Featuring Automatic Adaptivity through Workflow Enactment and Planning

Andrea Marrella, Massimo Mecella, Alessandro Russo
Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti
SAPIENZA - Università di Roma
Email: {marrella,mecella,arusso}@dis.uniroma1.it

***Abstract*—Process Management Systems (PMSs, a.k.a. Workflow Management Systems - WfMSs) are currently more and more used as a supporting tool to coordinate the enactment of processes. In 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 adapt the process without completely replacing it. In this paper, we propose a general approach and a conceptual architecture to automatic adaptation, based on the concept of declarative modeling of processes and the use of continuous planning techniques. We show the feasibility of the proposed approach by discussing its deployment on top of YAWL, one of the most famous research prototypes of PMSs, in order to demonstrate the modularity and nice integrability with existing models and techniques. A running example shows the practical applicability of the approach.**

***Index Terms*—Process Management Systems, Process Adaptivity, Continuous Planning, YAWL**

I. Introduction

Process Management Systems (PMSs, a.k.a. Workflow Management Systems) are applied to support and automate process enactment, aiming at increasing the efficiency and effectiveness in its execution. Classical PMSs offer good process support as long as the processes are structured and do not require much flexibility. In the last years, the trade-off between *flexibility* and *support* has become an important issue in workflow technology. On the one hand, there is a desire to control processes and to avoid incorrect or undesirable executions of these processes. On the other hand, users want flexible processes that do not constrain them in their actions [18]. To deal with evolving processes, exceptions and uncertainly, the need for flexible and easy adaptable PMSs has been recognized as one of the critical success factors for any PMS [9], [18]. In this range, automatic adaptivity can be seen as the ability to efficiently deal with process changes and exceptions that may occur during runtime by dynamically adapting the process instance under execution. To "adapt" a process means to deviate from the process schema at instance level through high-level change operations (e.g., add a new process fragment in parallel to the existing control flow), without altering the goals prescribed at design time. A detailed set of adaptation patterns that constitute solutions for realizing change in PMSs is proposed in [19].

Nowadays, in classical PMSs processes are mainly defined through imperative approaches. However, to be only "process aware" makes it difficult to deal with process change [12]. Most work [7], [17] about resolving exceptions refers to failing processes or concerns design-time solutions, used to specify the needed compensation actions in case of exception. The design-time specification of all possible compensation actions require an extensive manual effort for the process designer, that has to anticipate all potential problems and ways to overcome them in advance. This is particularly true in real world scenarios, where the process designer often lacks the needed knowledge to model all the possible contingencies at design-time, or this knowledge can become obsolete during the process progressing, by making useless his/her initial effort.

This paper discusses how a modeling approach towards a declarative specification of process tasks, i.e., comprising the specification of input/output artefacts and task preconditions and effects, allows to layer planning techniques on top of traditional PMSs, in order to enable automatic adaptivity - without defining explicitly any recovery policy - and to balance between flexibility and support. Our approach relies on a domain-independent planner where the process designer just states *what* properties have to be satisfied without having to anticipate *how* these can be fulfilled. We present the overall approach and the underlying conceptual architecture, and we demonstrate its general validity by showing how to integrate it with YAWL [17], that is among the most well-known PMSs coming from academia.

The rest of the paper is organized as follows. Section II covers the state of the art in adaptivity in PMSs and relevant results in planning. Section III presents the general approach and the conceptual architecture, by discussing the notions of preconditions and effects and the use of planning techniques. Section IV discusses how concretely the approach can be built on top of YAWL, thus providing a real instantiation of the conceptual architecture. An example, presented in Section V, clarifies all the peculiarities of the approach. Finally Section VI concludes the paper by discussing limitations and future developments of the approach.

II. Related Works

There are two ways to handling exceptions: *manual* and *automatic*. In the first case, once exceptions occur, a responsible person, expert on the process domain, modifies manually the affected instances. There exist many works that deal with manual adaptivity [1], [6], [7], [13], [17], [20], [21].

COLLABORATECOM 2011, October 15-18, Orlando, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2011.247096

Among them, interesting approaches are ProCycle [20] and ADEPT2 [6]. The first uses a case-based reasoning approach to support adaptivity of workflow specifications to changing circumstances. Case-based reasoning (CBR) is the way of solving new problems based on the solutions of similar past problems: users are supported to adapt processes by taking into account how previously similar events have been managed. However, adaptivity remains manual, since users need to decide how to manage the events though they are provided with suggestions. ADEPT2 features a check of "semantic" correctness to evaluate whether events can prevent processes from completing successfully. But the semantic correctness relies on some semantic constraints that are defined manually by designers at design-time and are not inferred, e.g., over pre- and post-conditions of tasks.

On the contrary, a PMS that supports *automatic* adaptivity is able to automatically change the schema of affected instances in a way they can still be completed, according to the exceptions that have been raised. 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 [7], [17], with the challenge that in many cases such a compensation cannot be performed by simply undoing actions and doing them again. Such an exception handler can be compared to the `try-catch` approach used in some programming languages such as `Java`; the `catch` is the definition of the possible exception and the specification, defined at *design-time* by the process engineer, of how to deal with it. The novelty proposed in this work is the automatic construction of exception handlers at run time, i.e., the *run-time automatic synthesis* of the `catch` block.

A. Planning Algorithms

Planning systems are problem-solving algorithms that operate on explicit representations of states and actions. The standard representation language of classical planners is known as the Planning Domain Definition Language [11] (PDDL); it allows to formulate a problem through the description of the *initial state* of the world, the description of the desired *goal* and a set of possible actions. An *action definition* defines the conditions under which an action can be executed, called *pre-conditions*, and its *effects* on the state of the world, called also post-conditions. The set of all action definitions represents the *domain* of the planning problem. A planner that works on such inputs generates a sequence of actions (the *plan*) that leads from the initial state to a state meeting the goal. Section III clarifies (with an example) how to define a planning problem through PDDL.

