Supporting adaptiveness of cyber-physical processes through action-based formalisms

Andrea Marrella [a][*] Massimo Mecella [a]
Sebastian Sardiña [b]
[a] *Sapienza Universitá di Roma, Italy*
[b] *RMIT University, Melbourne, Australia*

Cyber Physical Processes (CPPs) refer to a new generation of business processes enacted in many application environments (e.g., emergency management, smart manufacturing, etc.), in which the presence of Internet-of-Things devices and embedded ICT systems (e.g., smartphones, sensors, actuators) strongly influences the coordination of the real-world entities (e.g., humans, robots, etc.) inhabitating such environments. A Process Management System (PMS) employed for executing CPPs is required to automatically adapt its running processes to anomalous situations and exogenous events by minimising any human intervention. In this paper, we tackle this issue by introducing an approach and an adaptive Cognitive PMS, called SmartPM, which combines process execution monitoring, unanticipated exception detection and automated resolution strategies leveraging on three well-established action-based formalisms developed for reasoning about actions in Artificial Intelligence (AI), including the situation calculus, IndiGolog and automated planning. Interestingly, the use of SmartPM does not require any expertise of the internal working of the AI tools involved in the system.

Keywords: Cyber-Physical Processes, Process Adaptation and Recovery, Situation Calculus, IndiGolog, Automated Planning

1. Introduction

In the last years, we have witnessed the emergence of new computing paradigms, such as Health 2.0 [69,13], Industry 4.0 [38] and mobile-based emergency management (e.g., see [17] and the ISCRAM conference on Information Systems for Crisis Response and Management, `http://www.iscram.org/`), in which the interplay of Internet-of-Things (IoT) devices, i.e., devices attached to the Internet, Artificial Intelligence (AI) based techniques, cloud computing, Software-as-a-Service (SaaS), and Business Process Management (BPM) create the so-called *cyber-physical environments* and give rise to the concept of *Cyber-Physical Systems* (CPSs). The role of CPSs is to monitor the *physical processes* enacted in cyber-physical environments, create a virtual copy of the physical world and make decentralized decisions, by introducing methods of self-optimization, self-configuration, self-diagnosis, and intelligent support of workers in their increasingly complex work [64].

A relevant aspect in these environments lies in the fundamental role played by the processes orchestrating the different actors (software, humans, robots, etc.) involved in the CPS. We refer to these processes as *cyber-physical processes* (CPPs), whose enactment is influenced by user decision making and coupled with contextual data and knowledge production coming from the cyber-physical environment. According to [33], *Cognitive Process Management Systems* (CPMSs) are the key technology for supporting CPPs.

A conventional Process Management System (PMS) is a software system that manages and executes processes on the basis of *process models* [20]. The basic constituents of a process model are *tasks*, describing the various activities to be performed by process participants. The procedural rules to control such tasks, described by so-called "routing" constructs such as sequences, loops, parallel, and alternative branches, define the *control flow* of the process. A PMS, then, takes a process model (containing the process' tasks and control flow) and manages the process routing by deciding which tasks are enabled for execution. Once a task is ready for execution, the PMS assigns it to those participants capable of carrying it on. The representation of a single execution of a process model is called a *process instance* [19].

A PMS is said to be *cognitive* when it involves additional processing constructs that are at a semantic level higher than those of conventional PMSs. These constructs are called *cognitive BPM constructs* and tend to include data-driven activities and constraints, goals, and plans [33]. Their usage can open opportunities for

[*]Corresponding author: marrella@diag.uniroma1.it

new levels of automation and support for CPPs, such as - for example - *the automated synthesis of adaptation strategies at run-time exploiting solely the process knowledge and its expected evolution*.

During the enactment of CPPs, variations or divergence from structured reference models are common due to exceptional circumstances arising (e.g., autonomous user decisions, exogenous events, or contextual changes), thus requiring the ability to properly *adapt* the process behavior. *Process adaptation* can be seen as the ability of a process to react to exceptional circumstances (that may or may not be foreseen) and to adapt/modify its structure accordingly. Exceptions can be either *anticipated* or *unanticipated*. An anticipated exception can be planned at design-time and incorporated into the process model, i.e., a (human) process designer can provide an *exception handler* that is invoked during run-time to cope with the exception. Conversely, *unanticipated exceptions* refer to situations, unplanned at design-time, that may emerge at run-time and can be detected only during the execution of a process instance, when a mismatch between the computerized version of the process and the corresponding real-world process occurs. To cope with those exceptions, a PMS is required to allow *ad-hoc process changes* for adapting running process instances in a situation- and context-dependent way.

The fact is that, in cyber-physical environments, the number of possible anticipated exceptions is often too large, and traditional manual implementation of exception handlers at design-time is not feasible for the process designer, who has to anticipate all potential problems and ways to overcome them in advance [58]. Furthermore, anticipated exceptions cover only partially relevant situations, as in such scenarios many unanticipated exceptional circumstances may arise during the process instance execution. Therefore, the process designer often lacks the needed knowledge to model all the possible exceptions at the outset, or this knowledge can become obsolete as process instances are executed and evolve, by making useless her/his initial effort.

To tackle this issue, in this paper we present and review our ongoing work on SmartPM[1], a CPMS able to *automatically adapt CPPs at run-time* when *unanticipated exceptions* occur, thus requiring no specification of recovery policies at design-time. The general idea builds on the dualism between an *expected reality* and a *physical reality*: process execution steps and exogenous events have an impact on the physical reality and any deviation from the expected reality results in a mismatch to be removed to allow process progression.

To that end, we resort to three popular *action-based formalisms* and *technologies* from the field of Knowledge Representation and Reasoning (KR&R): situation calculus [60], IndiGolog [14], and automated planning [49,28]. We use the situation calculus logical formalism to model the underlying domain in which processes are to be executed, including the description of available tasks, contextual properties, tasks' preconditions and effects, and the initial state. On top of such model, we use the IndiGolog high-level agent programming language for the specification of the structure and control flow of processes. Importantly, we customize IndiGolog to monitor the online execution of processes and detect potential mismatches between the model and the actual execution. If an exception invalidates the enactment of the processes being executed, an external state-of-the-art planner is invoked to synthesise a recovery procedure to adapt the faulty process instance.

The choice of adopting action-based formalisms from the KR&R field is motivated by their ability to provide the right *cognitive* level needed when dealing with dynamic situations in which data (values) play a relevant role in system enactment and automated reasoning over the system progress. In the field of BPM, many other formalisms (in particular Petri Nets-based and process algebras) have been successfully adopted for process management, but all of them are somehow based on synthesis techniques of the control-flow, when considering their automated reasoning capabilities. This implies the level of abstraction over dealing with data and dynamic situations is fairly "raw", when compared with KR&R methods in which automated reasoning over data values and situations is much more developed [59,5,60]. As we will see below, the choice of KR&R technologies allows us to develop a principled, clean and simple-to-manage framework for process adaptation based on relevant data manipulated by the process, without compromising efficiency and effectiveness of the proposed solution.

The rest of the paper is organized as follow. In Section 2, after presenting an overview of our application scenario, i.e., a real CPP enacted in an Italian ceramic-ware factory, we provide a conceptual architecture for implementing adaptive CPMSs for CPPs. In Section 3

[1]The technical content regarding SmartPM follows and improves our previous work [42] by introducing the multi-instance construct (cf. Section 5), whereas the real-world application scenario of Section 2.1, the conceptual architecture of Section 2.2, the cyber-physical layer implemented in the SmartPM system to cope with CPPs of Section 3 and the usability evaluation of Section 6 are specific to this contribution.

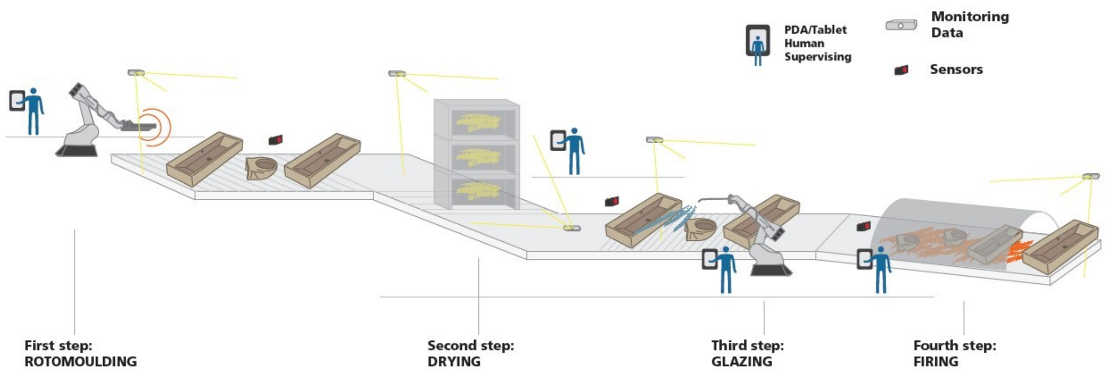


Fig. 1. Overview of the production process of a ceramic sanitary ware factory.

we introduce the general SmartPM approach to handle unanticipated exceptions in CPPs, and we present the architecture of the implemented SmartPM system. Then, in Sections 4 and 5, we provide the formalization of the SmartPM approach through action-based formalisms. In Section 6, we investigate the practical applicability of SmartPM from the user perspective through a set of experiments performed with real users. Finally, in Section 7, we discuss the state-of-the-art approaches to process adaptation, while in Section 8 we conclude the article by drawing conclusions and discussing limitations and future work.

2. Cyber-Physical Processes

CPSs are having widespread applicability and proven impact in multiple areas [53], like aerospace, automotive, traffic management, healthcare, manufacturing, emergency management, entertainments. According to [39], any physical environment which contains computing-enabled devices can be considered as a cyber-physical environment.

The trend of managing CPPs, i.e., processes enacted in cyber-physical environments, has been fueled by two main factors. On the one hand, the recent development of powerful mobile computing devices providing wireless communication capabilities have become useful to support mobile workers to execute tasks in such dynamic settings. On the other hand, the increased availability of sensors disseminated in the world has lead to the possibility to monitor in detail the evolution of several real-world objects of interest. *The knowledge extracted from such objects allows to depict the contingencies and the context in which processes are carried out, by providing a fine-grained monitoring, mining, and decision support for them.*

In this section, we first present a real-world application scenario that comes from the smart manufacturing domain to motivate the need of designing an adaptive CPMS in cyber-physical environments (cf. Section 2.1). Then, we devise a conceptual architecture to concretely build such a CPMS (cf. Section 2.2).

2.1. Application Scenario

Smart manufacturing is rapidly transforming how products are invented and manufactured. A modern manufacturing plant uses sensors, actuators and computerized controls to manage each specific stage of a manufacturing process. The challenge is to integrate individual stages of manufacturing production enabling data sharing throughout the plant, with the purpose to allow complete production lines to run with real-time flexibility in order to ensure trouble-free manufacturing and optimize outputs [9].

In a modern *ceramic factory*, a new product (e.g. a sanitary item) is first designed with the help of Computer-Aided Design (CAD) tools, and then passed to the production line. Fig. 1 shows a fragment of the real production process of an Italian ceramic sanitary-ware factory. Each element of the factory is attached to sensors and actuators, and a process-oriented system is used to coordinate the working of the robot arms and the machinery employed in the various steps of the production process.

Specifically, starting from a CAD model, an initial mould model of a ceramic product is generated through a *rotomoulding* step. A mould model has an higher volume (of about 11%-12%) if compared to the final product's volume, since in the subsequent steps of the production process, which are the *drying*, *glazing* and *firing* steps, it will lose part of its volume and will be affected by deformation influenced by different factors (gravity, humidity, glazing temperature, etc.).

The BPMN process in Fig. 2(a) describes (in a simplified way) the underlying workflow of such a production process. Notice that the process is a sequence of *multi-instance tasks*. The purpose of a multi-instance task is to generate multiple independent instances of the task that run sequentially (if the task marker includes three horizontal lines) or concurrently (if the task marker includes three vertical lines). In our example, the number of generated instances for any task is determined by the number of *ceramic elements* that are not *broken* at the moment of the task execution.

The initial collection of ceramic elements (for the sake of simplicity we are considering three elements, denoted as *obj1*, *obj2* and *obj3*) is delivered through a specific *conveyor belt* to the first step of the production process, that is *rotomoulding*, from a starting location (denoted as *loc_start*). The same conveyor belt will also move the ceramic elements between each step of the production line. After the *firing step*, each element is moved in a final location (denoted as *loc_end*).

