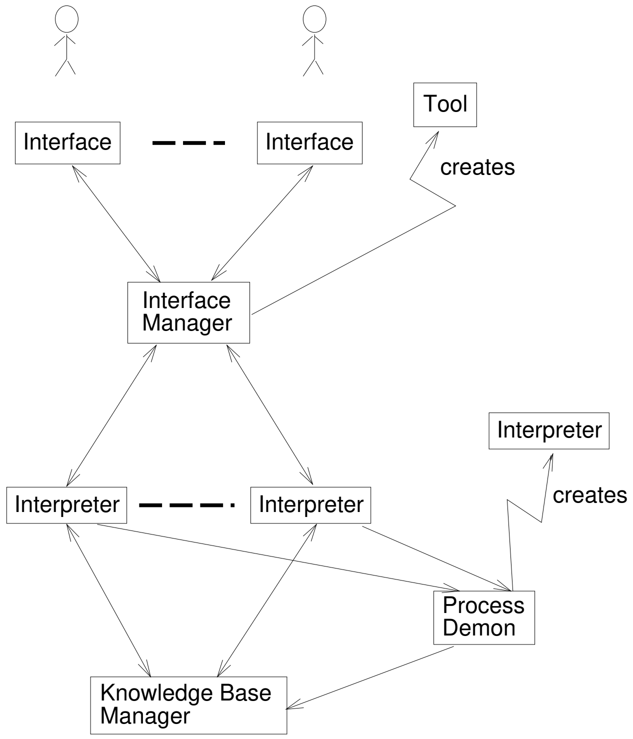

Figure 2: SENTINEL architecture.

transitions fire correctly and will not cause pollution propagation; i.e. our method has been able to tolerate a temporary deviation and then return within the boundaries of the process description.

However, if Jack decides to check in his version of the module, an invariant violation occurs, (version 2 of "`userInterface`" will have two children) and some repairing actions have to be performed. In this new state, pollution analysis gives the following results, in addition to the previous ones:

7. $AT(Polluted(task2.check-in.temp), t4)$
8. $AT(AddPolluted(global.releasedModule,$ $\{name = userInterface, version = 3,$ $ancestor = 2, text = ..., busy = FALSE\}), t4)$

At this point, if we decide that Jack's action should be permitted, we need to change the process model. In particular, the configuration management policy should be changed by allowing a tree-structured version organization instead of a simple linear structure.

6 SENTINEL Architecture

SENTINEL (Software Engineering eNvironment Tolerating INconsistencies Employing Logic) is the prototype PSEE developed to experiment with the enactment of LATIN process models. Its architecture is shown in Figure 2. The Knowledge Base Manager (KBM) is the kernel of the system. It maintains the process information history introduced in Section 4.

Each active task instance has an associated interpreter. Interpreters evolve according to the state transitions defined for the tasks. They ask the Knowledge Base Manager (KBM) to evaluate the ENTRYs of the transitions before executing them. If the instantiation of a new task is required in the ACTION part of a transition, the interpreter sends to the Process Demon a request to instantiate a new interpreter. The Process Demon spawns a new interpreter, assigns it a task identifier, and communicates the creation of the new task instance to KBM. Upon completion of the EXIT part of a transition, the interpreter notifies the new value of the state variables to KBM.

The interaction with the external environment is performed through the Interface Manager (IM). IM hides to the interpreters all the problems related to the distribution of the users, working on different workstations. In fact, it manages the interfaces through which the users can ask for the execution of the exported transitions of the active tasks, and is in charge of executing tools when requested by the interpreters.

After the occurrence of an illegally fired transition, at the end of each transition execution, the interpreter requires KBM to verify the invariants. The enactment is suspended if an invariant is violated, and KBM starts pollution analysis. Pollution analysis can also be required by the users, through their interfaces.

7 Related work and conclusion

The present work is a preliminary exploration into the problem of improving the usability of PSEEs by tolerating deviations from the process description. So far there has not been much work dealing with this issue.

[5] and [4] provide two different attempts to define a framework for dealing with process evolution. In particular, [5] provides a first distinction, widely adopted, among *process definition*, *process performance*, and *process definition enactment*. Process definition corresponds to what we called process model, process performance corresponds to the observed process, and process definition enactment corresponds to the process model instance. In the paper, the possibility for the process performance to deviate from the process definition is recognized, but mechanisms for dealing with this problem are not discussed.