In the literature, there exists a wide range of different planning techniques, that are characterized by the specific assumptions provided to find a plan fulfilling the goal. In order to demonstrate our approach, we focus on the Partial Order Planning (POP) algorithm provided by UCPOP [15], that guarantees to return only consistent plans, i.e., plans in which there are no cycles and no conflicts in the ordering constraints. Basically, POP algorithms take as input a planning problem defined through PDDL and explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, by adding actions to the plan to achieve each subgoal. A POP planner produces a set of partial ordering constraints of the form $A \prec B$, which is read as "A before B" and means that action A must be executed sometime before action B, but not necessarily immediately before. Moreover, it returns a set of *causal links* of the form $C \xrightarrow{p} D$, which is read as "C achieves p for D" and means that p is an effect of action C and a precondition for action D. It also asserts that p must remain $true$ from the time of action C to the time of action D. In other words, the plan may not be extended by adding a new action that conflicts with the causal link. The main advantage of the least-commitment philosophy is based on the fact that decisions about actions ordering is postponed until a decision is forced, guaranteing flexibility in the execution of the plan itself. Section V shows the working of the UCPOP algorithm on a practical example.

In well-known environments planning can be done *offline*, and solutions can be found and evaluated prior to execution. On the contrary, in a dynamic environment, since new world information is continuously sensed, a planner should adapt *online* to it by the refinement of the plan that is under construction. *Continuous Planning* [14] refers to the process of planning in a world under continual change, where the planning problem is often a matter of adapting to the world when new information is sensed. A continuous planner is designed to persist indefinitely in the environment. Thus it is not a "problem solver" that is given a single goal and then plans and acts until the goal is achieved; rather, it lives through a series of ever-changing goal formulation, planning, and acting phases.

B. Techniques for Workflow Enactment and Planning

The idea of using AI techniques to handle adaptivity is not strictly new [2]–[5], [8]. In [5] it is presented a concept for dynamic and automated workflow re-planning that allows to recover from task failures. To handle the situation of a partially executed workflow, the paper proposes a multi-step procedure that includes the termination of failed activities, the sound suspension of the workflow, the generation of a new process definition and the adequate process resumption. In [8] 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 capabilities offered by AI techniques. In particular, the work describes how planning can be interleaved with process execution and plan refinement. It also investigates plan patching and plan repair as a mean to enhance flexibility and responsiveness. The work [3] proposes a new life cycle for workflow management based on the continuous interplay between learning and planning. The approach is based on learning business activities as planning operators and feeding them to a planner that generates the process model. A fundamental result obtained is that it is possible to produce fully accurate process models even though the

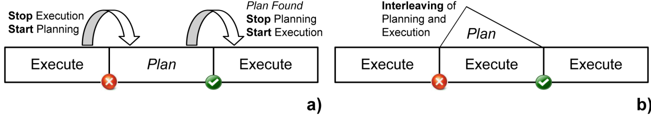

Fig. 1. Traditional "plan then execute" cycle (a) and the *continuous planning* (b) approach

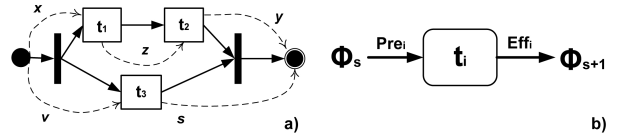

Fig. 2. An example of process definition (a) and the task anatomy (b) in our approach

activities (i.e., the operators) may not be accurately described. The approach depicted in [2] 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 the 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. The work [4] proposes a self-healing approach to handle exceptions in service-based processes and to repair the faulty activities with a model-based approach. During execution, when an exception arises, alternative repair plans are generated by taking into account constraints posed by the process structure, dependencies among data, and available repair actions defined in the process model.

The above approaches for interleaving workflow execution and planning manage the two phases as completely independent one from another. When an exception is identified, the process execution is stopped and the planner is invoked with a new set of goals and the process state as the initial state. When the planner finds a process fragment that compensate the exception, the main process is resumed by including the compensate actions just computed (cf. Fig. 1.a). This means that planning is considered an offline process which requires considerable computational effort and there is a significant delay from the time the planner is invoked to the time that the planner produces a new plan. Moreover, if a negative event occurs (e.g., a plan failure), the response time until a new plan may be significant. To achieve a higher level of responsiveness in real-life environments, rather than considering the planner as an offline process, we make use of a *continuous planning* approach, that allows an "online interleaving" of workflow enactment and planning (cf. Fig. 1.b). When an exception arises, the planner computes the recovery compensation actions in *concurrency* with the execution of the remaining part of the main process which is not affected by the exception. In contrast with the above works, our 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 is a non-blocking technique; it does not stop directly any task in the main process during the computation of the recovery process.

III. The Proposed Approach

Process adaptivity can be seen as the ability of the PMS to reduce the gap from the *expected reality* - the (idealized) model of reality that is used by the PMS to reason - and the *physical reality* - the real world with the actual values of conditions and outcomes. Our approach is based on an interleaving between workflow execution, monitoring the two realities and planning, and allows to recover from exceptions without defining explicitly any recovery policy.