Each of the steps of the production process is performed by a different static *robotic arm* or specialized *machinery* located in a fixed position of the factory:

- the rotomoulding step is performed by the robotic arm *rb_arm_1* that is located in *loc_rotomoulding*;
- the drying step is performed by the drying system *dryer_1* that is located in *loc_drying*;
- the glazing step is performed by the robotic arm *rb_arm_2* that is located in *loc_glazing*;
- the firing step is performed by the oven system *oven_1* that is located in *loc_firing*.

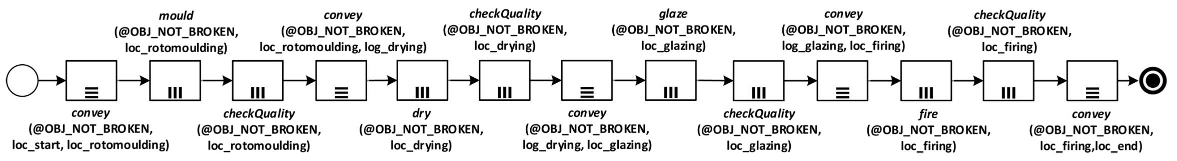
When the task associated to any of the steps completes, a *quality check* is performed by activating a *digital 3D scanner* that analyzes the surface of the ceramic elements to identify the presence of ruptures or defects.

Finally, a maintenance crew composed by two skilled technicians, called *actors*, and a *moving robot rb_mv_1*, initially all located in a warehouse containing ceramic elements with defects (situated in *loc_warehouse*), is ready to intervene if something wrong happens during the enactment of the production process. For example, actor *act1* is able to fix the working environment parameters of a machinery during the manipulation of a ceramic element if they may cause defects to the element itself, while *act2* is in charge of removing debris of broken ceramic elements to keep the conveyer belt clean. The moving robot *rb_mv_1*, in turn, is designed to pick up ceramic elements with defects from the conveyor belt and deposit them in the warehouse. Notice that it is required that *at least one actor is always present in the warehouse* for guaranteeing a correct coordination of the operations of deposit. When the battery of a moving robot is discharged, actor *act2* can charge it.

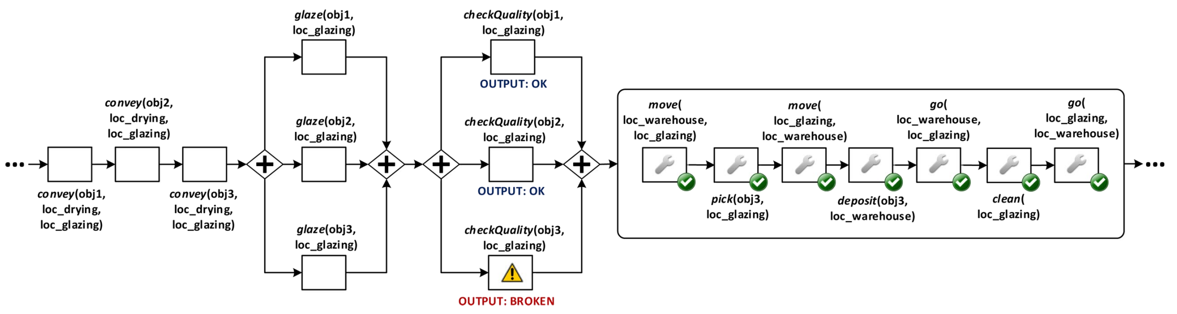
During the enactment of the production process, there is a wide range of exceptions that can ensue. Some of these exceptions may be caused by the *deformation* of ceramic materials during the drying and glazing steps [35], where an incorrect thermal expansion of the body of the ceramic elements may cause their rupture. While nowadays the data collected by 3D scanning during the CHECKQUALITY tasks are used for the optimization of the design of the CAD model *off-line*, an intelligent adaptive CPMS could act *on-line* by repairing the production process through one or more recovery procedures that possibly mitigate the consequences of the deformation on the performances of the whole production process.

For example, in Fig. 2(b), the outcome of the CHECKQUALITY task denotes that *obj3* is broken. Therefore, if a deformation after the glazing step is evaluated as critic and not anymore “adjustable”, it is useless to proceed to the next step of the process, i.e., to fire the ceramic element. To that end, provided that *rb_mv_1* has enough battery charge, the CPMS may first instruct *rb_mv_1* to reach *loc_glazing* in order to pick up *obj3*, and then to move back towards *loc_warehouse* to deposit *obj3* in the warehouse. Then, actor *act2*, if free from any other task assignment, may be instructed to reach *loc_glazing* and clean the conveyor belt from possible debris. The corresponding updated process is shown in Fig. 2(b), with the encircled section being the recovery (adaptation) procedure. Notice that after the recovery procedure, the enactment of the original process can be resumed to its normal flow.

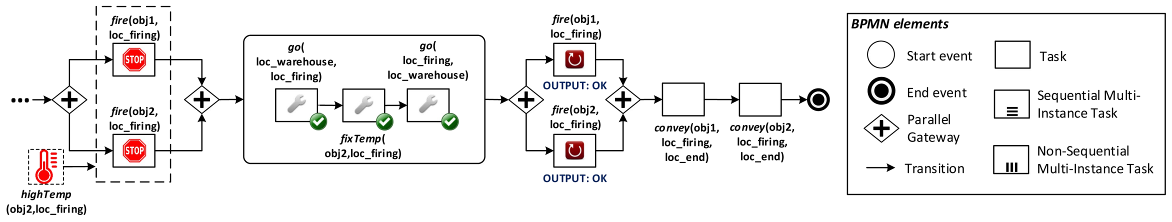
The execution of a manufacturing process can also be jeopardized by the occurrence of *exogenous events*. Indeed, exogenous events could change, in asynchronous manner, some contextual properties of the environment in which the process is under execution, hence possibly requiring the process to be adapted accordingly. In our example, since any step of the production process is monitored through the usage of specific sensors that allow the online tracking of the relevant environmental parameters (temperature, humidity, pressure, etc.), it may happen that an incorrect value of one of such parameters affects the quality of the transformations of the ceramic material. For example, suppose that during the firing step the environmental


mould (@OBJ_NOT_BROKEN, loc_rotomoulding)
convey (@OBJ_NOT_BROKEN, loc_rotomoulding, log_drying)
checkQuality (@OBJ_NOT_BROKEN, loc_drying)
glaze (@OBJ_NOT_BROKEN, loc_glazing)
convey (@OBJ_NOT_BROKEN, log_glazing, loc_firing)
checkQuality (@OBJ_NOT_BROKEN, loc_firing)
convey (@OBJ_NOT_BROKEN, loc_start, loc_rotomoulding)
checkQuality (@OBJ_NOT_BROKEN, loc_rotomoulding)
dry (@OBJ_NOT_BROKEN, loc_drying)
convey (@OBJ_NOT_BROKEN, log_drying, loc_glazing)
checkQuality (@OBJ_NOT_BROKEN, loc_glazing)
fire (@OBJ_NOT_BROKEN, loc_firing)
convey (@OBJ_NOT_BROKEN, loc_firing,loc_end)

(a) Main production process


glaze(obj1, loc_glazing)
checkQuality(obj1, loc_glazing)
OUTPUT: OK
convey(obj2, loc_drying, loc_glazing)
glaze(obj2, loc_glazing)
checkQuality(obj2, loc_glazing)
OUTPUT: OK
move(loc_warehouse, loc_glazing)
move(loc_glazing, loc_warehouse)
go(loc_warehouse, loc_glazing)
go(loc_glazing, loc_warehouse)
pick(obj3, loc_glazing)
deposit(obj3, loc_warehouse)
clean(loc_glazing)
convey(obj1, loc_drying, loc_glazing)
convey(obj3, loc_drying, loc_glazing)
glaze(obj3, loc_glazing)
checkQuality(obj3, loc_glazing)
OUTPUT: BROKEN

(b) Failed CHECKQUALITY(*obj3,loc_glazing*)


fire(obj1, loc_firing)
fire(obj2, loc_firing)
highTemp (obj2,loc_firing)
go(loc_warehouse, loc_firing)
go(loc_firing, loc_warehouse)
fixTemp(obj2,loc_firing)
fire(obj1, loc_firing)
OUTPUT: OK
fire(obj2, loc_firing)
OUTPUT: OK
convey(obj1, loc_firing, loc_end)
convey(obj2, loc_firing, loc_end)
BPMN elements
Start event
End event
Parallel Gateway
Transition
Task
Sequential Multi-Instance Task
Non-Sequential Multi-Instance Task

(c) Fix temperature of the oven machine to deal with the exogenous event HIGHTEMP(*obj2,loc_firing*)

Fig. 2. The production process of a ceramic sanitary ware factory and its adaptation.

temperature of *obj2* reaches a dangerous value (see Fig. 2(c)). In such a case, the CPMS needs first to stop all the running tasks, and then to find a recovery procedure that allows to “normalize” the situation before it causes defects to the ceramic materials under firing. The corresponding adapted process is shown in Fig. 2(c), and consists of instructing *act1* to reach *loc_firing* for configuring the oven system to modify its temperature to a reasonable value (see task FIXTEMP), for a correct firing of *obj2*.

The idea underlying the two above (simple) examples of adaptation is therefore to reason over the big data generated by the 3D scanners and the sensors and leave an adaptive CPMS to detect and resolve disturbances at run-time before they escalate and result in product defects, with the target to optimize the whole production process by reducing the time to production.

The point is that it is not adequate to assume that the process designer can pre-define all possible recovery activities for dealing with unanticipated exceptions and exogenous events in environments that are cyber-physical as the one just described: the recovery procedure will depend on the actual context (e.g., the positions of actors and robots, robot’s battery levels, the range of the sensors, whether a location has become dangerous to get it, etc.) and there are too many of them to be considered. Because of that, there is not always a clear anticipated correlation between a change in the context and a change in the process.

2.2. A Conceptual Architecture for CPPs

The previous application scenario emphasizes that the management of a CPP requires additional challenges to be considered if compared with a traditional “static” business process. On the one hand, there is the need of representing explicitly real-world objects and technical aspects like device capability constraints, sensors range, actors and robots mobility, etc. On the other hand, since cyber-physical environments are intrinsically “dynamic”, a CPMS providing real-time monitoring and automated adaptation features during process execution is required.

To this end, the role of the data perspective becomes fundamental. Data, including information pro-

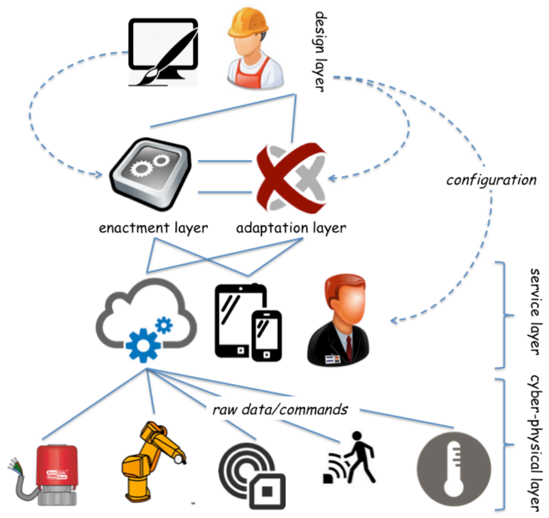


Fig. 3. A conceptual architecture for CPPs.

cessed by process tasks as well as contextual information, are the main driver for triggering process adaptation, as focusing on the control flow perspective only - as traditional PMSs do - would be insufficient.

In fact, in a cyber-physical environment, a CPP is genuinely knowledge and data centric: its control flow must be coupled with contextual data and knowledge production and process progression may be influenced by user decision making. This means that traditional imperative models have to be extended and complemented with the introduction of specific *cognitive constructs* such as *data-driven activities* and *declarative elements* (e.g., tasks preconditions and effects) which enable a precise description of data elements and their relations, so as to go beyond simple process variables, and allow establishing a link between the control flow and the data perspective.

Starting from the above considerations, coupled with the experience gained in the area and lessons learned from several projects involving CPSs (a partial list of such projects is detailed in the Acknowledgments), we have devised a conceptual architecture to build a CPMS for the management of CPPs, which supports the so-called *Plan-Act-Learn* cycle for cognitively-enabled processes [33]. As shown in Fig. 3, we identified 5 main architectural layers that we present in a bottom-up fashion.

The ***cyber-physical layer*** consists mainly of two classes of physical components: *(i)* sensors (such as GPS receivers, RFID chips, 3D scanners, cameras, etc.) that collect data from the physical environment by monitoring real-world objects and *(ii)* actuators (robotic arms, 3D printers, electric pistons, etc.), whose effects affect the state of the physical environment. The cyber-physical layer is also in charge of providing a physical-to-digital interface, which is used to transform *raw* data collected by the sensors into machine-readable events, and to convert *high-level* commands sent by the upper layers into *raw* instructions readable by the actuators. The cyber-physical layer does not provide any intelligent mechanism neither to clean, analyse or correlate data, nor to compose high-level commands into more complex ones; such tasks are in charge of the upper layers.

