Planlets: Automatically Recovering Dynamic Processes in YAWL

Andrea Marrella, Alessandro Russo, and Massimo Mecella

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza Università di Roma, Rome, Italy
{marrella,arusso,mecella}@dis.uniroma1.it

Abstract. Process Management Systems (PMSs) are currently more and more used as a supporting tool to coordinate the enactment of processes. YAWL, one of the best-known PMSs coming from academia, allows to define stable and well-understood processes and provides support for the handling of expected exceptions, which can be anticipated at design time. But in some real world scenarios, the environment may change in unexpected ways so as to prevent a process from being successfully carried out. In order to cope with these anomalous situations, a PMS should automatically recover the process at run-time, by considering the context of the specific case under execution. In this paper, we propose the approach of Planlets, self-contained YAWL specifications with recovery features, based on modeling of pre- and post-conditions of tasks and the use of planning techniques. We show the feasibility of the proposed approach by discussing its deployment on top of YAWL.

Keywords: Process Management Systems, YAWL, recovery, planning.

1 Introduction

In the last years, the increasing demand in solutions for *dynamic processes* and the need to provide support for flexible and adaptive process management has emerged as a leading research topic in the BPM domain [15,20] and has led to reconsider the trade-off between flexibility and support provided by existing Process Management Systems (PMSs). Research efforts in this field try to enhance the ability of processes and their support environments to modify their behavior in order to deal with contextual changes and exceptions that may occur in the operating environment during process enactment and execution. On the one hand, existing PMSs like YAWL [17] provide the support for the handling of expected exceptions. The process schemas are designed in order to cope with potential exceptions, i.e., for each kind of exception that is envisioned to occur, a specific contingency process (a.k.a. exception handler or compensation flow) is defined. On the other hand, adaptive PMSs like ADEPT2 [19] support the handling of unanticipated exceptions, by enabling different kinds of ad-hoc deviations from the pre-modeled process instance at run-time, according to the structural process change patterns defined in [18].

However, in a dynamic process the sequence of tasks heavily depends on the specifics of the context (e.g., which resources are available and what particular options exist at that time), and it is often unpredictable the way it unfolds. The use of processes for supporting the work in highly dynamic contexts like healthcare and emergency management has become a reality, thanks also to the growing use of mobile devices in everyday life, which offer a simple way for picking up and executing tasks. To deal with exceptions and uncertainty introduced by such contexts, the need for flexible and easy adaptable processes has been recognized as critical [10]. However, traditional approaches that try to anticipate how the work will happen by solving each problem at design time, as well as approaches that allow to manually change the process structure at run time, are often ineffective or not applicable in rapidly evolving contexts. The design-time specification of all possible compensation actions requires an extensive manual effort for the process designer, that has to anticipate all potential problems and ways to overcome them in advance, in an attempt to deal with the unpredictable nature of dynamic processes. Moreover, the designer often lacks the needed knowledge to model all the possible contingencies, or this knowledge can become obsolete as process instances are executed and evolve, by making useless his/her initial effort.

This paper, based on our previous work [11,12,1], introduces the notion of Planlets, as self-contained YAWL nets where tasks are annotated with pre-conditions, desired effects and post-conditions. The main characteristic of a Planlet is in the ability to recover itself - if an exception arises - automatically, without explicitly defining any recovery policy at design-time. This feature is critical for processes executed in dynamic environments where requirements and context can change rapidly and unpredictably. An external planner is in charge of synthesizing the needed recovery procedure on-the-fly, by contextually selecting the compensation tasks from a specific repository linked to the Planlet under execution.

The rest of the paper is organized as follows. Section 2 presents and discusses related works. Section 3 introduces our running example that helps to clarify the scope of the approach. Section 4 presents the general approach and shows how it can be concretely built on top of the YAWL architecture, whereas Section 5 discusses task annotations and the use of planning techniques for automatically recovering dynamic processes. Section 6 reports on experimental evaluation results and Section 7 concludes the paper by discussing limitations and future developments of the approach.

2 Related Works

Recently, techniques from the field of artificial intelligence (AI) have been applied to process management. In [5], the authors present a concept for dynamic and automated workflow re-planning that allows recovering from task failures. To handle the situation of a partially executed workflow, a multi-step procedure is proposed that includes the termination of failed activities, the sound suspension of the workflow, the generation of a new complete process definition and the adequate process resumption. In [9], the authors take a much broader view of the problem of adaptive workflow systems, and show that there is a strong mapping between the requirements of such systems and the capabilities offered by AI techniques. In particular, the work describes how planning can be interleaved with process execution and plan refinement, and investigates plan patching and plan repair as means to enhance flexibility and responsiveness. A new life cycle for workflow management based on the continuous interplay between learning and planning is proposed in [3]. The approach is based on learning business activities as planning operators and feeding them to a planner that generates the process model. The main result is that it is possible to produce fully accurate process models even though the activities (i.e., the operators) may not be accurately described. The approach presented in [13] highlights the improvements that a legacy workflow application can gain by incorporating planning techniques into its day-to-day operation. The use of contingency planning to deal with uncertainty (instead of replanning) increases system flexibility, but it does suffer from a number of problems. Specifically, contingency planning is often highly time-consuming and does not guarantee a correct execution under all possible circumstances. Planning techniques are also used in [4] to define a self-healing approach for handling exceptions in service-based processes and repairing faulty activities with a model-based approach. During the process execution, when an exception occurs, a new repair plan is generated by taking into account constraints posed by the process structure and by applying or deleting actions taken from a given generic repair plan, defined manually at design time.

If compared with the above works, the Planlet approach provides some interesting features in dealing with exceptions: *(i)* it modifies only those parts of the process that need to be changed/adapted by keeping other parts stable; *(ii)* it synthesizes the recovery procedure at run-time, without the need to define any recovery policy at design-time.