Let us now introduce a conceptual model to formalize processes. A process model is defined as a set of n task definitions, where each task t_i can be considered as a single step that consumes input data and produces output data. Data are represented through some process variables whose definition depends strictly on the specific process domain of interest. The model allows to define logical constraints based on process variables through a set F of predicates f_j. Such predicates can be used to constrain the task assignment (in terms of *task preconditions*), to assess the outcome of a task (in terms of *task effects*) and as guards into the expressions at decision points (e.g., for cycles or conditional statements). Since tasks are described in terms of preconditions and effects, it is convenient to define the behavior of each task directly in terms of its PDDL specification, as follows:

```
(define (domain example)
    (:action t1
    :precondition (x)
    :effect (and (not(x) (z) (k))))
    (:action t2
    :precondition (z)
    :effect (y))
    (:action t3
    :precondition (v)
    :effect (s))
    (:action t4
    :precondition (k)
    :effect (x)))
```

The meaning is straightforward. For example, the first action definition states that for executing t_1, the predicate x must hold. Then, it states that a successful execution of t_1 guarantees that predicates $\neg x$, z and k will hold together. In Fig. 2.a a simple process P_0 is depicted, whose task ordering is imposed by a control flow defined through an UML activity diagram. The control flow is composed by a subset of the tasks provided in the PDDL specification (e.g., task t_4 is currently not needed for the process instance built at design time). Tasks in the control flow are partially ordered in a way that the effects of preceding activities satisfy the preconditions of subsequent tasks. Dashed arrows in Fig. 2.a represent causal links (whose meaning is described in Section II) that imply an ordering constraint between two tasks. For example, the ordering constraint from t_1 and t_2 is derived from the fact that t_1 has the effect z that is needed by t_2 as a precondition. Our dynamic world is modeled as progressing through a

series of *states*. Each state is the result of various tasks being performed so far. Predicates may be thought of as "properties" of the world whose values may vary across states. Let us now formalize some preliminary concepts.

Definition 1: A *physical reality* Φ_s is represented by the set of predicates $F_{\Phi_s} \subseteq F$ that hold in the state s.

The physical reality Φ_s reflects the concept of "now", i.e., what is happening in the real environment whilst the process is under execution. In general, a task can only be performed in a given physical reality Φ_s if and only if that reality satisfies the *preconditions* Pre_i of that task. Moreover, each task has also a set of *effects* Eff_i that change the current physical reality Φ_s into a new physical reality Φ_{s+1}. Note that, as enforced in Fig. 2.b, the approach treats each task as a "black box" that consumes input data and produces output data, and no assumption is made about its internal behavior. A PMS that takes in input such a process specification should guarantee that each task is executed correctly, i.e., with an output that satisfies the process specification itself. In fact, at execution time, the process can be easily invalidated because of task failures or since the environment may change due to some external event. For this purpose, the concept of *expected reality* Ψ_s is given :

Definition 2: An expected reality Ψ_s is represented by the set of predicates $F_{\Psi_s} \subseteq F$ that should hold in the state s.

A recovery procedure is needed if the two realities are different from each other. Let us assume that the task t_1 of Fig. 2.a ends its execution without having all anticipated effects. For example, it has the effect k and $\neg x$ but not the effect z. It results in a physical reality Φ_s different from the expected reality Ψ_s, in which, instead, predicates p, $\neg x$ and z hold together. Formally, a state s is known as $Relevant$ - candidate for adaptation - iff :

$$Relevant(s) \equiv \neg sameState(\Phi_s, \Psi_s) \quad (1)$$

Predicate $sameState(\Phi_s, \Psi_s)$ holds iff the states denoted by Φ_s and Ψ_s are the same. If a discrepancy between the two realities is verified, the PMS needs to derive a flow of repairing actions that turns the physical reality into the expected reality and modifies the process specification to ensure that, at the end, the above gap is removed. Our approach allows a PMS to recover from exceptions without defining explicitly any recovery policy. In fact, the only needed trigger for adaptation in a state s is when the predicate $Relevant(s)$ holds, meaning that something has gone wrong during the process execution. The PMS never takes care of any exception; instead, it sends the information about the two realities as well as the task definitions to an external planner and carries on with the execution of the process. In this sense we can advocate that our repairing technique is non-blocking. The idea is that the planner builds the recovery process P_h in parallel with the execution of the main process P_0, avoiding to stop directly any task in the main process. The planner takes in input a concrete planning problem, whose domain is constituted by the set of task definitions, whereas the initial state and the goal correspond with physical reality Φ_s and expected reality Ψ_s that have to be aligned. The part of P_0 that is not affected by the exception can proceed with its execution. At any time, during the plan computation, an incremental update to the physical reality (that reflects the main process progressing) may update the current state of the plan. Such update is the result of an external event (that modifies Φ_s in an asynchronous way) or simply of a task just terminated. The planner is then responsible for maintaining a consistent plan with the most current information. From the point of view of the planner, after each task termination (or external event happening), that results in turning Φ_s and Ψ_s into Φ_{s+1} and Ψ_{s+1}, the following occurs:

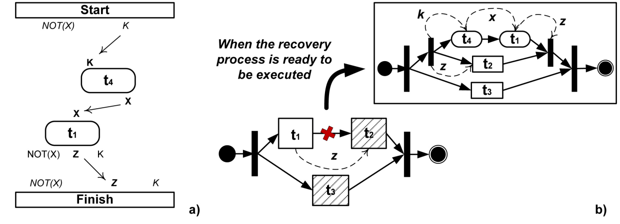

Fig. 3. An example of recovery process

- the execution of the planner is stopped (together with the "old plan" built so far);
- changes to the goals and the initial state first posted to the plan: now, a new planning problem is built, with Φ_{s+1} as initial state, Ψ_{s+1} as the goal;
- effects of these changes are propagated through the old plan (that include conflict identification);
- plan repair algorithms are invoked to remove conflicts and make the old plan appropriate for the current state and goals;
- the planner can resume its execution, starting from that fragment of the old plan that is consistent (i.e., has no conflicts) with the new realities.

Now, let's suppose that the current process is $P_0 = (P_1; P_2)$, in which P_1 is the part of the process already executed and P_2 is the part of the process which remains to be executed. Once synthesized, P_h will be inserted in parallel with P_2 and will be executed in concurrency with every other task yet to be executed. This means that the process yet to be executed by the PMS is $P_2 || P_h$.