On top of the cyber-physical layer lies the ***service layer***, which contains the set of services offered by the real-world entities (software components, robots, agents, humans, etc.) to perform specific process tasks. In the service layer, available data can be aggregated and correlated, and high-level commands can be orchestrated to provide higher abstractions to the upper layers. For example, a smartphone equipped with an application allowing to sense the position and the posture of a user is at this layer, as it collects the raw GPS, accellerometer and motion sensor data and correlates them to provide discrete and meaningful information.

On top of the service layer, there are two further layers interacting with each other. The ***enactment layer*** is in charge of *(i)* enacting complex processes by deciding which tasks are enabled for execution, *(ii)* orchestrating the different available services to perform those tasks and *(iii)* providing an execution monitor to detect the anomalous situations that can possibly prevent the correct execution of process instances. The execution monitor is responsible for deciding if process adaptation is required. If this is the case, the ***adaptation layer*** will provide the required algorithms to *(i)* reason on the available process tasks and contextual data and to *(ii)* find a recovery procedure for adapting the process instance under consideration, i.e., to re-align the process to its expected behaviour. Once a recovery procedure has been synthesized, it is passed back to the enactment layer for being executed.

Finally, the ***design layer*** provides a GUI-based tool to define new process specifications. A process designer must be allowed not only to build the process control flow, but also to explicitly formalize the data reflecting the contextual knowledge of the cyber-physical environment under observation. It is important to underline that data formalization must be performed without any knowledge of the internal working of the physical components that collect/affect data in the cyber-physical layer. In order to link tasks to contextual data, the GUI-based tool must go beyond the classical "task model" as known in the literature, by allowing the process designer to explicitly state what data may constrain a task execution or may be affected after a task completion. Finally, besides specifying the process, configuration files should also be produced to properly configure the enactment, the services and the sensors/actuators in the bottom layers.

Based on this conceptual architecture, we realized an approach – called SmartPM – and an implemented CPMS for adapting CPPs at run-time, whose details will be discussed in the next section.

3. The SmartPM Approach and System

SmartPM (Smart Process Management) is an approach and an adaptive CPMS implementing a set of techniques that enable to automatically adapt process instances at run-time in the presence of unanticipated exceptions, without requiring an explicit definition of handlers/policies to recover from tasks failures and exogenous events. SmartPM adopts a *layered service-based* approach to process management, i.e., *tasks are executed by services*, such as software applications, humans, robots, etc. Each task can be thus seen as a single step consuming input data and producing output data.

To monitor and deal with exceptions, the SmartPM approach leverages on [16]'s technique of adaptation from the field of agent-oriented programming, by specializing it to our CPP setting (see Fig. 4). We consider adaptation as *reducing the gap* between the *expected reality* **Exp**, the (idealized) model of reality used by the CPMS to reason, and the *physical reality* **Phy**, the real world with the actual conditions and outcomes. While **Phy** records what is *concretely* happening in the real environment during a process execution, **Exp** reflects what it is *expected* to happen in the environment. Process execution steps and exogenous events have an impact on **Phy** and any deviation from **Exp** results in a mismatch to be removed to allow process progression. At this point, a state-of-the-art automated planner is invoked to synthesise a recovery procedure that adapts the faulty process instance by removing the gap between the two realities.

To realize the above approach, the implementation of SmartPM covers the modeling, execution and monitoring stages of the CPP life-cycle. To that end, on the basis of the conceptual architecture for building CPMS presented in Section 2.2, the architecture of SmartPM relies on five architectural layers.

The ***design layer*** provides a graphical editor developed in Java that assists the process designer in the definition of the process model at design-time. Process knowledge is represented as a *domain theory* that includes all the contextual information of the domain of concern, such as the people/services that may be involved in performing the process, the tasks, the data and so forth. Data are represented through some *atomic terms* that range over a set of *data objects*, which depict entities of interest (e.g., capabilities, services, etc.), while atomic terms can be used to express properties of domain objects (and relations over objects). *Tasks* are collected in a repository and are described in terms of *preconditions* - defined over atomic terms - and *effects*, which establish their expected outcomes. Finally, a process designer can specify which *exogenous events* may be caught at run-time and which atomic terms will be modified after their occurrence. Once a valid domain theory is ready, the process designer uses the graphical editor to define the process control flow through the standard BPMN notation among a set of tasks selected from the tasks repository.

The ***enactment layer*** is in charge of managing the process execution. First of all, the domain theory specification and the BPMN process are automatically translated into situation calculus [60] and IndiGolog [14] readable formats. Situation calculus is used for providing a declarative specification of the domain of interest (i.e., available tasks, contextual properties, tasks preconditions and effects, what is known about the initial state). Then, an *executable model* is obtained in the form of an IndiGolog program to be executed through an IndiGolog engine. To that end, we customized an existing IndiGolog engine[2] to *(i)* build a physical/expected reality by taking the initial context from the external environment; *(ii)* manage the process routing; *(iii)* collect exogenous events from the external environment; *(iv)* monitor contextual data to identify changes or events which may affect process execution. Once a task is ready for being executed, the IndiGolog engine assigns it to a proper process participant (that could be a software, a human actor, a robot, etc.) that provides all the required capabilities for task execution.

The ***service layer*** acts as a middleware between process participants, the enactment layer and the cyber-physical layer. Specifically, in the service layer, process participants interact with the engine through a *Task Handler*, an interactive GUI-based software ap-

[2] `http://sourceforge.net/projects/indigolog/`

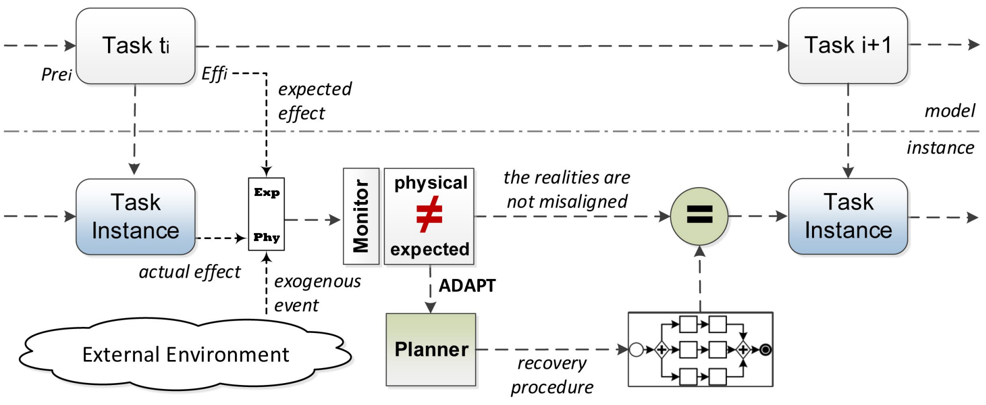


Fig. 4. An overview of the SmartPM approach.

plication realized for Android devices that supports the visualization/execution of assigned tasks by selecting an appropriate outcome. Possibly such an Android application can exploit sensors and actuators (e.g., an Arduino board connected through Bluetooth, as currently realized in our implementation), thus effectively offering services over the *cyber-physical layer*. Every step of the task life cycle - ranging from the assignment to the release of a task - requires an interaction between the IndiGolog engine and the task handlers. The communication between the IndiGolog engine and the task handlers is mediated by the *Communicator Manager* component (which is essentially a web server) and established using the Google Cloud Messaging service.

To enable the automated synthesis of a recovery procedure, the ***adaptation layer*** relies on the capabilities provided by a PDDL-based planner component (the LPG-td planner [29]), which assumes the availability of a planning problem, i.e., an initial state and a goal to be achieved, and of a planning domain definition that includes the actions to be composed to achieve the goal, the domain predicates and data types. Specifically, if process adaptation is required, we translate *(i)* the domain theory defined at design-time into a planning domain, *(ii)* the physical reality into the initial state of the planning problem and *(iii)* the expected reality into the goal state of the planning problem. The planning domain and problem are the input for the planner component. If the planner is able to synthesize a recovery procedure δ_a, the *Synchronization* component combines δ' (which is the remaining part of the faulty process instance δ still to be executed), with the recovery plan δ_a, builds an adapted process $\delta'' = (\delta_a; \delta')$ and converts it into an executable IndiGolog program so that it can be enacted by the IndiGolog engine. Otherwise, if no plan exists for the current planning problem, the control passes back to the process designer, who can try to manually adapt the process instance.

The ***cyber-physical layer*** is tightly coupled with the physical components available in the domain of interest. Since the IndiGolog engine can only work with defined discrete values, while data gathered from physical sensors have naturally continuous values, the system provides several web tools that allow process designers to associate some of the data objects defined in the domain theory with the continuous data values collected from the environment. For example, we developed several web tools to associate the data collected from sensors (GPS, temperature, noise level, etc.) to discrete values. In Section 6.1, we provide a concrete example of a location web tool that allows process designers to mark areas of interest from a real map and associate them to discrete locations. The mapping rules generated are then saved into the Communication Manager and retrieved at run-time to allow the matching of the continuous data values collected by the specific sensor into discrete data objects.

In the following sections, we will first present the formal approach of SmartPM developed through action-based formalisms for covering the *enactment* and *adaptation* layers of the architecture, and we then provide some technical details of the tools and of the plugins employed in the *design* and *service* layers for concretely interacting with the system.

4. AI Agent Programming and Planning

Before describing the theoretical framework underlying SmartPM, we first provide some preliminary notions required to understand the rest of the paper.

4.1. *Situation Calculus and Basic Action Theories*

The *situation calculus* is a logical language designed for representing and reasoning about dynamic domains [60]. The dynamic world is seen as progressing through a series of situations as a result of various *actions* being performed. A *situation* s is a first-order term denoting the sequence of actions performed so far. The special constant S_0 stands for the initial situation, where no action has yet occurred, whereas a binary function symbol $do(a,s)$ denotes the situation resulting from the performance of action a in situation s. A situation s is a sub-situation of s', denoted $s \sqsubseteq s'$, iff s is a prefix of s'. Formally, the relation is axiomatized as follows: $s \sqsubseteq s' \equiv [s = s' \vee (\exists a, s'').s' = do(a,s'') \wedge s \sqsubseteq s'']$.

In situation calculus, relations and functions whose value may change from one situation to the next are modeled by means of so-called *relational fluents* and *functional fluents*. They are denoted by predicate and function symbols (respectively) taking a situation term as their last argument. A special predicate $Poss(a,s)$ is used to state that action a is executable in situation s, whereas special (situation-independent) predicate $Exog(a)$ is used to denote that a is an exogenous event originated from the external environment. We write $\phi(\vec{x})$ to denote a formula whose free variables are among variables $\vec{x}$. A fluent-formula is one whose only situation term mentioned is situation variable s.

Within this language, one can formulate action theories describing how the world changes as the result of the available actions. A *basic action theory* (BAT) [60] $\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{poss} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{una}$ includes:

- Σ domain-independent foundational axioms to describe the structure of situations and some auxiliary relations like $\sqsubseteq$ and $Executable(s)$;
- $\mathcal{D}_{ssa}$ one successor state axiom per fluent capturing the effects and non-effects (i.e., *frame*) of actions;
- $\mathcal{D}_{poss}$ one precondition axiom per action specifying when the action is executable;
- $\mathcal{D}_{una}$ unique name axioms for actions;
- $\mathcal{D}_{S_0}$ initial state axioms describing what is true initially in S_0, and auxiliary situation independent axioms for predicate $Exog(a)$.

In particular, the *successor state axiom* (SSA) for a relational fluent $F(\vec{x},s)$ is an axiom of the form $F(\vec{x},do(a,s)) \equiv \Psi_F(\vec{x},a,s)$,[3] where $\Psi_F(\vec{x},a,s)$ is a fluent-formula characterizing the dynamics of fluent $F(\vec{x},s)$. When $F(\vec{x},s)$ is a functional fluent, its successor state axiom has the form $[F(\vec{x},do(a,s)) = v] \equiv \Gamma_F(\vec{x},a,v,s)$, where $\Gamma_F(\vec{x},a,v,s)$ states that the fluent takes value v when action a is executed in situation s and satisfies the functional constraint $\models (\forall \vec{x},a,s)\exists v.\Gamma_F(\vec{x},a,v,s) \wedge (\forall v').v' \neq v \supset \neg\Gamma_F(\vec{x},a,v',s)$. Importantly, $\Psi_F(\vec{x},a,s)$ and $\Gamma_F(\vec{x},a,v,s)$ can accommodate Reiter's solution to the frame problem [60], avoiding to represent explicitly a large number of intuitively obvious non-effects. In addition, precondition axioms are of the form $Poss(a(\vec{x}),s) \equiv \Pi_a(\vec{x},s)$, where $\Pi_a(\vec{x},s)$ is a fluent-formula defining the conditions under which action a can be legally executed in situation s. Using *Poss*, we can define what it means for a situation s to be executable, using the following definition:

$$Executable(s) \equiv s = S_0 \vee (\exists a, s').s = do(a,s') \wedge Poss(a,s') \wedge Executable(s').$$

[3]Free variables are assumed to be universally quantified.