3 Running Example

As an application scenario, we consider an emergency management process defined for train derailments and inspired by a real process used by the main Italian Railway Company. The corresponding YAWL process, introduced in [12], is shown in Fig. 1.a. The process starts when the railway traffic control center receives an accident notification from the train driver and collects some information about the derailment, including the GPS location and the number of coaches and passengers. In Fig. 2.a a possible map of the area is depicted as a 4x4 grid of locations. For the sake of simplicity, we supposed that the train is composed by a locomotive (located in *loc(3,3)*) and two coaches (located in *loc(3,2)* and *loc(3,1)* respectively). Then, it may be required to cut off the power in the area and to interrupt the railway traffic near the derailment scene. In parallel, after having collected additional information about the train (e.g., security equipment) and emergency services available in the area, a response team can be sent to the derailment scene. Such a team is composed by four first responders (in the rest of the paper, we refer to them also as *actors*) and two robots,

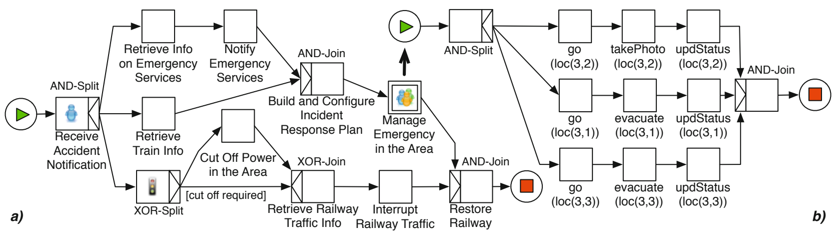


Fig. 1. The YAWL process defined for a train derailment scenario (a), in which the composite task "Manage Emergency in the Area" is a PLANLET (b)

initially located in *loc(0,0)*. We assume that the actors are equipped with mobile devices (for picking up and executing tasks) and provide specific skills. For example, actor $a1$ is able to take pictures and to extinguish fire, whereas $a2$ and $a3$ are in charge of evacuating people from train coaches. The connection between mobile devices is supported by a network provided by a fixed antenna (whose range is limited to the dotted squares in Fig. 2.a), and the robots $rb1$ and $rb2$ can act as wireless routers for extending the network range in the area. A robot provides a connection limited to the locations adjacent (in any direction) to its position. Each robot can move in the area, but it is constrained to be always connected to the main network. This is guaranteed if the intersection between the squares covered by the main network and the squares covered by the robot connection is not empty. A robot connected to the main network can act as a "bridge", allowing the other robot to be connected through it to the main network. Robots have a battery that discharges a fixed quantity after each movement. The actor $a4$, in charge of checking the correct working of the antenna, can change the battery of a robot if empty. Collected information is used for defining and configuring at run-time an incident response plan, defined by a contextually and dynamically selected set of activities to be executed on the field by first responders. Such activities are abstracted into the composite task[1] "Manage Emergency in the Area" (cf. Fig. 1.b). The subnet is composed by three parallel branches with tasks that instruct first responders to act for evacuating people from train coaches, to take pictures and to assess the gravity of the accident. Despite the simple structure of the incident response plan, the high dynamism of the operating environment can lead to a wide range of exceptions. In general, for dynamic processes there is not a clear, anticipated correlation between a change in the context and corresponding process changes. Suppose, for example, that the task *go(loc(3,3))* is assigned to actor $a1$ (cf. Fig. 1.b), which reaches instead the location *loc(0,3)*. This means that $a1$ is now located in a different position than the desired one, and s/he is out of the network range. Since all the actors/robots need to be continually inter-connected to execute the process, the PMS has to find a recovery procedure that first instructs the robots

[1] A composite task is a container for another YAWL sub-net, with its own set of elements.

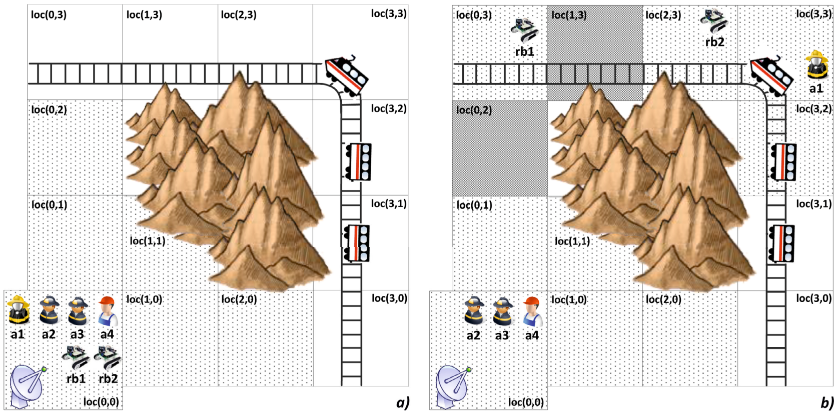


Fig. 2. Area (and context) of the intervention

to move in specific positions for maintaining the network connection, and then re-assign the task $go(loc(3,3))$ to $a1$. It is unrealistic to assume that the process designer can pre-define all possible compensation activities for dealing with this exception (apparently simple), since the process may be different every time it runs and the recovery procedure strictly depends on the actual contextual information (the positions of operators/robots, the range of the main network, the battery level of each robot, etc.). For the same reason, it is also difficult to manually define an ad-hoc recovery procedure at run-time, as the correctness of the process execution is highly constrained by the values (or combination of values) of contextual data.