Let us consider again the above example. Task t_1 has ended its execution having as effects k and $\neg x$, and it turned Φ_s into $\Phi_{s+1} = \{k, \neg x\}$. On the contrary, the expected reality in state $s+1$ is equal to $\Psi_{s+1} = \{k, \neg x, z\}$, as if the task t_1 has been executed correctly, having all its anticipated effects. Note also that in state $s+1$ the task t_2 cannot proceed because its preconditions are not satisfied (i.e., predicate z in Φ_{s+1} does not hold). The PDDL planning problem of the example cited above is:

```
(define (problem A)
    (:domain example)
    (:init and((k) (not x)))
    (:goal and((k) (not x) (z))))
```

It can be specified as follows: the initial state is $k \wedge \neg x$, the goal is $k \wedge \neg x \wedge z$ and the actions needed for building the plan are defined in the planning domain provided above. Fig. 3.a shows a plan found through the partial-order planning (POP) algorithm of UCPOP (the rounded-edge rectangles represent the tasks introduced for repairing). The plan found constitutes exactly the recovery process P_h that has to be run in concurrency with P_2 (cf. Fig. 3.b). The execution of P_h ends by having as effect z and by "unlocking" the task t_2, previously stopped because its preconditions were not satisfied.

We focus now on some termination issues about the recovery procedure proposed in the approach.

Definition 3: Given a task t_i whose effects are denoted by Eff_i, we state that t_i affects a predicate f_j iff $f_j \in Eff_i$. We denote it with $t_i \triangleright f_j$.

Every task defined in the process specification affects a finite number of predicates. Let us now formalize the concept of $strong\ consistency$ for a process P_0.

Definition 4: Let P_0 a process composed by n tasks $t_1, .., t_n$. P_0 is strongly consistent iff $\forall j, \nexists (t_i, t_k)_{i \neq k}\ s.t. (t_i \triangleright f_j \wedge t_k \triangleright f_j)$.

Intuitively, a process P_0 is strongly consistent if do not exist two different tasks in the process that affect the same predicate.

Theorem 1: Let P_0 be a strongly consistent process composed by a finite number of tasks $t_1, ..., t_n$. If P_0 does not contain `while` constructs, and the number of external exceptions is finite, then the planner terminates.

The termination could not be guaranteed if P_0 contains loops, since potentially the two realities could indefinitely change. The same is true if the number of external exceptions is unbounded. More technical details about the working of the algorithm can be also found in [10].

Compared to the other works that involve workflow enactment and planning [2]–[5], [8], our approach allows the planner to continuously sense the environment during the process execution. Changes in realities during enactment are reflected directly in the partial plan under construction. In this last case, the planner first updates the initial state and the goal with the new values of the two realities, then revises the partial recovery plan (built until that moment) to the new realities by deleting possible conflicts, and finally restore the planning procedure. In this way, it is guaranteed that the plan under construction is always synchronized with the realities of the process that is carry on with its execution. We want to underline that we chose to use a simple PDDL specification (with deterministic effects) and UCPOP to make the approach more understandable to the reader. Anyway, the whole approach works also with more expressive PDDL variants, that capture non-deterministic effects, and that the use of UCPOP is not mandatory. The strength of the approach is in the continuous planning algorithm, that is able to work on top of a whatever planning technique.

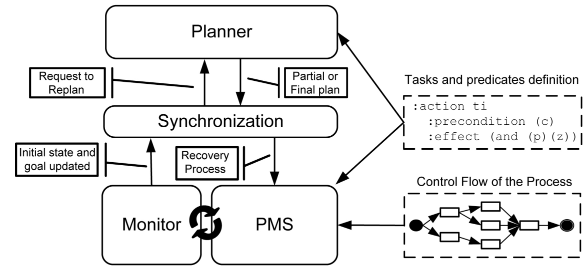

Fig. 4. Conceptual architecture of the approach

A. Conceptual Architecture

Our approach to integration of workflow execution and planning relies on four main components shown in Fig. 4. The **PMS** takes in input a process representation - described as a list of predicates and tasks with preconditions and effects, as well as a control flow that drives tasks within the process - and coordinates the enactment of tasks along a specific control flow. The PMS provides a proper execution engine that manages the process routing and decides which tasks are enabled for execution, by taking into account the control flow, the value of predicates and preconditions and effects of each task. Before a process starts its execution, the PMS builds its physical reality Φ_s by taking the initial context from the environment. Once a task is ready for being assigned, the PMS engine is also in charge of assigning it to a proper service (which may be a human actor, a robot, a software application, etc.).

The **Monitor** component interacts continually with the PMS and it is in charge to decide whether adaptivity is needed. At each execution step - i.e., when the ending of a task or an exception has turned Φ_s into Φ_{s+1} - the monitor checks if the new state $s+1$ can be classified as relevant (cf. Equation 1). If this is the case, the monitor collects the physical reality Φ_{s+1}, the expected reality Ψ_{s+1} and sends them to the synchronization component.

The **Synchronization** component enforces synchronization between the PMS, the monitor and the planner. Every time it receives from the monitor the two realities, it builds a corresponding planning problem in PDDL, by converting the physical reality into the initial state and the expected reality into the goal. Basically, synchronization component can be seen as a conflict-removal procedure that revises the partial recovery plan to the new realities.

The **Planner** is invoked when the synchronization component builds a new planning problem. In addition to the initial state and the goal, it can accept as input also a partial plan, that represents a fragment of the solution. The domain of the planning problem (that is, the PDDL specification with tasks and predicates) is loaded exactly when the PMS starts the execution of the specific process. When a plan satisfying the goal is found, it is sent back to the synchronization component that, in turn, converts it in a readable format for the PMS. The recovery process just obtained is now ready to be executed by the PMS.