4.2. *The IndiGolog high-level language*

On top of situation calculus action theories, logic-based programming languages can be defined, which, in addition to the primitive actions, allow the definition of complex actions. In particular, we focus on IndiGolog [14], the latest in the Golog-like family of programming languages for autonomous agents providing a formal account of interleaved action, sensing, and planning. IndiGolog programs are meant to be executed *online*, in that, at every step, a legal next action is selected for execution, performed in the world, and its sensing outcome gathered. To account for planning, a special look-ahead construct $\Sigma(\delta)$—the search operator—is provided to encode the need for solving (i.e., finding a complete execution) program δ offline.

IndiGolog allows us to define every well-structured process as defined in [70]; it is equipped with all standard imperative constructs (e.g., sequence, conditional, iteration, etc.) to be used on top of situation calculus primitives actions. An IndiGolog program is meant to run relative to a BAT. Here we concentrate on the fragment defined by the following constructs:

a	atomic action
$\phi?$	test for a condition
$\delta_1;\delta_2$	sequence
$\pi x.\delta(x)$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	while loop
proc $P(\vec{x})$ **do** $\delta(x)$ **endProc**	procedure
$\delta_1 \| \delta_2$	concurrency

$\delta_1 \rangle\rangle \delta_2$	prioritized concurrency
$\langle \phi \rightarrow \delta \rangle$	interrupt
$\Sigma(\delta)$	lookahead search

Test program $\phi?$ can be executed if condition ϕ holds true, whereas program $\pi x.\delta(x)$ executes program $\delta(x)$ for *some* nondeterministic choice of a binding for variable x, and δ^* executes δ zero, one, or more times. The interleaved concurrent execution of two programs is represented with constructs $\delta_1 \| \delta_2$ and $\delta_1 \rangle\rangle \delta_2$; the latter considering δ_1 at higher priority level (i.e., δ_2 can perform a step only if δ_1 is blocked or completed). The interrupt construct $\langle \phi \rightarrow \delta \rangle$ states that program δ ought to be executed to completion if ϕ happens to become true. Let's focus on it:

```
$\langle \phi \rightarrow \delta \rangle \stackrel{\text{def}}{=}$ **while** *Interrupts_running* **do**
if $\phi$ then $\delta$ else false? endIf
endWhile
```

To see how this works, first assume that the special fluent *Interrupts_running* is true. When an interrupt $\langle \phi \rightarrow \delta \rangle$ gets control from higher priority processes, suspending any lower priority processes that may have been advancing, it repeatedly executes δ until ϕ becomes false. Once the interrupt body δ completes its execution, the suspended lower priority processes may resume. The control release also occurs if ϕ cannot progress (e.g., since no action meets its precondition). Finally, IndiGolog incorporates the so-called search construct $\Sigma(\delta)$, which performs lookahead reasoning on δ to guarantee that a full, terminating, execution of δ will be eventually achieved. Concretely, every step performed on δ will be one that is part of a terminating execution (see [14] for its formal semantics).

By properly combining prioritized concurrency and interrupts, together with IndiGolog's default online execution style, it is possible to design processes that are sufficiently open and reactive to dynamic environments. Furthermore, by resorting to the search operator, one can specify local places in programs where lookahead reasoning is required. Both aspects will end up being fundamental for our adaptive process management framework in the next sections.

4.3. AI Automated Planning

Planning systems are problem-solving algorithms that operate on explicit representations of states and actions [49,28]. PDDL [43] is the standard planning representation language; it allows one to formulate a *planning problem* $\mathcal{P} = \langle I, G, \mathcal{P}_D \rangle$, where I is the initial state, G is the goal state, and $\mathcal{P}_D$ is the planning domain. In turn, a planning domain $\mathcal{P}_D$ is built from a set of *propositions* describing the state of the world (a state is characterized by the set of propositions that are true) and a set of *actions* that can be executed in the domain. An *action schema* $a \in \Omega$ is of the form $a = \langle Par_a, Pre_a, Eff_a \rangle$, where Par_a is the list of *input parameters* for a, Pre_a defines the *preconditions* under which a can be executed, and Eff_a specifies the *effects* of a on the state of the world. Both preconditions and effects are stated in terms of the *propositions* in $\mathcal{P}_{\mathcal{D}}$. Propositions can be represented through boolean predicates and numeric fluents.

There exist several forms of planning in the AI literature. In this paper, we focus on planning techniques characterized by *fully observable*, *static* and *deterministic* domains, i.e., we rely on the *classical planning assumption* of a "perfect world description" [74]. Under this assumption, a solution for a planning problem $\mathcal{P}$ is a sequence of actions—a plan—whose execution transforms the initial state I into a state satisfying the goal G. Such a plan is computed in advance and then carried out (unconditionally). The field of automated planning has experienced huge advances in the last twenty years, leading to a variety of concrete solvers (i.e., planning systems) that are able to create plans with thousands of actions for problems containing hundreds of propositions. In this work, we represent planning domains and planning problems using PDDL 2.2 [21] (see Section 5.3 for a discussion on which specific features of the language we employed in our system).

5. The SmartPM approach through action-based formalisms

In this section we show how one can put together the three action-based formalisms and frameworks described above to build our adaptive CPMS. Specifically, we first describe how to explicitly formalize processes in situation calculus, which will be used to model the contextual information in which the process is meant to run. Then, starting from this formalization, we discuss how SmartPM has been coded by the interpreter of IndiGolog for devising a technique that automatically detect and recover from failures, and finally we show how planning systems will support the automated adaptation of a process when needed.

5.1. The Framework

5.1.1. SmartPM Basic Action Theory

Following [42], a situation calculus BAT $\mathcal{D}_{\textsf{SmartPM}}$ for a SmartPM application specifies:

1. the tasks and services of the domain of concern;
2. the framework for managing the task life-cycle;
3. the contextual setting in which processes operate;
4. the framework for the monitoring of processes.

To encode tasks and services, we use some non-fluent "rigid" predicates, i.e., their truth values do not depend on a situation term:

– $Service(srv)$: *srv* is a service (i.e., a process participant). The predicate can be specialized into further predicates, e.g., $Actor(srv)$, $MovingRobot(srv)$, etc., describing the specific services' roles;
– $Task(t)$: t is a task, e.g., GO or CHECKQUALITY, may denote the tasks of navigating or checking the quality of ceramic material after a manipulation;
– $Capability(c)$: c is a capability, e.g., *camera* may denote the ability to perform a 3D scanning of the surface of ceramic material;
– $Provides(srv, c)$: service *srv* provides capability c;
– $Requires(t, c)$: task t requires the capability c.

A service *srv* is able to perform certain task t iff *srv* provides all capabilities required by the task t. This is captured formally using the following abbreviation:[4]

$$\begin{aligned}&Capable(srv,t) \overset{\text{def}}{=}\\ &\quad Service(srv) \land Task(t) \land \forall c.Requires(t,c) \supset Provides(srv,c).\end{aligned}$$

To talk about concrete runs of tasks, we associate them with unique identifiers. A *task instance* is then a tuple $t:id$, where t is a task and *id* is an identifier.

The life-cycle of a generic task t involves the execution of four primitive actions executed by the CPMS and two external actions arising from services:

1. First, the CPMS assigns task instance $t : id$ to a service *srv* by performing primitive action $\textsc{assign}(srv, id, t, \vec{i}, \vec{o}_e)$, where $\vec{i}$ is an *input data vector* associated to $t:id$ and $\vec{o}_e$ is a vector of *expected* (sensing result) outputs.
2. When a service is ready for task execution, it generates the external action $\textsc{readyToStart}(srv, id, t)$.
3. Next, the CPMS performs primitive action $\textsc{start}(srv, id, t)$ to authorize the service in question to start carrying out the task instance.
4. When the service completes the task, it generates the external action $\textsc{finished}(srv, id, t, \vec{i}, \vec{o}_r)$, with $\vec{o}_r$ representing a vector of *physical* actual outcomes returned by the task execution (we use ε to denote the empty output).
5. At this point, the CPMS updates the properties (i.e., the fluents) to reflect the effects of the task just completed.
6. Finally, the CPMS acknowledges the completion of the task and releases the task from the service via primitive actions $\textsc{ackCompl}(srv, id, t)$ and $\textsc{release}(srv, id, t)$.

[4]An abbreviation is a predicate (with situation argument s) defined by means of a formula uniform in s. Abbreviations, unlike fluents, are not directly affected by actions.

Note that we suppose that in all application domains services, tasks, input and output data vectors range over finite values.

The above protocol for the life-cycle of tasks is captured by means of a set of *domain-independent* fluents and actions. For example, fluent $Free(srv,s)$ denotes whether a service *srv* is available for task assignments in situation s.

$$\begin{aligned}&Free(srv, do(a,s)) \overset{\text{def}}{=}\\ &\quad (\exists t, id)a = \textsc{release}(srv, id, t) \lor\\ &\quad [Free(srv,s) \land (\forall t, id, \vec{i}, \vec{o})a \neq \textsc{assign}(srv, id, t, \vec{i}, \vec{o})];\end{aligned}$$

That is, a service is free (for task assignment) after the execution of an action a iff a releases the service from some task assignment, or it was free before the execution of a and a does not assign a task to it.

Observe that SmartPM employs a *push-based* approach to task assignment. Specifically, the system dynamically selects an available service qualified for executing a given task and directly allocates the task item to the selected service. Conversely, traditional PMSs typically adopt a *pull-based* approach, with the system that offers each task to one or more services qualified for it and a service chooses one task for execution among the offered items. However, a pull-based approach to task assignment is not suitable for CPPs [58], which are usually highly critical and time demanding, and the risk exists to have some task(s) waiting indefinitely for being selected and executed. Therefore, in SmartPM each task is directly assigned to only one available service at a time, and each service gets assigned at most one task.

To record the expected output of task instance $t : id$ when assigned to a service, we define a functional fluent $ExOut(id,t,s)$, whose SSA is as follows:

$$\begin{aligned}&ExOut(id,t,do(a,s)) = \vec{o} \equiv\\ &\quad(\exists srv,\vec{i})a = \textsc{assign}(srv,id,t,\vec{i},\vec{o}) \lor ExOut(id,t,s) = \vec{o}.\end{aligned}$$

That is, the expected output of a task instance is determined by the assignment step (and never changes). Initial expected outcomes are initialized to "no value" (ε) via an axiom $(\forall t,id).ExOut(id,t,S_0) = \varepsilon$ in the set of axioms $\mathcal{D}_{S_0}$. Other similar fluents are used to manage the life-cycle of tasks.

The BAT shall also contain a set of *domain-dependent fluents*, together with their corresponding precondition and SSAs, capturing the contextual scenario in which the process is meant to be executed. We call such fluents *data fluents*. They can be seen as features of the world whose value may change from situation to situation. In general, data fluents will be affected upon the release of an assignment task, that is, whenever a task is considered fully executed. In addition, the actual outcome result of a task is used to define the fluent in question. Importantly, the SSAs will follow the following template:

$$\begin{aligned}&F(\vec{x},do(a,s)) = v \equiv\\ &\quad\gamma_F(\vec{x},a,v,s) \lor [F(\vec{x},s) = v \land (\neg\exists v)\gamma_F(\vec{x},a,v,s)],\end{aligned} \quad (1)$$

where $\gamma_F(\vec{x},a,v,s)$ states the conditions under which action a executed in situation s will cause data fluent $F(\vec{x},s)$ to take value v. In the context of our setting, two type of actions will be mentioned in $\gamma_F(\vec{x},a,v,s)$, namely, *(i)* actions of the form $\textsc{finished}(srv,id,T_k,\vec{i},\vec{o_r})$ reporting the completion of some task instance $T_k : id$ that would affect the data fluent and *(ii)* exogenous events that would also affect the value of the data fluent.