4 The General Approach and Architecture

4.1 Introducing Planlets

Most of current PMSs are not sufficiently able to deal with dynamic processes, as they do not automatically adapt process instance executions in order to align them to the changes to the environment. In this paper we propose a solution that builds on top of YAWL [17], and consists of annotating at design-time a YAWL specification with additional information which allows process instances to be automatically recovered. In particular, we assume the tasks of a YAWL process specification to be annotated with *pre-conditions*, *desired effects* and *post-conditions*. Failures arise either when associated pre-conditions for a task are not satisfied at the time the task is to be started, or when post-conditions do not hold after the execution of the task. Effects represent the changes that a successful task execution imposes on the *state* of the world, reflecting the current value of the contextual properties that constraint the process under execution. Hence, the process designer just states *what* conditions have to be satisfied, without having to anticipate *how* these can be fulfilled. In order to formalize the concept, we introduce the definition of PLANLET:

Definition 1 (Planlet). *Let YN be a YAWL net, T be the tasks defined in YN, and V be the set of variables defined in YN. Let $Expr(V)$ be the set of expressions over the variables in V. A* Planlet *is a tuple $(YN, Pre, Post, Eff)$ where* (i) *$Pre : T \rightarrow Expr(V)$ returns an expression representing the pre-conditions of tasks in T;* (ii) *$Post : T \rightarrow Expr(V)$ returns an expression representing the post-conditions of tasks in T;* (iii) *$Eff : T \rightarrow Expr(V)$ returns an expression representing the effects of tasks T.*

The role of pre/post-conditions and effects for a YAWL task is twofold: *(i)* pre-conditions and post-conditions enable run-time process execution monitoring and exception detection: they are checked respectively before and after task executions, and the violation of a pre-condition or post-condition results in an exception to be handled; *(ii)* along with the input/output parameters consumed/produced by the task, pre-conditions and effects provide a complete specification of the task: this allows the task to be represented as an action in a planning domain description and used for solving a planning problem built to handle an exception.

At design-time, the annotated tasks are stored in a repository linked to the Planlet specification, which may contain also other annotated tasks deriving from previous executions on the same contextual domain. At run-time, while instances of the YAWL specification are carried on, tasks become enabled. Every time a task $t \in T$ becomes enabled, expression $Pre(t)$ is evaluated; similarly, upon the completion of t, expression $Post(t)$ is evaluated. If an evaluation returns false upon enablement or completion of a task, the system is in an *invalid state* and, hence, the YAWL specification instance needs to be adapted to come back into the "right track". In order to do that, the case execution is suspended, and a recovery procedure is automatically synthesized. To provide more details, let us assume that the current Planlet is $\delta_0 = (\delta_1; \delta_2)$ in which δ_1 is the part of the Planlet already executed and δ_2 is the part of the Planlet which remains to be executed when an exception is identified. The adapted Planlet is $\delta_0' = (\delta_1; \delta_h; \delta_2)$. However, whenever a Planlet needs to be adapted, every running task is interrupted, since the "repair" sequence of tasks $\delta_h = [t_1, \ldots, t_n]$ is placed before them. Thus, active branches can only resume their execution after the repair sequence has been executed. This last requirement is fundamental to avoid the risk of introducing data inconsistencies during a repair.

The automatic synthesis of the recovery procedure δ_h is enacted on-the-fly by an external planner. A planner solves the problem to find a sequence of actions that move a system state from the initial one to a target goal, using a predefined set of admissible actions. Each action is associated the set of pre-conditions $Pre(t)$ in order for that step to be chosen, as well as the effects $Eff(t)$ obtained as result of the action's execution. Along with defining the set of admissible actions, it is also crucial to define how the state is represented, since pre-conditions and effects of actions are given in term of the chosen state representation. The actions' set and the state definition are often referred to as *planning domain.* The standard representation language of planners to define actions and state is the Planning Domain Definition Language (PDDL) [2]. In the context of adaptation of instances of YAWL specifications, each task specification is associated with a different action in the planning domain; the task's pre-conditions and effects are translated in PDDL and associated to the corresponding action. In addition to the so-created planning domain, when an exception arises, the invalid state and the pre-condition (or post-condition) violated is given in input to a planner, which can try to build a plan. If the plan exists, the planner is eventually going to return it. In this case, the plan is converted into a sequence of YAWL tasks which are assigned to qualifying participants. When the converted plan is carried out, the original suspended process is restored for execution.

Let us consider the example introduced in Section 3. The composite activity "Manage Emergency in the Area" may be modeled as a self-contained PLANLET *specification(cf. Fig. 1.b), linked to a repository containing a set of emergency management (annotated) tasks, that range from the simple activity of taking pictures to the more complex extinguishment of a fire. An explicit representation of contextual information (the connection of each actor to the network, the map of the area, the battery charge level of each robot etc.) is needed for preserving the correct* PLANLET *execution. The same exception shown in Section 3 (the actor a1 is not more connected to the network and s/he is in a position different than the desired one) results in a post-condition failure, and now may be easily catched and solved. The planner builds a planning problem by taking as initial state the invalid state of the* PLANLET*, and as goal a state where all actors/robots are inter-connected to the network and a1 is in the desired location loc(3,3). The recovery plan is automatically synthesized by contextually selecting tasks from the repository linked to the* PLANLET*. Suppose, for example, that the two robots rb1 and rb2 have an empty battery. In such a case, the planner devises on-the-fly a possible solution, composed by a sequence of 5 tasks*[2] *[chargeBattery(a4,rb1),move(rb1,loc(1,3)),go(a1,loc(3,3)),chargeBattery(a4,rb2),move(rb2, loc(3,3))] that change the state of the world as shown in Fig. 2.b.*

4.2 Incorporating Planlets into YAWL

The architectural extension and integration we designed takes advantage of YAWL's exception detection capabilities and leverages the flexibility of the exlet-based handling techniques.

Exception Handling in YAWL. The exception handling capabilities provided by YAWL[3] build on the conceptual framework presented in [14]. In order to understand how exceptions are detected and handled in YAWL we refer to the architecture in Fig. 3 (for now, do not consider the *Planning Service* and the PLANLET *Repositories*[4]). For each exception that can be anticipated, it is

[2] The recovery plan is synthesized by taking care of the skills of process participants, and their availability for task assignment and execution. Hence, each task composing the plan is already associated to the participant that will execute it.

[3] In this paper we refer to the final release of YAWL 2.1.

[4] With the exclusion of the *Planning Service* and of the PLANLET *Repositories*, the picture refers to the architecture defined in [17].