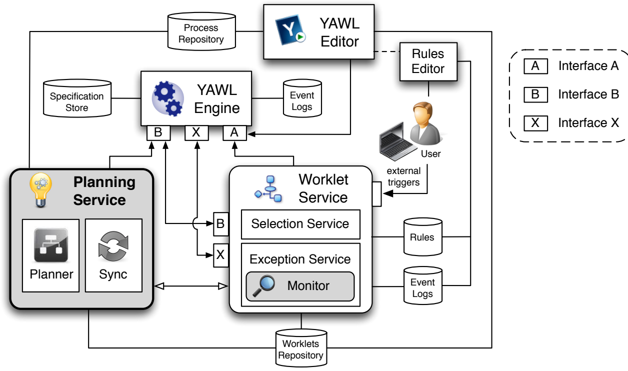

Fig. 5. The YAWL architecture extended with the *Planning Service*

IV. Deploying the Approach on Top of YAWL

In the following we briefly focus on the exception handling approach implemented in the YAWL system, as presented in [17], and then we discuss how our approach can be integrated in the YAWL architecture.

A. Exception Handling in YAWL

The exception handling capabilities provided by YAWL[1] were designed and implemented starting from the conceptual framework for workflow exception handling presented in [16]. In order to understand how exceptions are detected and handled in YAWL we refer to the architecture in Fig. 5 (for now, do not consider the *Planning Service*, which is introduced later in the paper[2]).

For each exception that can be anticipated, it is possible to define an exception handling process, named *exlet*, which includes a number of exception handling primitives (for removing, suspending, continuing, completing, failing and restarting a work item/case) and one or more compensatory processes in the form of *worklets* (i.e., self-contained YAWL specifications executed as a replacement for a workitem or as compensatory processes [17]). Exlets are linked to specifications by defining specific *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 runtime, exceptions are detected and managed by the *Exception Service*, a sub-service of the *Worklet Service* [17]. The *Exception Service* is notified by the engine via *Interface X* of exception triggering events (which include timeouts, resource unavailabilities and notifications fired when a case/workitem begins/ends in order to check pre/post-execution constraints) that may result in an exception[3]. For each event notification, the service determines whether an exception has occurred and, if so, it executes the corresponding exlet. Exception handling primitives are directly executed invoking the corresponding methods (for removing, suspending, etc. a workitem/case) provided by the engine-side of *Interface X*. If the exlet includes a compensation worklet, the *Exception Service* first retrieves it from the repository, then loads it into the engine via *Interface A* and finally starts it via *Interface B*. The worklet is then executed by the engine as a new separate case, possibly in parallel with the parent case if it was not suspended by the exlet.

B. Enabling Planning-based Exception Handling in YAWL

The exception handling approach we propose is not meant to replace existing consolidated approaches, but it rather aims to complement them. Therefore, the architectural extension and integration we designed takes advantage of YAWL's exception detection capabilities and leverages the flexibility of the exlet/worklet-based handling techniques. Specifically, we identified four main steps towards the integration of a planning-based exception handling approach in YAWL: *(i)* extend process and task specifications in order to enable the definition of preconditions and effects; *(ii)* integrate a *Planning Service* into the YAWL architecture; *(iii)* allow process designers to specify at design time whether an occurring exception should be handled by the *Planning Service*; *(iv)* extend the capabilities of the YAWL *Exception Service* in order to enable at runtime exception handling through planning techniques. As a planning-based exception handling approach can be considered as data-driven (i.e., case data affect the evaluation of preconditions, effects and constraints, as well as the definition of planning goals), we focus on workitem/case pre/post-execution constraints' violations.

The specification formalism for preconditions and effects depends on the planning language and tool being used. Here, we assume that tasks and processes are annotated with the PDDL notation and specified over task/process variables that at runtime define the execution status.

From an architectural perspective, as shown in Fig. 5, planning capabilities are provided by a *Planning Service* that implements the continuous 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 lifecycle, from process design, enactment and monitoring to exception detection, handling and (possibly) resolution.

Exception definition and detection.: At design time, the process designer identifies possible exceptions that may occur and defines through the *Rules Editor* the triggering conditions and the corresponding exlets. In order to allow the process designer 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 the generation of the compensation worklet is delegated to the *Planning Service*. Fig. 6 shows an excerpt of the rule file produced by the *Rules Editor* and defined for detecting and handling a workitem-level pre-execution constraint violation.

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

[2] With the exclusion of the *Planning Service* and the *Monitor* component, the picture refers to the architecture defined in [17].

[3] Externally triggered exceptions are instead notified by the environment, specifically by a client involved in process execution.