In [4] three basic mechanisms for dealing with "on the fly" deviations are defined. The *bypass mechanism* allows the observed process to perform operations that are not allowed by the process model; the *triggering mechanism* catches the events that produced deviation; the *reconciling mechanism* supports the reconciling of the observed process with the process model. In our environment, the bypass is the mechanism that allows to fire a transition even if the corresponding precondition does not hold. The triggering mechanism is the one that supports pollution analysis, and the reconciling is the mechanism that supports the users in fixing pollution, supplying the list of polluted data.

[1] deals with the inconsistencies caused by the uncertain and incomplete knowledge in observed process. In the paper, a goal-oriented language is defined (PEACE), which formalizes parts of a process model using an autoepistemic logic. In this way, the user's beliefs can be modeled, and it is possible to reason with beliefs that change during the progress of the process.

Finkelstein et al. [6] deal with inconsistency in the domain of multi-perspective specifications. They view the specification of a system as a set of *ViewPoints*. Each ViewPoint is the specification of part of the system, seen from a particular domain. ViewPoints are related among them by *inter-ViewPoint* relations, that "...specify dependencies and mappings between system components". In general, ViewPoints are loosely coupled distributed objects, and it is difficult to maintain consistency among them. Therefore, it is necessary to handle inconsistency, by performing repairing actions at specific check points. As we do, they use a logic approach to deal with inconsistencies. In particular, they translate the "partial information knowledge" in the ViewPoints in a first order logic, together with the inter-ViewPoint relations. In this way, inconsistencies can be detected, and meta-level rules can be invoked to perform the repairing actions. Meta-level rules are formalized in a temporal logic. Their use implies the knowledge of the history of the data in the ViewPoints, and of the previous repairing actions on the inconsistent data. Our pollution rules can be seen as meta-level rules. We assume that repairing from inconsistency requires the human intervention. Therefore, the actions in the meta-level rules detect the data that are potentially incorrect. This information is supplied to the users, who are supposed to fix the divergence with specific repairing actions, depending on the state of the observed process.

Our approach allows deviations to be tolerated as long as they do not affect the correctness of the system. When the system enters an incorrect state, i.e., the observed process becomes unacceptable, two possible solutions can be pursued. The first one is to fix the incorrect data generated by the deviation, i.e., to constrain the observed process in the boundaries of the process model. The second one is to change the process model, or its active instance. In both cases pollution analysis offers a good (even if low-level) support, by identifying and isolating pollution.

Based on our experience and on the literature on the subject, we feel that a logic approach has potential in detecting inconsistencies and deviations. By defining the pollution rules, we basically define a policy that tells us when and what data can be considered potentially incorrect. Defining new pollution rules corresponds to adopting different policies.

Although we have done some experiments with our initial prototype of SENTINEL, more experimentation of our approach will be done in the future. We also plan to reengineer the architecture of SENTINEL, in order to eliminate the bottleneck of the centralized Knowledge Base Manager, and improve the mechanism of interaction with the external environment. Further work is also needed on the issue of reducing the number of events to be stored in the knowledge base. Finally, once the approach will be fully validated, we still study how to implant the approach in the context of different formalisms.

References

[1] S. Arbaoui and F. Oquendo. PEACE: Goal-oriented Logic-based Formalism for Process Modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*. Research Studies Press, J. Wiley, 1994.

[2] Robert Balzer. Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering–ICSE 13*. IEEE Computer Society, 1991.

[3] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[4] Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Policies and Mechanisms to Support Process Evolution in PSEEs. In *Third International Conference on the Software Process*, Reston, Virginia, USA, October 1994.

[5] M. Dowson and C. Fernström. Towards Requirements for Enacment Mechanisms. In *Third European Workshop on Software Process Technology EWSPT'94*, Villard de Lans (Grenoble) France, February 1994.

[6] A. Finkelstein, D. Gabbay, A. Hunter, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. In *Fourth European Software Engineering Conference (ESEC'93)*, Garmish, Germany, September 1993.

[7] C. Ghezzi and R.A. Kemmerer. ASTRAL: An Assertion Language for Specifying Real-Time Systems. In *Proceedings of the 3th. European Software Engineering Conference (ESEC'91)*, Milan, Italy, October 1991.

[8] Philippe Jamart and Axel van Lamsweerde. A Reflective Approach to Process Model Customization, Enactment and Evolution. In *Third International Conference on the Software Process*, Reston, Virginia, USA, October 1994.

[9] R.A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[10] N. H. Minsky. Law-governed systems. *Software Engineering Journal*, pages 285–302, September 1991.

[11] SPADE Team. Slang 1.1, process modeling user manual. Technical report, CEFRIEL, Via Emanueli, 15 20126 Milano, 1994.