Example 5.1. Suppose that, in our manufacturing scenario, functional data fluent $At(srv,s)$ is used to keep track of the location of service srv in the domain. The fluent is affected by tasks GO and MOVE. Hence, the SSA for the fluent follows the template (1) above with $\gamma_F(\vec{x},a,v,s)$ being instantiated as follows:

$$\begin{aligned}&\gamma_{At}(srv,a,v,s) \overset{\text{def}}{=}\\ &\quad(\exists id,l_s,l_d,t)\\ &\qquad(a = \textsc{finished}(srv,id,t,[l_s,l_d],[v])\land\\ &\qquad t \in \{\textsc{go},\textsc{move}\} \land (Actor(srv) \lor MovingRobot(srv))) \lor\\ &\qquad(a = \textsc{pushed}(srv) \land Actor(srv) \land v = \text{"lost"}).\end{aligned}$$

In words, actor/robot srv is in location v if srv has just completed the task GO or MOVE whose actual physical outcome result is v, or if human service actor srv has just been unexpectedly pushed away from the factory during her/his working time (with the exogenous event PUSHED) and v records the fact that we lost track of its position. Observe that even when a navigation task has just been finished, the new location v of srv may happen to be different to the expected (destination) location l_d. □

Besides data fluents, an application will generally require further domain-dependent rigid predicates to represent the static properties of a contextual scenario. Such predicates do not change their value during process execution. For example, our application scenario requires to define a predicate *Neigh*(*loc1,loc2*) to expresses the neighborhood property between two locations of the factory.

Using the core data fluents and static predicates, one can also define helpful abbreviations, such as the following ones to capture that at least an actor must be always present (at any point in time) in the warehouse containing the ceramic materials with defects.

Example 5.2. The abbreviation $atLeastOne(s)$ denotes that in any situation at least an actor is present in the warehouse. This abbreviation will be used for exception monitoring and is defined as follows:

$$\begin{aligned}&atLeastOne(s) \overset{\text{def}}{=}\\ &\quad\exists srv.At(srv,s) = loc\_warehouse \land Actor(srv).\end{aligned}$$

The abbreviation $atLeastAnotherOne(srv,s)$ is used to check that at least an actor different from srv is present in the warehouse. This abbreviation will be used in the precondition of the navigation tasks for human actors, and is defined as follows:

$$\begin{aligned}&atLeastAnotherOne(srv,s) \overset{\text{def}}{=}\\ &\quad\exists srv2.At(srv2,s) = loc\_warehouse \land\\ &\quad Actor(srv2) \land (srv \neq srv2).\end{aligned}$$

□

Data fluents and abbreviations will be used for defining the *preconditions* of domain tasks. By doing so, the CPMS can reason, at run-time, about the active process instance relative to the current context. For example, while we have given above a generic precondition for assigning tasks to services, one can also incorporate domain-specific restrictions for task assignment.

Example 5.3. The following precondition axiom defines when the CPMS can assign a navigation task to a human actor and robot services:

$$\begin{array}{l} Poss(\textsc{assign}(srv, id, \textsc{go}, [l_s, l_d], [l_e]), s) \equiv \\ \quad Free(srv, s) \wedge Capable(srv, \textsc{go}) \wedge \\ \quad Actor(srv) \wedge (At(srv, s) = l_s) \wedge \\ \quad atLeastAnotherOne(srv, s) \wedge (l_e = l_d); \end{array}$$

$$\begin{array}{l} Poss(\textsc{assign}(srv, id, \textsc{move}, [l_s, l_d], [l_e]), s) \equiv \\ \quad Free(srv, s) \wedge Capable(srv, \textsc{move}) \wedge \\ \quad MovingRobot(srv) \wedge (At(srv, s) = l_s) \wedge (l_e = l_d) \wedge \\ \quad EnoughBattery(BatteryLevel(srv, s), MoveStep(l_s, l_d)). \end{array}$$

So, besides the service being available and capable of carrying out the navigation task, it also needs to be an actor located at the source location l_s and another different actor located in the warehouse. In addition, the expected outcome of the task instance ought to be the destination location l_d. A similar MOVE action can be used to instruct robots to move between two locations, though it is also required the robot's current battery level is enough to move from source l_s to destination l_d. This latter "constraint" is represented through the abbreviation *EnoughBattery*, which is defined over the fluent *BatteryLevel* (it records the battery charge level of a robot in a specific situation) and the static predicate *MoveStep* (it indicates the cost for a robot to move between two specific locations). $\square$

Before addressing the issue of how monitoring is modeled in CPPs, we provide the full successor state axiom of another data fluent which is affected by both actions and asynchronous exogenous events.

Example 5.4. Data fluent $Status(obj, s)$, which denotes the state of ceramic element obj at situation s, is affected by three domain tasks as well as by the exogenous event HIGHTEMP(obj, l). The latter indicates that the temperature of obj during the enactment of one of the step of the production process (situated at location l) is too high if compared with its normal expected value.

$$\begin{array}{l} Status(obj, do(a, s)) = v \equiv \\ \quad [(\exists srv, id, t, l) a = \textsc{finished}(srv, id, t, [obj, l], [v]) \wedge \\ \qquad t \in \{\textsc{checkquality}, \textsc{deposit}, \textsc{fixtemp}\}] \vee \\ \quad [(\exists l) a = \textsc{hightemp}(obj, l) \wedge v = \text{'}high\_temperature\text{'}] \vee \\ \quad [Status(obj, s) = v \wedge \\ \qquad \neg(\exists srv, id, t, l, v')(a = \textsc{finished}(srv, id, t, [l], [v']) \wedge \\ \qquad\quad t \in \{\textsc{checkquality}, \textsc{deposit}, \textsc{fixtemp}\}) \wedge \\ \qquad a \neq \textsc{hightemp}(obj, l)]. \end{array}$$

In words, tasks CHECKQUALITY, DEPOSIT and FIXTEMP are all meant to report the status of the ceramic element they operate on, upon completion. Moreover, high temperature events also change the status of the corresponding ceramic element. $\square$

This concludes the exposition of the first three aspects of a SmartPM action theory. Now we will focus on how our framework is able to deal with exception monitoring and management.

5.1.2. Exception Monitoring

We now turn our attention to the mechanism for automatically detecting failures/exceptions. In a nutshell, an exception occurs when a task does not produce the expected outcomes or an exogenous event arises.

The "physical" reality captures the actual value of fluents (and abbreviations) as observed in the system, and is encoded via the data fluents (and abbreviations), as described above, e.g., fluents $At(srv, s)$, $Status(obj, s)$ and abbreviation $atLeastOne(srv, s)$.

The "expected" reality, in turn, is captured with another set of fluents and abbreviations, one for each one in the physical reality. So, for every data fluent $F(\vec{x}, s)$, a new fluent $F_{\exp}(\vec{x}, s)$ (F-expected) is used to represent the value of $F(\vec{x}, s)$ in the "expected" (or "desired") execution. Technically, if $F(\vec{x}, do(a, s))$ is a relational data fluent with successor state axiom $F(\vec{x}, do(a, s)) \equiv \Psi_F(\vec{x}, a, s)$, then we build $F_{\exp}(\vec{x}, s)$'s counterpart as follows (the one for functional fluent is built in analogous way):

$$\begin{array}{l} F_{\exp}(\vec{x}, do(a, s)) \equiv \\ \quad [a = \textsc{align} \supset F(\vec{x}, s)] \vee \\ \quad [(\exists srv, id, t, \vec{i}, \vec{o_r}) a = \textsc{finished}(srv, id, t, \vec{i}, \vec{o_r}) \supset \\ \qquad \Psi^*_F(\vec{x}, \textsc{finished}(srv, id, t, \vec{i}, ExOut(id, t, s)), s)] \vee \\ \quad [a \notin \{\textsc{align}, \textsc{finished}\} \supset F_{\exp}(\vec{x}, s)]. \end{array} \qquad (2)$$

where $\Psi^*_F(\vec{x}, a, s)$ is obtained by replacing every fluent X mentioned in $\Psi_F(\vec{x}, a, s)$ (the right-hand-side formula of F's successor state axioms) with its expected version $X_{\exp}$.

The first disjunct states that the special action ALIGN, whose execution is always possible, assigns the actual value of the fluent to its expected value, thus providing a synchronization mechanism between the expected and physical realities.

The second condition states that the expected value of the fluent is the value that the fluent would get if all tasks executed so far since the last alignment point end with their expected output (denoted with term $ExOut(id, t, s)$). Basically, when a FINISHED action is invoked, the F's successor state axiom is used with the expected outcome in place of the real outcome (i.e., $\vec{o_r}$ is replaced with $ExOut(id, t, s)$).

Finally, the third condition states that for all other cases, the expected fluent keeps the value it had before (i.e., inertia law is applicable).

Similarly, for each abbreviation $A(s) \stackrel{\text{def}}{=} \Psi(s)$, an extra abbreviation $A_{\text{exp}}(s)$ is defined as $A_{\text{exp}}(s) \stackrel{\text{def}}{=} \Psi^*(s)$, where $\Psi^*(s)$ is obtained by replacing each fluent (or abbreviation) X in $\Psi(s)$ with its expected version X_{exp}.

Observe that the value of the expected version of a fluent is predicated on the assumption that exogenous action FINISHED will carry the expected output of the task of concern. This may indeed deviate from the actual output, thus yielding a mismatch between physical and expected reality.

Therefore, given a physical reality, an expected reality can be seen as the collection of values of the expected versions of data fluents and abbreviations.

Example 5.5. Upon (syntactic and semantic) simplification, the expected version for fluent $At_{\text{exp}}(srv,s)$ is as follows:

$$
\begin{array}{l}
At_{\text{exp}}(srv, do(a,s)) = l \equiv \\
\quad a = \text{ALIGN} \supset At(srv,s) \vee \\
\quad [(\exists id, \vec{i}, t, \vec{o}_r) a = \text{FINISHED}(srv, id, t, \vec{i}, \vec{o}_r) \supset \\
\qquad t \in \{\text{GO}, \text{MOVE}\} \wedge (Actor(srv) \vee MovingRobot(srv)) \supset \\
\qquad\quad l = ExOut(id, t, s))] \vee \\
\quad [a \neq \text{ALIGN} \wedge \neg(\exists id, \vec{i}, t, \vec{o}_r) a = \text{FINISHED}(srv, id, t, \vec{i}, \vec{o}_r) \wedge \\
\qquad t \in \{\text{GO}, \text{MOVE}\} \supset At_{\text{exp}}(srv,s) = l].
\end{array}
$$

For abbreviation $atLeastOne(s)$, however, we want to force the constraint/expectation that at least one actor is always located in the warehouse, and hence we simply take $atLeastOne_{\text{exp}}(s) \stackrel{\text{def}}{=} \texttt{true}$. □

Next, using the data fluents and their expected versions, a misalignment can be recognized and a recovery procedure may be needed. However, it may only be important to check for mismatches among some properties of the world. So, a data fluent (or abbreviation) is considered *relevant* by the process designer if its evolution should be monitored during process enactment. We assume then that the designer specifies abbreviation $Misaligned(s)$ to characterize misalignment situations that would require process adaptation. The general form of such an abbreviation is as follows:

$$
\begin{array}{l}
Misaligned(s) \stackrel{\text{def}}{=} \\
\qquad \exists \vec{x}_1 . \Phi_{F^1}(\vec{x}_1, s) \supset \neg[F^1(\vec{x}_1, s) \equiv F^1_{\text{exp}}(\vec{x}_1, s)] \vee \\
\qquad\qquad \vdots \\
\qquad \exists \vec{x}_n . \Phi_{F^n}(\vec{x}_n, s) \supset \neg[F^n(\vec{x}_n, s) \equiv F^n_{\text{exp}}(\vec{x}_n, s)],
\end{array}
$$

where $F^i(\vec{x}_i, s)$, with $i \in \{1, \ldots, n\}$, are all the data fluents and abbreviations used in the SmartPM application. Each condition $\Phi_{F^i}(\vec{x}_i, s)$ states the conditions under which data fluent $F^i(\vec{x}_i, s)$ is relevant and needs to be traced for "misalignment."

Example 5.6. In our case study, we are interested, among other things, in monitoring the correct location of human actors, the good "status" of ceramic elements under transformation and the presence in any moment of at least one actor in the warehouse. Technically, we model that as follows:

$$
\begin{array}{l}
Misaligned(s) \stackrel{\text{def}}{=} \\
\quad \exists x_1 . Actor(x_1) \supset \neg[At(x_1, s) \equiv At_{\text{exp}}(x_1, s)] \vee \\
\quad \exists ob_1 . CeramicElement(ob_1) \supset \\
\qquad \neg[Status(ob_1, s) \equiv Status_{\text{exp}}(ob_1, s)] \vee \\
\quad \neg[atLeastOne(s) \equiv atLeastOne_{\text{exp}}(s)] \vee \\
\qquad\quad \vdots
\end{array}
$$

Observe the definition is not concerned about exceptions on the location of moving robots, for example. □

This concludes the explanation on what type of situation calculus BAT we shall use in a SmartPM application. Let us call such a theory $\mathcal{D}_{\text{SmartPM}}$.