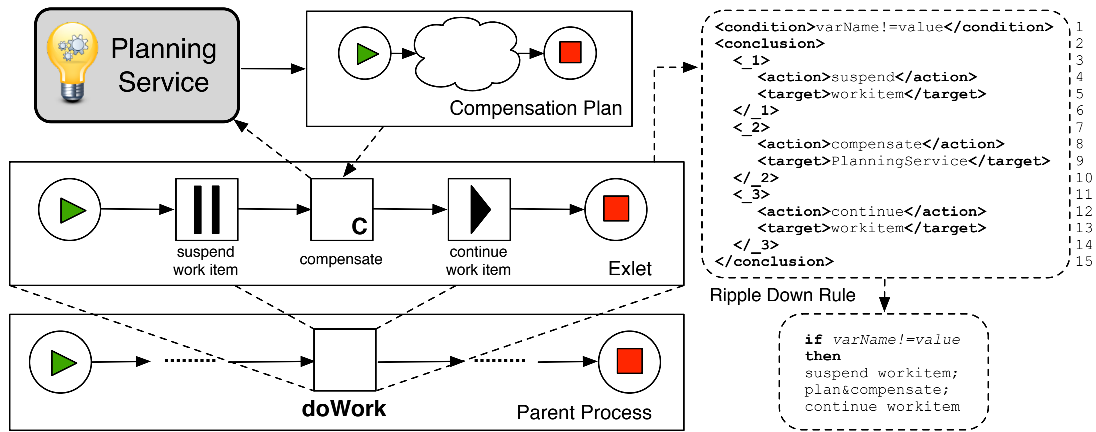

Fig. 6. *Planning Service* activation hierarchy for exception handling

Line 1 defines the exception triggering condition, while lines 2-15 define the exception handling exlet (which consists in suspending the current work item, performing some compensation activities and then resuming the suspended work item). In the current YAWL implementation, the `<target>` element corresponding to the `compensate` action (lines 8-9) contains the name of the worklet to be executed as compensation process. 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). At runtime, as explained in Section IV-A, exception triggering events and case data are matched by the *Exception Service* against the defined rules to detect an occurring exception and activate the corresponding exlet. However, the *Exception Service* has been extended in order to recognize a compensation activity mapped to the *Planning Service* and enact planning capabilities provided by the service. Consider in Fig. 6 the execution hierarchy that leads to the activation of the *Planning Service* when a workitem-level pre-constraint violation occurs for the *doWork* task. After suspending the workitem, the *Exception Service* recognizes the mapping performed at design time and delegates the generation of the compensation activities to the *Planning Service*. Exception triggering events for workitem/case enablement or completion (along with the related I/O data values) for which no explicit rules were defined are managed by the *Monitor* component we introduced in the *Exception Service*. The *Monitor* is able to identify violations over preconditions or effects by comparing current execution state (as given by workitem/case variables) with expected preconditions or effects defined in task/process specifications. If a violation is detected, the occurred exception is managed by executing predefined handling exlets, depending on the type of exception. For a workitem/case precondition violation, the recovery activities consist in suspending the workitem, activating the *Planning Service* and then resuming the suspended workitem/case. For a violation caused by a workitem effect, the default exlet consists in suspending the running case, activating the *Planning Service* and then resuming the suspended case, whereas an effect mismatch detected for a completed case is handled by directly activating planning procedures.

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 constraints, preconditions or effects. Based on this information, the *Synchronization* component of the *Planning Service* is able to define a planning problem (as explained in Section III-A) and submit it to the *Planner* module in charge of synthesizing a recovery plan. In order to produce a compensation plan and determine the set of executable tasks, the *Planner* has to access the specifications of available processes and tasks, stored in the *Process and Worklets Repositories* and in the *Specification Store* (containing specifications currently loaded into the engine and accessible via *Interface B*). If the running case was not suspended, while building the plan the *Planning Service* needs to monitor running workitems that may produce data values and effects which require to update planning goals and/or the plan built up to that moment. In order to enable a *continuous* planning approach, the *Synchronization* component listens for data change notifications produced by the *Exception Service* when it receives a notification from the engine for a completed workitem. By handling these events, the *Synchronization* component is able to identify status changes and, if required, to update the planning goal and/or the plan being synthesized. 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 in parallel with the parent 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. 6, as the compensation worklet synthesized by the *Planner* is supposed to recover from the constraint violation, the suspended work item can then be resumed and executed.

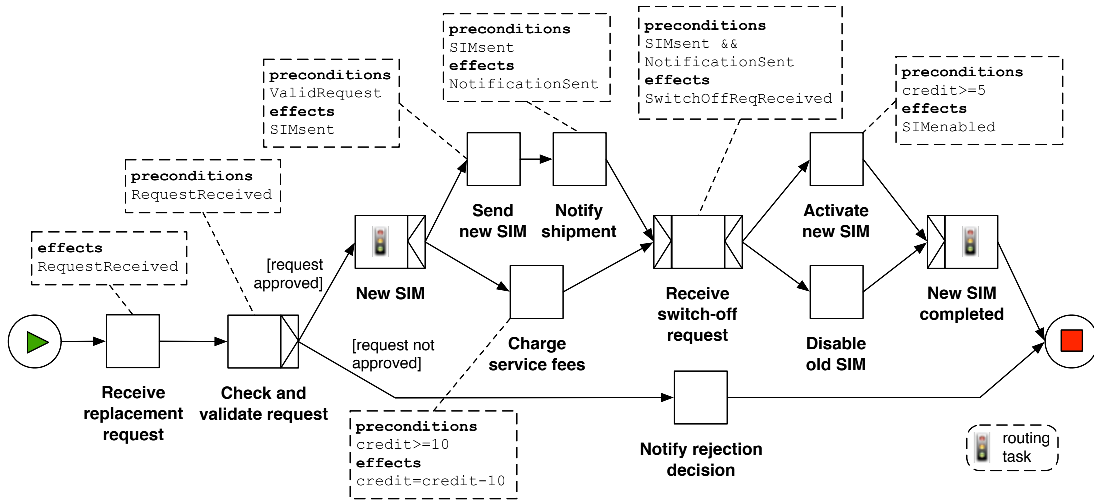

Fig. 7. The SIM card replacement process in YAWL

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 worklet or just canceling the process case.

V. A Practical Example

Mobile telecommunication companies acting as SIM card providers often need to deal with customers asking for a SIM card replacement. For instance, a customer may wish to replace his/her old card with a micro-SIM or with a new card with extra memory capacity. In the following example we consider the SIM card replacement process defined by a mobile telco to deal with replacement requests.