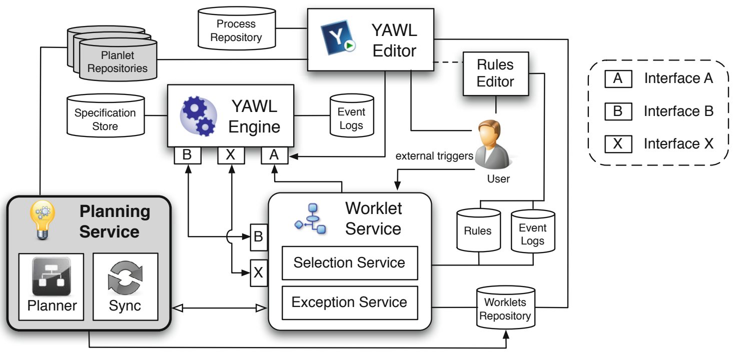
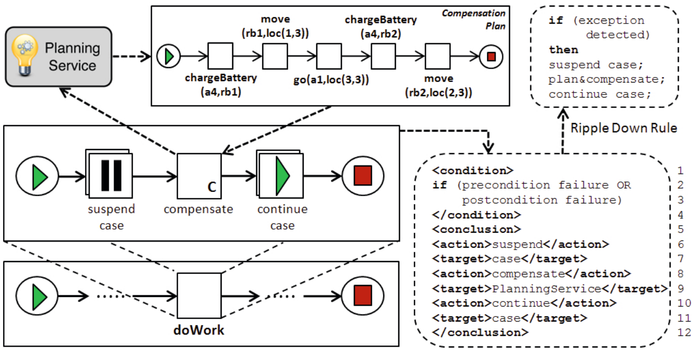


Fig. 3. The YAWL architecture extended with the *Planning Service*

possible to define an exception handling process, named *exlet*, which includes a number of exception handling primitives (for removing, suspending, continuing, etc. a work item/case) and one or more compensatory processes in the form of *worklets* (i.e., self-contained YAWL specifications executed as compensatory processes [17]). Exlets are linked to specifications by defining *rules* (through the *Rules Editor* graphical tool), in the shape of Ripple Down Rules specified as `if` *condition* `then` *conclusion*, where the *condition* defines the exception triggering condition and the *conclusion* defines the exlet. At run-time, exceptions are detected and managed by the *Exception Service* [17]. The service determines whether an exception has occurred and, if so, it executes the corresponding exlet. If the exlet includes a compensation worklet, the service retrieves it from the repository, loads it into the engine and executes it as a new separate case, possibly in parallel with the parent case if it was not suspended by the exlet.

Enabling Planlets in YAWL. From an architectural perspective, as shown in Fig. 3, planning capabilities are provided by a *Planning Service* that implements the planning logic and algorithm. In order to define the role of the *Planning Service* and clarify how it interacts with existing YAWL architectural components and services, we follow the process and exception handling life-cycle, from process design, enactment and monitoring to exception detection, handling and (possibly) resolution. At design time, the process designer builds one or more Planlet *Repositories* (or modifies the existing ones), by inserting/deleting annotated tasks and by (possibly) modifying the contextual domain linked to each repository. Tasks involved in a Planlet specification are selected from a specific Planlet repository, since they are thought to be enacted in a specific contextual domain. Before executing a Planlet, the process designer instantiates the initial values for the properties of the contextual domain. As shown in Section 5, tasks pre- and post-conditions are automatically translated in YAWL pre- and post-constraints. In order to delegate the exception handling to the *Planning Service*, we introduce the possibility of mapping a *compensation activity* to the *Planning Service*. By defining this mapping instead of explicitly selecting a compensation worklet, the process designer configures the *Exception Service* so that



```
if (exception
    detected)
then
suspend case;
plan&compensate;
continue case;
```

```
<condition>                                        1
if (precondition failure OR                        2
   postcondition failure)                          3
</condition>                                       4
<conclusion>                                       5
<action>suspend</action>                           6
<target>case</target>                              7
<action>compensate</action>                        8
<target>PlanningService</target>                   9
<action>continue</action>                          10
<target>case</target>                              11
</conclusion>                                      12
```

Fig. 4. *Planning Service* activation hierarchy for exception handling

the generation of the compensation worklet is delegated to the *Planning Service*. Fig. 4 shows an excerpt of the rule file defined for detecting and handling a workitem-level pre-execution (or post-execution) constraint violation. Lines 1-4 define the exception triggering condition (a pre- or a post-condition failure), while lines 5-12 define the exception handling exlet (which consists of suspending the current case, performing some compensation activities and then resuming the suspended case). In our extended version, the mapping of a compensation task to the *Planning Service* is identified by a `<target>` element containing the `PlanningService` value (line 9), in order to enact planning capabilities. *If we consider our running example, the compensation plan devised in Fig. 4 corresponds to the one needed for re-establishing the network connection between actors/robots and for instructing actor a1 to move in the desired location.*

Planning Service Activation. When the *Exception Service* activates the *Planning Service*, it provides as input all case data associated with the running case, along with the detected violation over pre- and post-conditions. Based on this information, and on the specifications of available tasks, stored in the repository linked to the PLANLET under execution, the *Synchronization* component of the *Planning Service* is able to build the planning domain and to define a planning problem, and submit them to the *Planner* module in charge of synthesizing a recovery plan. If the *Planner* is able to successfully synthesize a compensation plan, it stores it as an executable specification (i.e., a worklet) in the *Worklets Repository* and notifies the *Exception Service*. The *Exception Service* is then able to enact the execution of the compensation worklet as if it was manually selected at design time, by loading the specification into the engine and launching it as a separate case. When the execution completes, output data produced by the worklet are mapped back to the parent case and subsequent actions in the exlet are executed. Following the exlet defined in Fig. 4, as the compensation worklet synthesized by the *Planner* is supposed to recover from the constraint violation, the suspended case can then be resumed and executed. If no valid plan can be found by the *Planner*, a notification alert is sent to an administrator, who is charge of handling the unsolved exception, e.g., manually building a compensation process or just canceling the process case.

5 Annotating YAWL Specifications in Planlets

A main step of our approach in YAWL consists of enriching the process model with a specification of process tasks, in terms of pre-conditions, desired effects and post-conditions, and with an explicit representation of the contextual domain needed for the correct process enactment.