5.2. SmartPM High-Level Program

A SmartPM application involves the *online execution* of IndiGolog program $(\text{SmartPM} \| \delta_{exog})$ modeling the concurrent execution of the specific application with special program $\delta_{exog} = (\pi a. Exog(a)?; a)^*$ accounting for all potential exogenous events that may arise from the external environment. Algorithm 1 shows a fragment of the IndiGolog program for a SmartPM application. The program, as any high-level program, is meant to be executed relative to a $\mathcal{D}_{\text{SmartPM}}$ BAT as developed above, which shall give meaning to conditions and primitive statements in the program (i.e., actions). We note that the only domain-dependent part in Algorithm 1 is procedure Process–all other procedures remain unchanged across applications.

The top-level part of the CPMS involves five interrupts running at different priorities, as long as the domain process is not yet finished. The highest three priority programs deal with automated process adaptation; the fourth deals with actual process execution; and the last one forces the system to wait for further changes.

First, if the system has just been adapted, then the two realities—expected and actual—must be aligned, as a new repair plan has been found and a new synchronization point has been reached.

Second, the system checks for a misalignment between the actual reality and the expected one, as explained above. If a mismatch is recognized, process adaptation is initiated by calling procedure Adapt (see subsection 5.3 below).

The third interrupt triggers whenever there is a misalignment but the adaptation procedure (in the second priority interrupt) was not able to find a successful plan to repair such misalignment. In that case, the whole execution waits, for some exogenous events that can allow the system to adapt. Though outside the scope of this paper, another possibility in such cases would be to resort to alternative orthogonal adaptation techniques, such as planning from first-principles (as done in [68]) or just require human intervention.

Whenever there is no adaptation process in place, the CPMS runs the IndiGolog program reflecting the actual process (fourth interrupt), by executing procedure Process. Recall that the actual CPP (cf.Fig. 2(a)) is indeed modelled as yet another IndiGolog program. Managing the life-cycle of a task instance—procedure ManageExec—involves selecting a free service capable of carrying it out, assigning the task to the chosen service, allowing the start of the service, acknowledging its completion and fully releasing the service from the task. Note the use of the non-deterministic choice of argument $\pi srv.\delta(srv)$ to select (any) appropriate service, that is, one capable and free to be assigned the task. Also, the sub-program $\pi id.\text{GETID}(id)$ selects a “fresh” id, that is, an id not used so far, that will serve as the process id for the remaining part of the program ManageExec so that for each task-related action execution there is a unique id. To keep track of fresh identifiers, a fluent $FreshId(id)$ is used. The execution of $\text{GETID}(id)$ has the effect of setting $FreshId(id)$ to false. As expected, the action can only be executed on fresh ids, that is:

$$Poss(\text{GETID}(id), s) \equiv Fresh(id,s).$$

Finally, at the lowest priority (when the process cannot advance) the CPMS just waits for an external action to arrive from one of the services, e.g., a FINISHED action signaling the completion of a running task. We note that, while waiting, the (human) process designer could also manually intervene (for example, by adding new services or updating the capabilities of existing services).

```
Proc SmartPM
  $\langle \neg Finished \wedge MustAlign \rightarrow \text{ALIGN} \rangle \rangle\rangle$
  $\langle \neg Finished \wedge Misaligned \rightarrow \text{Adapt} \rangle \rangle\rangle$
  $\langle \neg Finished \wedge Misaligned \rightarrow \text{WAIT} \rangle \rangle\rangle$
  $\langle \neg Finished \rightarrow \text{Process; FINISH} \rangle \rangle\rangle$
  $\langle \neg Finished \rightarrow \text{WAIT} \rangle.$

Proc Adapt
  $\Sigma[\text{SETMUSTALIGN}; (\pi a.a)^*; \neg Misaligned?];$

Proc ManageExec$(Task, Input, ExpOut)$
  $(\pi\, id).$
    $\text{GETID}(id);$
    $(\pi\, srv).$
      $(Capable(srv, Task) \wedge Free(srv))?;$
      $\text{ASSIGN}(srv, id, Task, Input, ExpOut);$
      $\text{START}(srv, id, Task);$
      $\text{ACKCOMPL}(srv, id, task);$
      $\text{RELEASE}(srv, id, task).$

Proc Process
  Seq-MultiInstance-1;
  Conc-MultiInstance-2;
  Conc-MultiInstance-3;
            ⋮
  Seq-MultiInstance-13.

Proc Seq-MultiInstance-1
  for each $x$ in $CeramicElement(x) \wedge Status(x) = ok$ do$^*$
    ManageExec$(\text{CONVEY}, [x, loc\_start,$
          $loc\_rotomoulding], [loc\_rotomoulding])$
  endFor
Proc Conc-MultiInstance-2
  for each $x$ in $CeramicElement(x) \wedge Status(x) = ok$ do$^{\|}$
    ManageExec$(\text{MOULD}, [x, loc\_rotomoulding], [ok])$
  endFor
Proc Conc-MultiInstance-3
  for each $x$ in $CeramicElement(x) \wedge Status(x) = ok$ do$^{\|}$

ManageExec$(\text{CHECKQUALITY}, [x, loc\_rotomoulding], [ok])$
  endFor
  .....
Proc Seq-MultiInstance-13
  for each $x$ in $CeramicElement(x) \wedge Status(x) = ok$ do$^*$
    ManageExec$(\text{CONVEY}, [x, loc\_firing,$
            $loc\_end], [loc\_end])$
  endFor
```

Algorithm 1. IndiGolog high-level program for CPMS.

5.2.1. *Multi-instance Processes*

When modeling a business process, a commonly used control flow mechanism is *multi-instance* execution of sub-processes. Intuitively, the idea is to spawn the execution of the same process on each object in a given set. For example, in our domain, a process may instruct every idle robot to navigate to a given location. There are two versions that are commonly used. In the *sequential* case, a given subprocess is executed on each object one at a time. In turn, in the *non-sequential* case, multiple instances of the subprocess, one per relevant object, are concurrently executed. More concretely, if

$o_1, o_2, \cdots, o_n$ are the objects of interest (e.g., all currently idle robots) and $\delta(x)$ is the subprocess to be carried out (e.g., navigate to emergency location), a sequential multi-instance execution would amount to executing, for example, program $\delta(o_1); \delta(o_2); \cdots; \delta(o_n)$, whereas the non-sequential version would amount to any execution of program $\delta(o_1) \| \delta(o_2) \| \cdots \| \delta(o_n)$.

Unfortunately, while common in business processes languages, this control flow construct is not provided by any of the languages in the Golog family. It turns out, however, that we can realize such control flow by combining existing constructs together with extra bookkeeping information in the underlying action theory. Concretely, we first introduce a new construct defined as follows for the sequential version:

$$\begin{array}{l} \textbf{for each } x \textbf{ in } \phi(\vec{x}) \textbf{ do}^{*}\ \delta(\vec{x}) \textbf{ endFor} \overset{\text{def}}{=} \\ \quad \pi id.\text{STORE}_{\phi}(id); \\ \qquad [\pi \vec{x}.\text{REMOVE}_{\phi}(\vec{x}, id);\ \delta(\vec{x})]^{*}; \\ \qquad ?(\neg \exists \vec{x} M_{\phi}(id, \vec{x})) \end{array}$$

Here, action $\text{STORE}_{\phi}(id)$ is used to "save" every instance for which $\phi(\vec{x})$ holds true into a distinguished fluent $M_{\phi}(id, \vec{x})$. By doing this, we are able to then execute process δ on each of those instances, regardless of how formula $\phi(\vec{x})$ changes its truth value as actions are performed. The *id*'s is used in the auxiliary fluent M_{ϕ} to make sure that different instances of the same for-loop can be executed without clashing. To achieve that, the preconditions of the auxiliary actions are as follows:

$$\begin{array}{l} Poss(\text{STORE}_{\phi}(id), s) \equiv FreshId(id, s); \\ Poss(\text{REMOVE}_{\phi}(id, \vec{x}), s) \equiv M_{\phi}(id, \vec{x}, s). \end{array}$$

As with action $\text{GETID}(id)$, the SSA for $FreshId(id)$ encodes the negative effect of action $\text{STORE}_{\phi}(id)$ on id object *id* (i.e., *id* is not available anymore), and the SSA for fluent $M_{\phi}(id, \vec{x}, s)$ encodes the negative effect of action $\text{REMOVE}_{\phi}(id, \vec{x})$ on $(id, \vec{x})$ (i.e., object $\vec{x}$ in for-each process *id* has been processed).

Finally, the non-sequential version can be defined as follows:

$$\begin{array}{l} \textbf{for each } x \textbf{ in } \phi(\vec{x}) \textbf{ do}^{\|}\ \delta(\vec{x}) \textbf{ endFor} \overset{\text{def}}{=} \\ \quad \pi id.\text{STORE}_{\phi}(id); \\ \qquad [\pi \vec{x}.\text{REMOVE}_{\phi}(\vec{x}, id);\ \delta(\vec{x})]^{\|}; \\ \qquad ?(\neg \exists \vec{x} M_{\phi}(id, \vec{x})) \end{array}$$

The last test step in the program is used to enforce that any execution should process every (complex) object $\vec{t}$ for which $\phi(\vec{x})$ holds (i.e., any tuple $\vec{t}$ that has been stored into fluent M_{ϕ}).

5.3. CPP Adaptation via Automated Planning

The most interesting part of the procedure involves procedure Adapt. An adaptation allows to *find a sequence of actions that will resolve the misalignment.* This is exactly what the code inside the search operator Σ does: *pick and execute actions zero, one, or more times such that abbreviation Misaligned(s) becomes false.* The action SETMUSTALIGN at the front of the search construct will just make $MustAlign(s)$ true, which will trigger (provided an actual adaptation plan is found) the top-priority program in the main procedure, where the action ALIGN (whose execution is always possible) forces an alignment of the two realities, thus providing a synchronization mechanism between them. Observe that because the adaptation mechanism runs at higher priority than the actual process, the recovery plan found will be run *before* whatever part of the domain process remains to be executed.

Putting it all together, let us formally capture the type of adaptation realized by our approach.

Definition 5.1. Let S be a ground situation term such that $\mathcal{D}_{\text{SmartPM}} \models Misaligned(S)$. We say that situation S is *recoverable* if and only if it is the case that $\mathcal{D}_{\text{SmartPM}} \models \exists s'. S \sqsubset s' \land Executable(s') \land \neg Misaligned(s')$.

That is, a given situation can be recovered if there is an executable sequence of actions from it that will eventually resolve the misalignment between the physical and expected realities. Specifically, when the execution reaches a misalignment, the SmartPM procedure *(i)* will not execute the main process (encoded in procedure Process and running at the fourth priority level); and *(ii)* will execute the first action of a plan to recover the current situation from misalignment, if such a plan exists (otherwise, the system just waits; see above).

Importantly, the above result also makes explicit the limits of our adaptation framework: if there is no plan able to resolve the existing mismatch, the system just waits (third point) and other orthogonal adaptation techniques would need to be used, see discussion in Section 8.

While this specification of the automated adaptation procedure turns out to be extremely clean and simple, the direct use of the native search operator provided by the IndiGolog architecture [14] poses serious problems in terms of efficiency. The fact is that the search operator provided by IndiGolog amounts to a generic and incremental (i.e., done at every step) search; as a result, direct implementations are not able to cope with even extremely easy adaptation tasks.

But, while the search operator is meant to handle *any* IndiGolog high-level program (including ones containing nested search operators), our SmartPM system uses a *specific* type of program that turns out to encode a (classical) planning problem. So, leveraging on the recent progress of classical planning systems, we implement the search operator call in procedure Adapt, by using an off-the-shelf planner to synthesise the recovery plan, and then fit such a plan back into the IndiGolog framework. Specifically, we used the LPG-td system [29], one of the many state-of-the-art planning systems available. The basic search scheme of LPG-td is inspired by Walksat [65], an efficient procedure for solving SAT-problems; as expected, it outperforms the blind search operator by several orders of magnitudes (cf. [42]). Nonetheless, our approach is orthogonal to other planning systems.

Since the integration of PDDL planning with situation calculus and Golog-like languages has been already analysed in several research works, such as [12, 11,25], we just go over the main ingredients on how we realized the interface between IndiGolog and LPG-td.