The high-level process model represented as a YAWL net is shown in Fig. 7. The SIM card replacement process starts when a customer submits a SIM replacement request. Upon receiving an application (*Receive replacement request* task), the request is handled by a company employee who gathers customer data and then performs specific checks to validate the request and verify customer payment and billing history according to the contract (*Check and validate request* task). If for some reason the request is rejected, the telco company notifies the customer of the rejection decision (*Notify rejection decision* task) and the replacement process ends. Otherwise, if the application is successfully validated and accepted, the company issues a new SIM card and sends it to the customer (*Send new SIM* task) who is then notified of the shipment (*Notify SIM shipment* task). In parallel, the customer is charged 10€ as shipping and handling fees, withdrawn from his/her prepaid account (*Charge service fees* task). The telco company then waits for receiving from the customer a switch-off request for the old SIM and, upon receiving the card switch-off request (*Receive switch-off request*), the company disables the old SIM (*Disable old SIM* task) and in parallel activates the new card (*Activate new SIM* task). According to an internal business policy, a new SIM can be activated only if the customer has at least 5€ credit left.

Task definitions in the specification in Fig. 7 include simple annotations defining basic *preconditions* and *effects*. The requirement for the activation of a new SIM can be captured by defining a *precondition* over the *Activate new SIM* task (`credit>=5`) or by the process designer, who may recognize that an exception can occur and explicitly define a triggering rule in the form of workitem pre-execution constraint. Suppose now that a customer with 10€ credit left asks for replacing his/her card. In the corresponding case, when s/he is charged for the service (*Charge service fees* task), there is no credit left and this results in an exception triggered when the workitem for the *Activate new SIM* task is enabled. The exception, detected and handled in YAWL by the *Exception Service*, leads to the activation of the corresponding *exlet*, explicitly defined by the designer or activated as a result of a *precondition* violation. Assuming the execution of an *exlet* as defined in Fig. 6, the affected workitem is suspended and the *Planning Service* is activated.

The planning procedure is activated with a planning domain constituted by the set of task definitions taken by the process specification (with the list of preconditions and effects for each task) and with a planning problem, which includes a goal (i.e., an expected reality) defined as `credit>=5`, and an initial state (i.e., a physical reality) with `credit=0`. The following hold in both initial state and goal: `NotificationSent`, `SIMsent`, `SwitchOffReqReceived`, `ValidRequest` and `RequestReceived`. The planner should guarantee that the recovery procedure, after being executed, does not change any other predicate than `credit`. Fig. 8.a depicts the plan found by a generic partial-order planner like UCPOP [15]. Let us now detail how the plan has been effectively built. *Start* and *Finish* are "dummy" tasks which mark the beginning and end of the plan, with the ordering constraint $Start \prec Finish$. Start has no preconditions and its effect is composed by all the predicates that hold in the initial state of the planning problem. Finish has no effects and has as its preconditions the goal predicates of the planning problem. Before the beginning

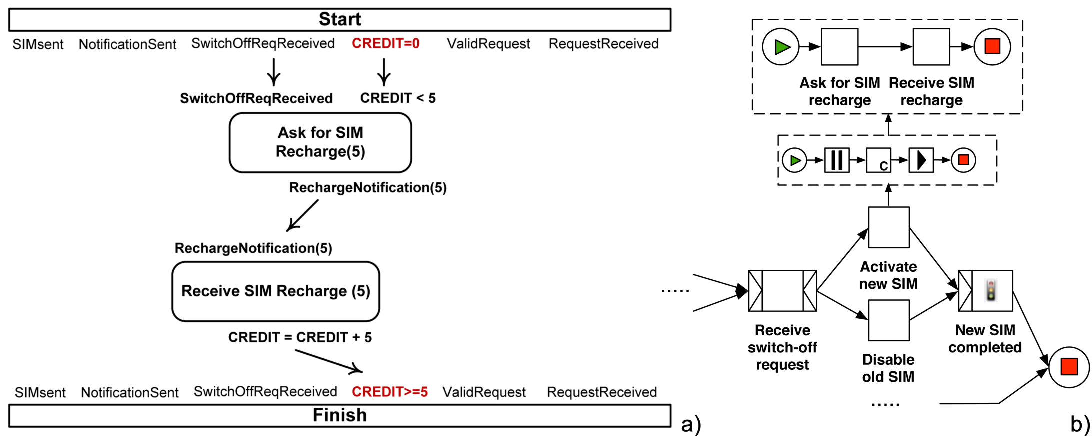

Fig. 8. The recovery plan for the SIM card replacement process in YAWL

of the planning procedure, all the preconditions in Finish can be seen as *open preconditions*, i.e., as preconditions that are not yet achieved by any task in the plan. The planner will work to reduce the set of open preconditions to the empty set, without introducing any conflict. It starts by analyzing all the open preconditions in Finish; each predicate (except than `credit>=5`) has already been satisfied by the effects of the action Start. Since `credit>=5` is currently the only open precondition, the planner checks if within the list of task definitions there exists a task that satisfies such a predicate, i.e., if one of its effects achieves `credit>=5`. As shown in Fig. 8.a, the planner chooses the *Receive SIM Recharge(x)* task (with x=5), whose effect is to increase the credit of 5€, and whose precondition corresponds to the holding of the predicate `RechargeNotification(5)`. Moreover, the planner adds the ordering constraints $Start \prec ReceiveSIMRecharge(5)$ and $ReceiveSIMRecharge(5) \prec Finish$. Now there is one new open precondition, that is `RechargeNotification(5)`. Since no effect in the Start task satisfies this precondition, the planner checks again within the list of task definitions and finds that the *Ask for SIM Recharge(x)* task (again, with x=5) achieves the predicate `RechargeNotification(5)`. A set of new ordering constraint are stated : $Start \prec AskforSIMRecharge(5)$, and $AskforSIMRecharge(5) \prec ReceiveSIMRecharge(5)$. Since the preconditions of the *Receive SIM Recharge(5)* task are both already satisfied by the effects of the task Start, the planner first certifies that there are no more open preconditions and then verifies whether the plan found is a solution to the original planning problem. The plan is returned as a list of ordering constraints between the tasks of the recovery plan; in our example, such list is composed by the following constraints : $Start \prec AskforSIMRecharge(5)$, $AskforSIMRecharge(5) \prec ReceiveSIMRecharge(5)$ and $ReceiveSIMRecharge(5) \prec Finish$. Fig. 8.b shows the resulting plan executed as a *worklet*.