In YAWL, each atomic task t can be linked to a decomposition. Decompositions can have a number of input and output *parameters*, each identified by a *name* and characterized by a *type* dictating valid values it may store, and define the so-called YAWL Service that will be responsible for task execution. As process data are represented through net-level variables, inbound and outbound mappings define how data is transferred from net variables to task variables and vice-versa [17]. We propose to extend task specifications at the decomposition level, with the possibility of defining pre-conditions, post-conditions and effects as logical formulae and expressions over task parameters.

Defining and Representing Finite Domain Types. The definition of a PLANLET requires the specification of the data types that characterize the information manipulated by process instances and define the domains over which predicates and functions are interpreted. In order to have a compact and finite representation of a process state, given by the values assumed by process variables at a given point in the execution, all data types must correspond to *finite* domains over which variables of that type can range; this requirement is imposed by the planning-based approach we propose. Examples of such domains are finite integer intervals or sets of strings, and other enumerated domains. As YAWL applies strong data typing and all data types are defined using XML Schemas, this can be easily achieved by defining data types as XML Schemas and using restrictions (e.g., via the `enumeration` constraint) to limit the content of an XML element to a set of acceptable values. In our example, we need to define data types for representing actors, robots[5] and locations in the area (e.g., data type $Loc = \{loc00, loc10, \ldots, loc33\}$), whose possible values are constant symbols that univocally identify objects in the domain of interest.

Defining and Representing Predicates and Functions. Predicates can be used to express properties of domain objects and relations over objects. A predicate consists of a predicate *symbol* P and a set of *typed parameters* or *arguments*[6]. Argument types (taken from the set of data types previously defined) represent the finite domains over which predicates are interpreted. In our example, we may need predicates for expressing the presence of a fire in a location or whether a location is covered by the network signal provided by the main an-

[5] Although emergency operators and robots can be considered as *resources* or *services* able to execute tasks and can be represented in the organizational model provided by YAWL, we also need to explicitly represent them in the process because we need to define predicates and functions over these domains.

[6] Predicates with no arguments, i.e., with arity 0, are allowed and can be considered as propositions; they are directly represented as boolean variables.

tenna, or relations, such as the adjacency between locations, i.e., $Fire(loc : Loc)$, $Covered(loc : Loc)$, $Adjacent(loc1 : Loc, loc2 : Loc)$.

In addition to basic predicates, we allow the designer to define *derived* predicates. They are declared as basic predicates, with the additional specification of a well-formed formula φ that determines the truth value for the predicate. In our domain, we may need to express that an actor is connected to the network if s/he is in a covered location or if s/he is in a location adjacent to a location where a robot is located (and is thus connected through the robot); assuming we have defined the data types $Robot = \{rb1, rb2\}$ and $Actor = \{a1, a2, a3, a4\}$, we have: $Connected(act : Actor)\,\{\,\text{EXISTS}(l1 : Loc, l2 : Loc, rbt : Robot)\,((at(act) = l1)\,\text{AND}\,(Covered(l1)\,\text{OR}\,(atRobot(rbt) = l2\,\text{AND}\,Adjacent(l1, l2)))))\}$

Numeric and object functions allow to represent and handle numeric values and domain objects as functions of other objects. Function declarations consist of a function *symbol* f, a set of *typed parameters*[7], and a *return type*. Numeric functions have as return type an integer or a real number, whereas object functions have a return type taken from the set of data types defined in the net specification. The arguments of functions range over *finite* domains, and for object functions the same requirement holds for result types. In our example, we need to keep track of the battery level of the robots. This can be represented through the numeric function $batteryLevel(robot : Robot) : Integer$.

Similarly, we can represent the position of actors and robots by defining the following functional predicates that map actors and robots to their location: $at(actor : Actor) : Loc$ and $atRobot(robot : Robot) : Loc$.

State Variables Representation. The use of predicates and functions requires that at run-time we represent the corresponding logical interpretations, as state variables that hold *(a)* the truth value of the defined predicates over domain objects, and *(b)* the values of the defined functions with respect to different argument assignments. The interpretations are used to evaluate pre- and post-conditions, and are modified as a result of task executions. As a consequence of the declaration of a predicate or function, two new data types are automatically generated and added to the XML data types definitions for the net:

T1. a complex data type that is able to represent the name of the predicate or function and
   - for predicates, all argument assignments for which the predicate holds[8] (i.e., the current interpretation $P^{\mathcal{I}}$ for the predicate);
   - for functions, all argument assignments for which the function is defined, along with the corresponding value[9] (i.e., the current interpretation $f^{\mathcal{I}}$ for the function);

[7] Numeric functions with no arguments are allowed, and can be considered as state variables rather than constants, as their value may change during process executions; they are represented as integer or float/double variables.

[8] Basically, a set containing all object tuples for which the predicate is true.

[9] Basically, a map where object tuples are mapped to objects.

T2. a complex data type that is able to represent a predicate or function instance, in terms of the name of the predicate or function, the set of arguments and their assignment, and the truth value or numeric/object value of the predicate or function with respect to the specific assignment; different parameters of this type can be defined for process tasks, to be used for representing the effects that they can have on the predicate or function interpretation.

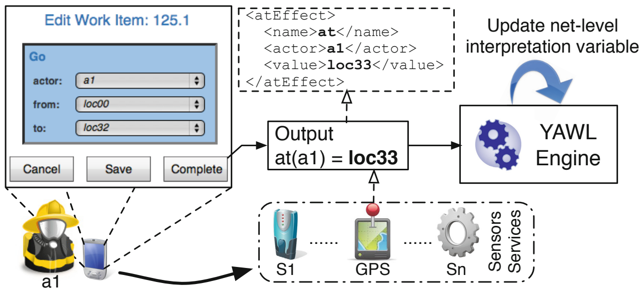
For each predicate and function, a single net-level state variable of type T1 is defined and it can be initialized so as to contain all values for the objects for which the predicate is true or the function is defined in the initial state. Derived predicates are not explicitly represented through net-level state variables, as their interpretation can be always derived from the corresponding formula, and they can not appear in task effects (but task effects can act on the basic predicates and functions that appear in the formula, thus indirectly modifying the truth value for the derived predicate).