First of all, given a BAT $\mathcal{D}_{\mathsf{SmartPM}}$ for a CPP application, the corresponding PDDL planning domain is built (and stored) offline. Because we are not concerned with the external actions generated by services to acknowledge the start and termination of assigned processes (i.e., actions READYTOSTART and FINISHED), we do not model the full assign-start-acknowledge-release task life-cycle, but just encapsulates them all in the actual name of the task being handled, e.g., GO. By doing that, we assume that the life-cycle of a task instance will follow its *expected* evolution. The SmartPM BAT will define a form of tasks and services repository, which may include entities not used in the current running process. So, the PDDL domain for our case study will contain, among others, the following action schema modeling the GO task:[5]

```
(:action go
 :parameters (?srv - actor ?from - location
              ?to - location)
 :precondition
       (and (provides ?srv movement) (free ?srv)
            (at ?srv ?from) (atLeastAnotherOne))
 :effect (and (not (at ?srv ?from))
              (at ?srv ?to)))
```

[5]In PDDL, the variables are distinguished by a '?' character at front, for example $?x1$ represents a variable. The dash '–' is used to assign types to the variables.

Note that the task-action in the planning system will contain the service in charge and its expected effects.

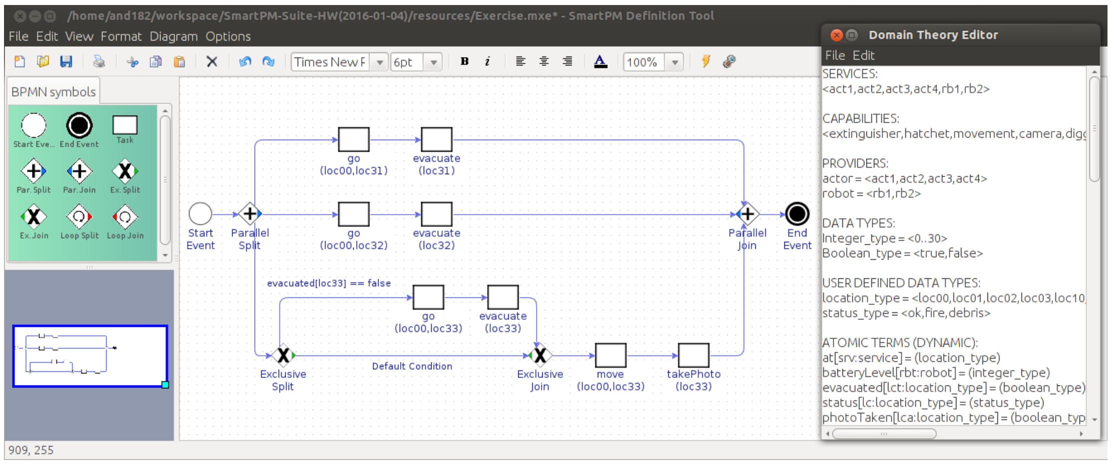
In this work we represent planning domains and problems making use of the STRIPS fragment of PDDL 2.2, enhanced with derived predicates and numeric features provided by the level 2 of the same language. Specifically, derived predicates are employed to encode all the required user-defined abbreviations (for example, see the derived predicate `atLeastAnotherOne` used in the preconditions of the planning action `go`), whereas numeric features are used to model non-binary resources (such as, for example, the battery level of a robot), to keep track of the costs of planning actions and to synthesize plans satisfying pre-specified metrics.

Whenever the adaptation program is called in procedure Adapt, the current physical reality is encoded as the planning problem *initial state* (as the set of fluents that are true) and the expected reality is encoded as the problem *goal state* (by taking the collection of relevant fluents to be as their expected versions). Those two states, together with the planning domain already pre-computed, are then passed to the LPG-td system.

Finally, if the planner finds a plan that brings about the (desired) expected reality, such a plan—built from task names only—is translated into the typical assign-start-acknowledge-release task life-cycle IndiGolog program. As stated above, such a plan will run *before* the actual domain process, which shall resume then, hopefully from the expected reality. In our running example, the full IndiGolog program would encode the CPP depicted in Fig. 2(a).

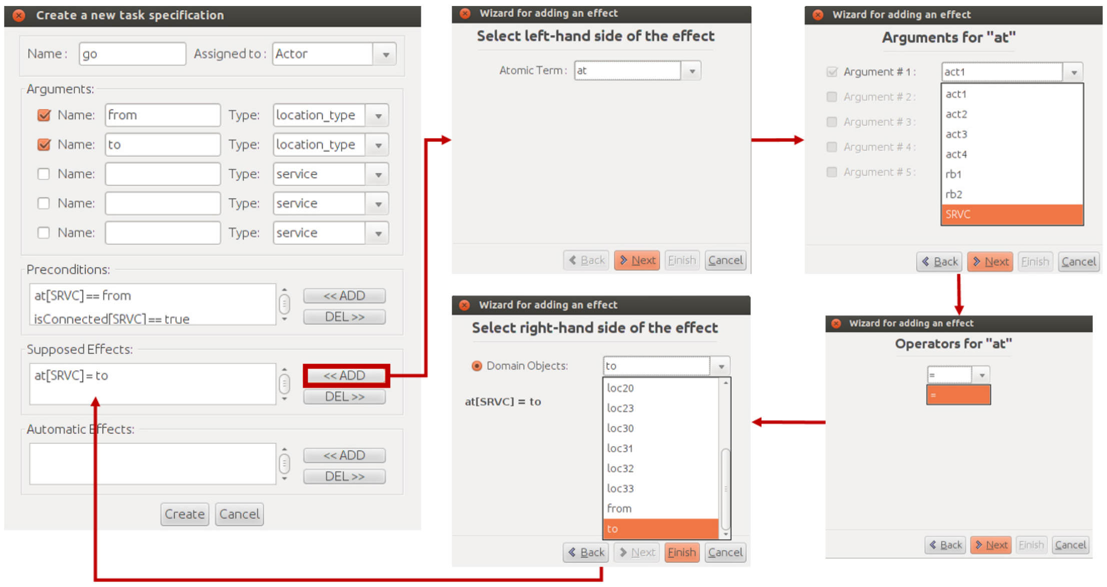
6. Evaluating SmartPM

While the framework presented in Section 5 focuses on the formalization of the approach through action-based formalisms (basically, we covered the *enactment* and *adaptation* layers of the SmartPM architecture), in this section we aim at providing some technical details on the *design* and *service* layers, which consist of tools and plugins that allow human process designers to concretely interact with the SmartPM system (see Section 6.1). Then, we show the results of some experiments performed with real users to investigate the practical applicability of the SmartPM system to model real-world CPPs (see Section 6.2).



```
SERVICES:
<act1,act2,act3,act4,rb1,rb2>

CAPABILITIES:
<extinguisher,hatchet,movement,camera,digg

PROVIDERS:
actor = <act1,act2,act3,act4>
robot = <rb1,rb2>

DATA TYPES:
Integer_type = <0..30>
Boolean_type = <true,false>

USER DEFINED DATA TYPES:
location_type = <loc00,loc01,loc02,loc03,loc10,
status_type = <ok,fire,debris>

ATOMIC TERMS (DYNAMIC):
at[srv:service] = (location_type)
batteryLevel[rbt:robot] = (integer_type)
evacuated[lct:location_type] = (boolean_type)
status[lc:location_type] = (status_type)
photoTaken[lca:location_type] = (boolean_typ
```

(a) The workspace provided by the SmartPM Definition Tool.


Create a new task specification
Wizard for adding an effect
Select left-hand side of the effect
Arguments for "at"
Select right-hand side of the effect
Operators for "at"
<< ADD
DEL >>
Create
Cancel
Back
Next
Finish
Cancel

Name: go Assigned to: Actor

Arguments:

✓	Name: from	Type: location_type
✓	Name: to	Type: location_type
☐	Name:	Type: service
☐	Name:	Type: service
☐	Name:	Type: service

Preconditions:
at[SRVC] == from
isConnected[SRVC] == true

Supposed Effects:
at[SRVC] = to

Automatic Effects:

Atomic Term: at

Argument # 1: act1
Argument # 2:
Argument # 3:
Argument # 4:
Argument # 5:

act1
act2
act3
act4
rb1
rb2
SRVC

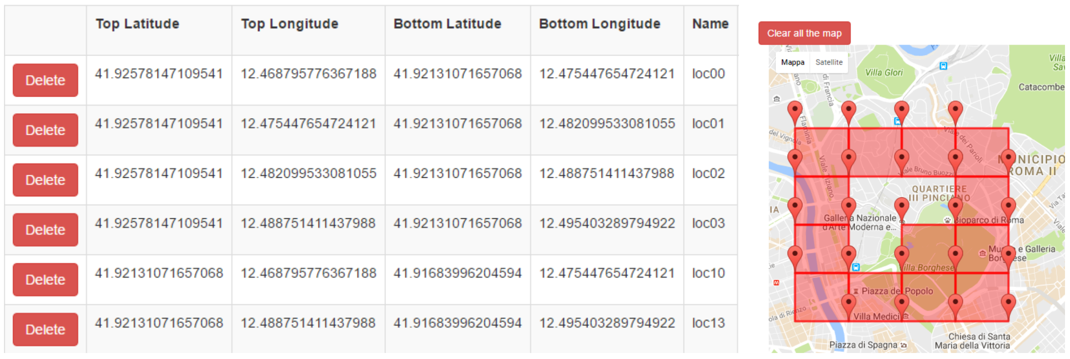
=

Domain Objects: to

at[SRVC] = to

loc20
loc23
loc30
loc31
loc32
loc33
from
to

(b) The wizard-based editor to build tasks specifications.



	Top Latitude	Top Longitude	Bottom Latitude	Bottom Longitude	Name
Delete	41.92578147109541	12.468795776367188	41.92131071657068	12.475447654724121	loc00
Delete	41.92578147109541	12.475447654724121	41.92131071657068	12.482099533081055	loc01
Delete	41.92578147109541	12.482099533081055	41.92131071657068	12.488751411437988	loc02
Delete	41.92578147109541	12.488751411437988	41.92131071657068	12.495403289794922	loc03
Delete	41.92131071657068	12.468795776367188	41.91683996204594	12.475447654724121	loc10
Delete	41.92131071657068	12.488751411437988	41.91683996204594	12.495403289794922	loc13

(c) The location web tool to mark areas of interest.

Fig. 5. Some screenshots of the SmartPM Definition Tool.

6.1. *Interacting with the SmartPM System*

One of the main obstacles in applying AI techniques to real problems is the difficulty to model the complexity of real-world domains. Let us take, for example, our SmartPM approach. It is evident that it would be extremely complex for a process designer to encode a (even simple) CPP in SmartPM by using directly our framework based on action-based formalisms.

In the SmartPM system, we have tackled this issue by developing a GUI-based tool in the range of the design layer, which is called the SmartPM Definition Tool. It supports the process design activity by providing *(i)* several wizard-based editors that assist the process designer in the definition of the process knowledge (i.e., data objects, atomic terms, tasks with preconditions and effects, etc.), and *(ii)* a graphical editor to design the control flow of a CPP using a relevant subset of the BPMN 2.0 notation. The SmartPM Definition Tool has been developed using the Java SE 7 Platform, and the JGraphX open source graphical library.[6] While a screenshot of the workspace of the SmartPM Definition Tool is provided in Fig. 5(a), in Fig. 5(b) we show one of the wizard-based editor provided by SmartPM, specifically the one to build a task specification by defining the single conditions composing the task preconditions and effects.

The SmartPM Definition Tool allows also to make explicit the connection of implemented processes with the real-world objects of the cyber-physical environment of interest. For example, we developed a web tool that allows a process designer to mark areas of interest from a real map (by selecting latidude/longitude values) and associate them to the discrete locations (e.g., *loc00*, *loc01*, etc.) defined during the design stage of a process. Fig. 5(c) shows a screenshot of the location web tool. Similarly, we developed further web tools for the other developed sensors (temperature, humidity, noise level, etc.).

The interaction between process participants and the SmartPM system during the enactment of a CPP is performed through a Task Handler that supports the visualization of assigned tasks and enables starting task execution and notifying of task completion by selecting an appropriate outcome. The SmartPM Task Handler is realized for Android devices from version 4.0 and up. Each device has an unique ID that matches the service name defined in the domain theory by the designer. A screenshot of the Task Handler is shown in Fig. 6.