VI. CONCLUSIONS

In this paper we have presented a general approach and a conceptual architecture for featuring automatic adaptivity in PMSs, based on a kind of declarative specification of tasks, including pre-conditions and effects, and the use of planning techniques. We have shown the feasibility of the approach by discussing its deployment on top of YAWL. The strength of the approach lies in the ability to incorporate execution feedback directly into the plan, without blocking directly the execution of the main process. This can result in a reduction of the overall response time. In fact, during the plan synthesis, if a positive event occurs (such as an external event or a task whose effects are to “adjust” the compromised situation), the system is able to take advantage of such opportunity without a new plan. However, even though our intent is to make the planning process very responsive, there still remains a synchronization process between planning and execution, which can require a significant response time.

Future works include an extensive validation of real processes, by considering not only classical business ones but also more collaborative processes in which the exchange of meaningful artefacts is the real drive in the process enactment. Such tests will provide useful insights on the cases in which an automatic approach is convenient wrt. more traditional exception handlers defined at design-time.

ACKNOWLEDGMENTS

This work has been partly supported by SAPIENZA Università di Roma through the grants FARI 2010 and TESTMED. The authors want to thank Arthur H.M. ter Hofstede for useful insights and discussions.

REFERENCES

[1] D. Chiu, Q. Li, and K. Karlapalem, “A Logical Framework for Exception Handling in ADOME Workflow Management System,” in *Proceedings of the 12th International Conference of Advanced Information Systems Engineering (CAiSE)*, 2000.

[2] R.-M. M. Dolores, B. Daniel, C. Amedeo, and O. Angelo, “Integrating Planning and Scheduling in Workflow Domains,” *Expert Syst. Appl.*, vol. 33, no. 2, 2007.

[3] H. Ferreira and D. Ferreira, "An Integrated Life Cycle for Workflow Management Based on Learning and Planning," *Int. J. Cooperative Information Systems*, vol. 15, pp. 485–505, 2006.

[4] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni, "Exception Handling for Repair in Service-Based Processes," *IEEE Transactions on Software Engineering*, vol. 36, pp. 198–215, 2010.

[5] M. Gajewski, H. Meyer, M. Momotko, H. Schuschel, and M. Weske, "Dynamic Failure Recovery of Generated Workflows," in *Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA)*. IEEE Computer Society Press, 2005, pp. 982–986.

[6] K. Goser, M. Jurisch, H. Acker, U. Kreher, M. Lauer, S. Rinderle-Ma, M. Reichert, and P. Dadam, "Next-generation Process Management with ADEPT2," in *Demonstration Program of the 5th International Conference on Business Process Management (BPM)*, 2007.

[7] C. Hagen and G. Alonso, "Exception Handling in Workflow Management Systems," *IEEE Trans. Software Engineering*, vol. 26, pp. 943–958, 2000.

[8] P. Jarvis, J. Moore, J. Stader, A. Macintosh, A. C. du Mont, and P. Chung, "Exploiting AI Technologies to Realise Adaptive Workflow Systems," *Proceedings of the AAAI Workshop on Agent-Based Systems in the Business Context*, 1999.

[9] R. Lenz and M. Reichert, "IT Support for Healthcare Processes - Premises, Challenges, Perspectives," *Data Knowl. Eng.*, vol. 61, pp. 39–58, 2007.

[10] A. Marrella and M. Mecella, "Continuous Planning for Solving Business Process Adaptivity," in *Proceedings of the 12th International Working Conference on Business Process Modeling, Development and Support (BPMDS)*, 2011, pp. 118–132.

[11] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL - The Planning Domain Definition Language," Yale Center for Computational Vision and Control, Tech. Rep., 1998.

[12] M. Mecella, "Adaptive Process Management. Issues and (Some) Solutions," 2008. [Online]. Available: http://www.dis.uniroma1.it/∼mecella/publications/PMS/MECELLA@PROGILITY2008.ppt

[13] R. Müller, U. Greiner, and E. Rahm, "AGENTWORK: a Workflow System Supporting Rule-based Workflow Adaptation," *Data & Knowledge Engineering*, vol. 51, pp. 223–256, 2004.

[14] K. Myers, "CPEF: A Continuous Planning and Execution Framework," *AI Magazine*, vol. 20, pp. 63–69, 1999.

[15] S. Penberthy and D. Weld, "UCPOP: A Sound, Complete, Partial Order Planner for ADL," in *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 1992, pp. 103–114.

[16] N. Russell, W. van der Aalst, and A. ter Hofstede, "Workflow Exception Patterns," in *Proceedings of the 18th International Conference of Advanced Information Systems Engineering (CAiSE)*, 2006, pp. 288–302.

[17] A. ter Hofstede, W. van der Aalst, M. Adams, and N. Russell, *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009.

[18] W. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative Workflows: Balancing between Flexibility and Support," *Computer Science - Research and Development*, vol. 23, no. 2, pp. 99–115, 2009.

[19] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change Patterns and Change Support Features - Enhancing Flexibility in Process-aware Information Systems," *Data Knowl. Eng.*, vol. 66, pp. 438–466, 2008.

[20] B. Weber, M. Reichert, S. Rinderle-Ma, and W. Wild, "Providing Integrated Life Cycle Support in Process-aware Information Systems," *Cooperative Information Systems*, vol. 18, pp. 115–165, 2009.

[21] M. Weske, "Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS)*, 2001.