Initial Interpretation for a Process Instance. It is given by an assignment of values to the state variables that represent truth values for predicates (initial facts) and initialization values for functions.

Pre-conditions, Post-conditions and Effects. They are defined at design time as logical annotations associated with tasks in a PLANLET. We assume a first-order predicate logic with numeric and object functions, with the restriction that free variables are not allowed and thus all variable symbols must be task parameter names or occur in the scope of a quantifier. The language is clearly inspired from PDDL, although we prefer an infix notation for the operators. Task pre/post-conditions and effects are represented in task specifications via the `<precondition>`, `<effect>` and `<postcondition>` markup elements. In our example, consider the task labeled as `go`, which requires that an actor moves from a location to another in the area. It defines two input parameters $from$ and *to* of type *Loc*, representing the starting and arrival locations, and an input parameter *actor* of type *Actor* representing the emergency operator that executes the task. An instance of this task can be executed only if s/he is currently at the starting location and is connected to the network. As an effect of task execution, the actor moves from the starting to the arrival location, but we need, as post-condition, to verify whether the arrival location has been reached and the actor is still connected to the network. We can thus define the following annotations:

```
<precondition>at(actor) == from AND Connected(actor)</precondition>
<effect>at(actor) = to</effect>
<postcondition>at(actor) == to AND Connected(actor)</postcondition>
```

The designer can distinguish between: *(i) direct effects*, i.e., effects that always take place after an execution, and therefore the corresponding changes on the state variables are automatically performed when the task completes (e.g., if an effect of the form $BatteryLevel(robot)$ += 5 is marked as automatic, after task execution the value for $BatteryLevel(robot)$ is directly increased by 5); and *(ii) supposed* effects, i.e., effects that define changes that are assumed to be performed only when the task is considered as an action in a planning domain.



```
<atEffect>
  <name>at</name>
  <actor>a1</actor>
  <value>loc33</value>
</atEffect>
```

Fig. 5. A task effect represented as a variable assignment

Supposed effects can be interpreted as the effects that a task is supposed to have, but the actual produced changes are defined at run-time as a result of the concrete execution, such as the actual truth value of a predicate or the actual value for a direct assignment. In our example, $at(actor) = to$ is a supposed effect, as the actual value for $at(actor)$ is produced as a task output and may be different from the desired one (i.e., the value of the *to* variable prescribed in the effect). If the designer needs to verify if a task execution has produced the intended effect, s/he has to define a corresponding post-condition (i.e., the $at(actor) == to$).

Direct effects can be directly represented by generating an outbound mapping with an XQuery expression that adds/removes a tuple to/from the state variable representing predicate's interpretation (for positive/negative predicates), or updates the value for a tuple in the state variable representing function's interpretation (for assignment effects). In supposed effects, the actual values are produced by workitem executions, and all predicates and functions that appear in the effect expression have to be represented as task variables, so as to allow to specify (according to task's execution logic) the truth value for predicates or the value for functions. To this end, we represent each predicate and function that appears in the supposed effects as task parameters of type T2, where the predicate/function name is given and fixed, the values for the argument variables (i.e., the grounding) are defined by the inbound mappings for task parameters and the predicate/function actual value will be defined as a result of task execution. For these variables, outbound mappings are then generated, including XQuery expressions to update net-level state variables as for direct effects. Fig. 5 shows an example of how a variable can be used to represent an effect and how the actual value for $at(\mathrm{a1})$ can be produced as output; we show that the output value for $at(\mathrm{a1})$ is produced by a sensor (i.e., a GPS device) supporting the worklist handler. The produced value, in the example 'loc33', is then used to update the net variable representing the *at* interpretation to reflect that $at(\mathrm{a1}) \mapsto \mathrm{loc33}$.

State Model and Exceptions. In a Planlet, a process state S is given by the token marking m_S (as defined in [17] for YAWL nets) and the logical interpretation $\mathcal{I}_S$ that assigns truth values to predicates and values to functions. The initial state over which a process instance is executed is given by the initial marking and an assignment of values to the state variables that represent the initial interpretation for predicates and functions. When a task t becomes enabled in a state S (as determined by m_S), its execution can start only if the task precondition formula φ_{pre} is true in $\mathcal{I}_S$, i.e., $\mathcal{I}_S \models \varphi_{pre}$. A task execution changes the interpretation according to actual task effects (which for a successful execution are given by the corresponding effects expression $expr_{\mathit{eff}}$) and leads to a new state S' where $m_{S'}$ is the produced marking and $\mathcal{I}_{S'}$ is the new interpretation. A completed task is considered as successfully executed if its postcondition formula φ_{post} is true in $\mathcal{I}_{S'}$, i.e., $\mathcal{I}_{S'} \models \varphi_{post}$. At run time all task executions are thus preceded and followed by the verification of whether $\mathcal{I} \models \varphi_{pre}$ and $\mathcal{I} \models \varphi_{post}$[10]. In this model, an exception occurs in a given state with an interpretation $\mathcal{I}$ if a task is enabled but $\mathcal{I} \not\models \varphi_{pre}$ or if a task has completed but $\mathcal{I} \not\models \varphi_{post}$.

From Pre-/Post-conditions to Pre-/Post-execution Constraints. As part of its exception handling mechanism, YAWL supports the definition of workitem-level pre- and post-execution constraints, as rules with conditions that *(i)* are checked when the workitem becomes enabled and when it is completed, and *(ii)* if violated, they trigger an exception and the execution of an exception handling process (i.e., a YAWL exlet [14]). Conditions are defined over case variables as strings of operands and arithmetic, comparison and logical operators; conditional expressions may also take the form of boolean XQuery expressions [17]. In our approach, we leverage on this built-in feature and map the evaluation of pre- and post-conditions to the evaluation of pre and post-execution constraints, by automatically translating φ_{pre} and φ_{post} formulae for each task into YAWL conditional expressions. While arithmetic, comparison and logical operators in our annotation language directly map to the operators supported by YAWL, predicates and functions can be resolved by appropriate XQuery expressions[11].

Representing Planlet Annotations in PDDL. In order to exploit our planning-based recovery mechanism, every task/annotation/property associated to a PLANLET needs to be translated in PDDL. A PDDL definition consists of two parts: the domain and the problem definition. The planning domain is built starting by the definition of basic/derived predicates, object/numeric functions and data types as shown in the previous sections, and by making explicit the *actions* associated to each annotated task stored in the repository linked to the PLANLET under execution, together with the associated pre-conditions, effects and input parameters. Basically, the planning domain describes how predicates and functions may vary after an action execution, and reflects the contextual properties constraining the execution of tasks stored in a specific PLANLET

[10] As φ_{pre} and φ_{post} are *closed* formulae, their truth values can be considered as the answers to the corresponding boolean queries, given the interpretation $\mathcal{I}$.

[11] We recall that no free variables are allowed and all formulae are closed.

repository. Our annotation syntax allows to represent planning domains and problems with the complexity of those describable in PDDL version 2.2[12] [2]. In the following, we discuss how our annotations are translated into a PDDL file representing the planning domain:

- the *name* and the *domain* of a data type corresponds to an *object type* in the planning domain;
- basic and derived predicates have a straightforward representation as *relational predicates* (templates for logical facts) and *derived predicates* (to model the dependency of given facts from other facts);
- numeric functions correspond to PDDL *numeric fluents*, and are used for modeling non-boolean resources (e.g., the battery level of a robot);
- object functions do not have a direct representation in PDDLv2.2, but may be replaced as relational predicates. Since an object function $f : Object^n \rightarrow Object$ map tuples of objects with domain types D^n to objects with co-domain type U, it may be coded in the planning domain as a relational predicate P of type (D^n, U);
- a given YAWL task, together with the associated pre-conditions and effects and input parameters, is translated in a PDDL *action schema*. An action schema describes how the relational predicates and/or numeric fluents may vary after the action execution.

When an exception arises, on a same planning domain a new planning problem is built at run-time, through the description of an initial state (that corresponds to the invalid state of the process s) and the description of the desired goal (a safe state s', derived from the violated pre- or post-condition).

- for each data type defined in the planning domain, all the possible object instances of that particular data type are explicitly instantiated as *constant symbols* in the planning problem (e.g., the fact that $a1$, $a2$, $a3$, $a4$ are *Actors*, $rb1$ and $rb2$ are *Robots*, $loc00$, ..., $loc33$ are *Locations*);
- a representation of the *initial state* of the planning environment is needed. Basically, the initial state of the planning problem corresponds to an invalid state (i.e., a state that needs to be fixed after a pre- or post-condition violation during the process execution). It is composed by a conjunction of relational predicates, derived predicates (e.g., the information about which actors/robots are currently connected to the network) and by the current value of each numeric fluent (e.g., the battery charge level for each robot);
- the *goal state* of the planning problem is a logical expression over facts. In our approach, the goal state is built in order to reflect a safe state to be reached after the execution of a recovery procedure. Suppose that t is the task whose pre-conditions $Pre(t)$ (or post-conditions $Post(t)$) are not verified. The safe

[12] PDDLv2.2 enables the representation of realistic planning domains, with actions and goals involving numerical expressions, operators with universally quantified effects or existentially quantified preconditions, operators with disjunctive or implicative preconditions, derived predicates and plan metrics. However, currently, our formalism does not allow to represent conditional and universally quantified effects.

Table 1. Time performances of LPG-td for adaptation problems of growing complexity

Length of the recovery proc.	Problem instances	Avg. time needed for a sub-optimal sol. (sec)	Avg. length of a sub-optimal sol.	Avg. time needed for a quality sol. (sec)
1	29	6,769	3	7,768
2	36	7,213	3	16,865
3	32	7,846	4	24,123
4	25	8,128	5	37,017
5	21	8,598	8	39,484
6	17	8,736	9	52,421
7	13	9,188	13	73,526
8	12	9,953	14	81,414

state s' corresponding to the goal state is generated starting from the invalid state s, by substituting the wrong facts that led to the exception with the content of the pre-conditions (or post-conditions) violated.

6 Experiments

In order to investigate the feasibility of the Planlet approach, we performed some testing to learn the time amount needed for synthesizing a recovery plan for different adaptation problems. We made our tests by using the LPG-td planner[13] [7]. Such a planner is based on a stochastic local search in the space of particular "action graphs" derived from the planning problem specification. The basic search scheme of LPG-td is inspired to Walksat [16], an efficient procedure for solving SAT-problems. More details on the search algorithm and heuristics devised for this planner can be found at [7,6]. We chose LPG-td as (i) it treats the full range of PDDL2.2 [2] (that is characterized for enabling the representation of realistic planning domains) and (ii) even if it is primarily thought as a satisficing planner, it is able to compute also quality plans under a pre-specified metric. In fact, LPG-td has been developed in two versions: a version tailored to computation speed, named LPG-td.speed, which produces *sub-optimal plans*, and a version tailored for *plan quality*, named LPG-td.quality. LPG-td.speed generates sub-optimal solutions that do not prove any guarantee other than the correctness of the solution. LPG-td.quality differs from LPG-td.speed basically for the fact that it does not stop when the first plan is found but continues until a stopping criterion is met. In our experiments, the optimization criteria was fixed as the minimum number of actions needed for the planner to reach the goal. It is important to underline that satisficing planning is easy (polynomial), while optimal planning is hard (NP-complete) [8]. The experimental setup was performed with the test case shown in our running example. We stored in the Planlet repository 20 different emergency management tasks, annotated with 28 relational predicates, 2 derived predicates and 4 numeric fluents, in order to make the planner search space very challenging. Then, we provided 185 different planning problems of different complexity, by manipulating ad-hoc the

[13] LPG-td was awarded at the 4th International Planning Competition (IPC 2004, `http://ipc.icaps-conference.org/`) as the "top performer in plan quality".

values of the initial state and the goal in order to devise adaptation problems of growing complexity[14]. As shown in Table 1, the column labeled as "Length of the recovery procedure" indicates the smallest number of actions needed for devising a plan of a specific length. Our purpose was to measure (in seconds) the computation time needed for finding a sub-optimal solution and a quality solution for problems that require a recovery procedure of growing complexity. The column labeled as "Average length of a sub-optimal solution" indicates the average number of actions that compose a sub-optimal solution for a problem of a given complexity. A sub-optimal solution is found in less time than a quality one, but generally it includes more tasks than the ones strictly needed. This means that when the complexity of the recovery procedure grows, the quality of a sub-optimal solution decreases. For example, as shown in table 1, on 21 different planning problems requiring a recovery procedure of length 5, the LPG-td planner is able to find, on average, a sub-optimal plan in 8,598 seconds (with 3 more tasks, on average) and a quality plan (which consists exactly of the 5 tasks needed for the recovery) in 39,484 seconds, without the need of any domain expert intervention. Consequently, the approach is feasible for medium-sized dynamic processes used in practice[15].

7 Conclusions

In this paper, we have introduced the concept of PLANLETS, self-contained YAWL specifications featuring automatic adaptation for dynamic processes, based on modeling of pre- and post-conditions of tasks and the use of planning techniques. In contrast to most existing approaches, PLANLET covers on automatic adaptation for processes at runtime that do not need any human interaction. We have shown the feasibility of the approach by discussing its deployment on top of YAWL and by showing some experimental tests based on a real process scenario. Such tests have provided useful insights on the cases in which an automatic approach is convenient wrt. more traditional exception handlers defined at design-time. The assumptions of classical planning (determinism in the action effects, model completeness, etc.) we used for modeling dynamic processes has a twofold consequence. On the one hand, we can exploit the good performance of classical planners (e.g., LPG-td) to solve real-world problems with a realistic complexity; on the other hand, classical planning imposes some restrictions for addressing more expressive problems, including incomplete information, preferences and multiple task effects. Future works will include an extension of our approach dealing with the above aspects, with the purpose to maintain the planning process very responsive.

[14] Some test instances, together with the inputs for the planner, are available at the URL: `http://www.dis.uniroma1.it/ marrella/public/Planlets_CoopIS2012_TestCases.zip`.

[15] We did our tests by using an Intel U7300 CPU 1.30GHz Dual Core, 4GB RAM machine.

Acknowledgements. This work has been partly supported by the Sapienza projects TESTMED and SUPER, by the Italian national PIA Calabria project COSM Factory and by the EU projects Greener Buildings and SmartVortex. The authors thank Arthur H.M. ter Hofstede and Massimiliano de Leoni for useful insights and discussions.

References

1. de Leoni, M., Mecella, M., De Giacomo, G.: Highly Dynamic Adaptation in Process Management Systems Through Execution Monitoring. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 182–197. Springer, Heidelberg (2007)
2. Edelkamp, S., Hoffmann, J.: PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Tech. rep., Albert-Ludwigs-Universitat Freiburg, Institut fur Informatik (2004)
3. Ferreira, H., Ferreira, D.: An integrated life cycle for workflow management based on learning and planning. Int. J. Coop. Inf. Syst. 15, 485–505 (2006)
4. Friedrich, G., Fugini, M., Mussi, E., Pernici, B., Tagni, G.: Exception handling for repair in service-based processes. IEEE Trans. on Soft. Eng. 36, 198–215 (2010)
5. Gajewski, M., Meyer, H., Momotko, M., Schuschel, H., Weske, M.: Dynamic failure recovery of generated workflows. In: DEXA 2005 (2005)
6. Gerevini, A., Saetti, A., Serina, I.: Planning through stochastic local search and temporal action graphs in Lpg. J. Art. Int. Res. 20(1), 239–290 (2003)
7. Gerevini, A., Saetti, A., Serina, I., Toninelli, P.: Lpg-td: a fully automated planner for PDDL2.2 domains. In: ICAPS 2004 (2004)
8. Helmert, M.: Complexity results for standard benchmark domains in planning. Art. Int. 143, 219–262 (2003)
9. Jarvis, P., Moore, J., Stader, J., Macintosh, A., du Mont, A.C., Chung, P.: Exploiting AI technologies to realise adaptive workflow systems. In: AAAI Workshop on Agent-Based Systems in the Business Context (1999)
10. Lenz, R., Reichert, M.: IT support for healthcare processes. Premises, challenges, perspectives. Data Knowl. Eng. 61, 39–58 (2007)
11. Marrella, A., Mecella, M., Russo, A.: Featuring automatic adaptivity through workflow enactment and planning. In: CollaborateCom 2011 (2011)
12. Marrella, A., Mecella, M., Russo, A., ter Hofstede, A.H.M., Sardiña, S.: Making YAWL and SmartPM interoperate: Managing highly dynamic processes by exploiting automatic adaptation features. In: BPM, Demos (2011)
13. R-Moreno, M.D., Borrajo, D., Cesta, A., Oddi, A.: Integrating planning and scheduling in workflow domains. Exp. Syst. with Applications 33(2) (2007)
14. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow Exception Patterns. In: Martinez, F.H., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006)
15. Schonenberg, H., Mans, R., Russell, N., Mulyar, N., van der Aalst, W.M.P.: Process flexibility: A survey of contemporary approaches. In: CIAO! / EOMAS 2008 (2008)
16. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: AAAI 1994 (1994)
17. ter Hofstede, A.H.M., van der Aalst, W.M.P., Adams, M., Russell, N.: Modern business process automation: YAWL and its support environment. Springer (2009)

18. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66, 438–466 (2008)
19. Weber, B., Wild, W., Lauer, M., Reichert, M.: Improving exception handling by discovering change dependencies in adaptive process management systems. In: BPI 2006 (2006)
20. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: HICSS 2001 (2001)