[6] http://www.jgraph.com/

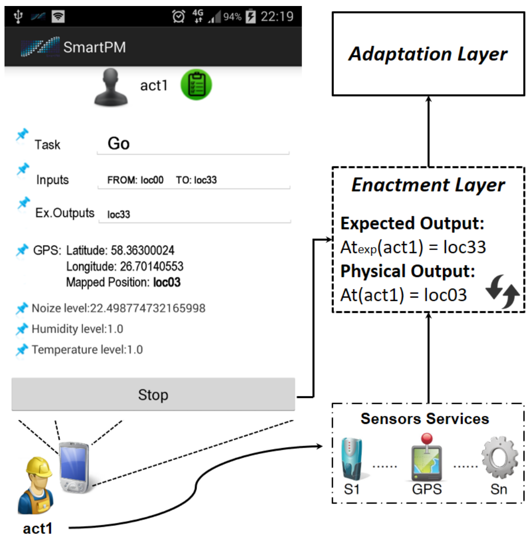

SmartPM
act1
Task Go
Inputs FROM: loc00 TO: loc33
Ex.Outputs loc33
GPS: Latitude: 58.36300024
Longitude: 26.70140553
Mapped Position: loc03
Noize level:22.498774732165998
Humidity level:1.0
Temperature level:1.0
Stop

Fig. 6. Screenshot of the Task Handler of SmartPM. In the figure, it is described how the data fluent *At*(*act1*) can be used to represent the effect of the task GO. We show that the output value for *At*(*act1*) (in the example '*loc03*', different from the task's expected outcome, that is '*loc33*') can be produced by a sensor (i.e., a GPS device) supporting the Task Handler.

We created several plugins for the Task Handler to obtain, for example, location data using built-in GPS sensors or get the current noise level near the device using its microphone. In addition, external sensors can be taken into use to gather automatic measurements - for prototyping purposes, the Arduino platform[7] can be used. The Task Handler can take advantage of this technology for gathering environmental data. In fact, Arduino has a large variety of sensors available to measure different environmental values, for example different gas levels in the air, water quality, radiation level, etc. Arduino can be connected with Android via Bluetooth for transferring the data.

6.2. *Investigating the practical applicability of the SmartPM system*

The SmartPM System has been extensively validated in our previous work [42] through empirical experiments based on 3600 different process models having control flows with different structures and domain theories associated to them. On the one hand, the experiments have confirmed the feasibility of the

[7] Arduino is an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board, cf. http://arduino.cc/en/guide/introduction

planning-based approach of SmartPM for adapting processes in medium-sized cyber-physical environments from the timing performance perspective. On the other hand, SmartPM was able to complete 2537 process instances without any domain expert intervention, corresponding to an effectiveness of about 70,5% (see [42] for a discussion of such results).

Conversely, in this paper we focused our attention on evaluating if the SmartPM system is *easy to be learnt and used* from non expert users in AI. The final purpose was to demonstrate that the modeling of CPPs at design-time and their execution with SmartPM does not require any specific expertise of the internal working of the AI tools involved in the system. Therefore, we performed experiments to assess the *learnability* of the SmartPM Definition Tool.

In the Human-Computer Interaction community, learnability is recognized as one of the most relevant components of usability [18]. Learnability is addressed in ISO 9126-1 as the *capability of the software product to enable the user to learn its application* and applies to the nature of the performance change of a user when interacting with a system. The more learnable a system is, the less time a user takes in order to understand how to do a specific task without having been previously trained and without using any documentation.

To measure the learnability of the SmartPM Definition Tool, we leveraged on the approach developed in [32], which allows to identify quantitatively *how much* the user actions that take place during a run of the system for achieving a specific objective (for example, the creation of a new data object) *deviate* from the *expected way* of achieving the same objective as foreseen by the system. The *weight* of such deviations is quantified through a *fitness value*, which estimates how much the actions performed by the users adhere to the expected way of using the system. The fitness value can vary from 0 to 1. A value equals to 1 means that the user was able to achieve perfectly her/his objective as foreseen by the system. According to [32], the learnability of the system can be estimated by analyzing the rate of the fitness values corresponding to subsequent executions of the system over time. An increasing rate will correspond to a system that is *easy to be learnt*.

To assess the learnability of SmartPM against the complexity of realistic cyber-physical environments, we performed a usability tests with 23 Master students in Engineering in Computer Science during the university course of *Seminars in Software and Services for the Information Society*[8] (Academic Year 2015-2016), held at Sapienza - University of Rome. After a preliminary training session to describe the usage of the SmartPM Definition Tool, we provided to the students 4 different homeworks of growing complexity in 4 consecutive weeks (one homework per week); each homework was targeted to model the domain theory and the control flow of a realistic CPP through the SmartPM Definition Tool. The students were requested to complete the homework assigned to them in a specific week within the end of the week itself. Before the assignment of a new homework, we shown to the students the optimal solution to perform correctly the previous homework.

To apply the approach described in [32], we recorded in a specific log all the user actions performed by the students during their interaction with the SmartPM Definition Tool. Then, to assess the learnability of the system, we replayed such logs over three *interaction models* of the system describing the expected ways to achieve three relevant objectives: *(i)* definition of a new data object, *(ii)* generation a new atomic term and *(iii)* creation of a new task specification.

The results of the experiments are provided in Fig. 7. Collected data are organized in 3 diagrams related to the achievement of the three above objectives. For any diagram, the x-axis indicates the specific homework considered, while the y-axis indicates the average fitness value obtained to achieve the specific objective. Notice that for each homework we represent two bars for separating the students that performed correctly the whole homework from the students that were not able to complete it or to compute a correct solution.

The analysis of the performed experiments points out some interesting aspects. For example, let us consider the first diagram in Fig. 7, that shows the fitness values related to the definition of a new data object. First of all, it is evident that both the rate of the fitness values and the number of students able to correctly complete an homework increases over time, i.e., homework after homework. In fact, while 6 students out of 23 were not able to complete the first homework (around 26% of failure), the third and the fourth homework were correctly completed by 20 and 21 students (percentage of success of 86.95% and 91.3%). It is interesting to notice a "discontinuity" of this trend between the first and the second homework, probably caused by the first relevant increase of complexity.

Concerning the fitness value, we can notice that it is always higher for those students that completed correctly the whole homework, even if this gap slightly decreases in the subsequent homeworks.

[8]`https://sites.google.com/a/dis.uniroma1.it/s4i-ss/`

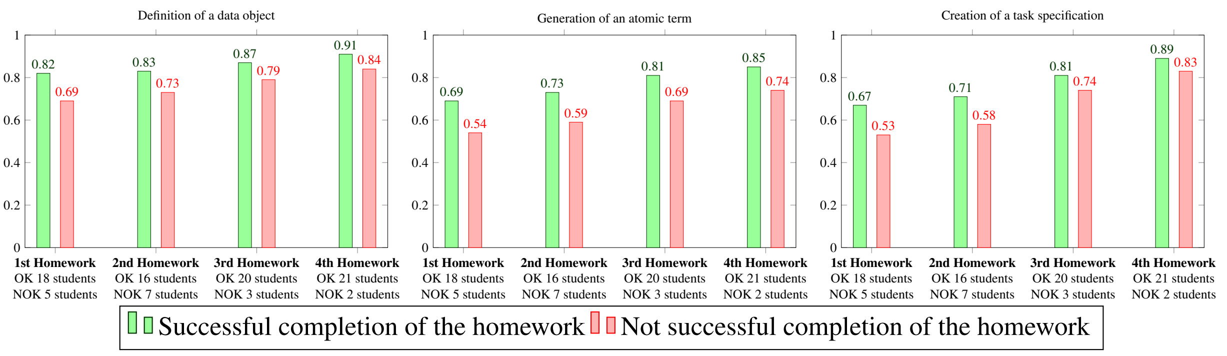


Fig. 7. Analysis of the learnability of the SmartPM Definition Tool.

The above considerations are valid also for the other diagrams, even if the fitness values emphasize that the definition of a data object is a simpler task if compared with the generation of an atomic term or the creation of a task specification. However, also in this case, this “complexity gap” seems to decrease over time.

To sum up, the results obtained allow to observe an increasing rate of the fitness value over time, which indicates that the students performing the homeworks were able to efficiently *learn* the usage of the SmartPM Definition Tool, without any knowledge of the AI technologies employed in the system.

7. Related work

Initial research efforts addressing the need for process adaptation in PMSs can be traced back to the late nineties and early two thousands [7,8,22,23,31,36,41]. Although possible sources of exceptions are different (as outlined in [8,22], they can be attributed to activity failures, deadline expirations, resource unavailabilities, constraint violations and external events) and go beyond technical failures, not surprisingly process adaptation approaches provided by academic prototypes and commercial PMSs trace and resemble exception handling mechanisms in programming languages (e.g., *try-catch* blocks) [1,10,67].

Typical strategies applied when defining exception handlers for *anticipated exceptions* have been systematized in the form of *exception handling patterns* [7,31, 63,40,58], i.e., for any given exception, a predefined explicit handling logic is defined as a sequence of corrective actions to resolve the issue.

Conversely, the handling of *unanticipated exceptions* does not assume the availability of predefined exception handlers and relies on the possibility of performing *ad hoc changes* over process instances at run-time [73,58]. This requires *structural adaptation* of the corresponding process model. As in the case of exception handling, structural adaptation techniques have been systematized through the identification of *adaptation patterns* [71], i.e., predefined change operations for adding, deleting or replacing process activities.

Strong support for structural adaptation is provided by the ADEPT system and its evolutions [55,47,56,57, 37]. However, while a good level of support can be provided to ensure correctness and compliance when structural adaptation is performed, the degree of automation is generally limited to manual ad hoc changes performed by experienced users [61].

In an attempt to increase the level of user support, semi-automated approaches have been proposed [62]. They aim at storing and exploiting available knowledge about previously performed changes, so that users can retrieve and apply it when adapting a process. Such an approach has been concretely put into practice using case-based reasoning techniques [72,45].

When compared with traditional exception handling approaches, we notice that adaptive PMSs deal with unanticipated exceptions by automatically deriving the `try` block as the situation in which the PMS does not adequately reflect the real-world process anymore. The `catch` block is defined *manually* or *semi-automatically* at run-time and includes those recovery procedures required for realigning the computerized processes with the real-world ones. However, in cyber-physical working environments, analyzing and defining these adaptations “manually” becomes time-demanding and error-prone. Indeed, the designer should have a global vision of the application and its context to define appropriate recovery actions, which becomes complicated when the number of relevant context features and their interleaving increases. Conversely, our SmartPM approach is able to *automatically synthesizing at run-time* the `catch` block, without the need of any manual intervention at run-time, and increases the level of "automation" in process adaptation.

The issue of having software systems automatically and autonomously adapt to changing conditions has been also addressed in the last years under the *autonomic* and *self-healing systems* literature [30,52]. In such research contexts, SmartPM can be considered as a specific kind of self-healing system addressing the management of processes through AI-based techniques. In particular, according to what was presented in [52], its main peculiarities are the use of a context- and data-aware process engine and the ability to reason over dynamic contexts, for which the adopted KR&R techniques are particularly suitable.

A further research area related to process adaptation is the one of Web service composition [44,4,50,51,2, 66,15]. Notably, most of the approaches to Web service composition adopt planning-based techniques, as the SmartPM approach does. However, there exists a notable difference in *what* SmartPM and Web service composition techniques synthesize through planning. In Web service composition, given a a set of available services and a target service, the challenge is to synthesize a possible composition (a.k.a. *orchestration*) of the available services by preserving their behaviors (which are in general quite rich, often modeled as transition systems) in order to obtain the target service. Conversely, in the SmartPM approach, the aim is to synthesize a recovery process that *repairs* the original one in a specific situation of the world by using a set of tasks stored in a tasks repository. The main issue here is to preserve as much as possible the original process, considering that process tasks are atomic and do not present rich behaviors to be preserved.

Finally, a number of techniques from the field of AI have been applied to BPM with the aim of increasing the degree of automated process adaptation at run-time. One of the first works dealing with this research challenge is [3], which discusses at high level how the use of an intelligent assistant based on planning techniques may suggest compensation procedures or the re-execution of activities if some anticipated failure arises during the process execution. In [34] the authors describe how planning can be interleaved with process execution and plan refinement, and investigate plan patching and plan repair as means to enhance flexibility and responsiveness.

A goal-based approach for enabling automated process instance change in case of exceptions is shown in [27]. If a task failure occurs at run-time and leads to a process goal violation, a multi-step procedure is activated. It includes the termination of the failed task, the sound suspension of the process, the automatic generation (through the use of a partial-order planner) of a new complete process definition that complies with the process goal and the adequate process resumption. A similar approach is proposed in [24]. The approach is based on learning business activities as planning operators and feeding them to a planner that generates a candidate process model that is able of achieving some business goals. If an activity fails during process execution at run-time, an alternative candidate plan is provided on the same business goals. The major issue of [27,24] lies in the replanning stage used for adapting a faulty process instance. In fact, it forces to completely redefine the process specification at run-time when the process goal changes (due to some activity failure), by completely revolutionizing the work-list of tasks assigned to the process participants (that are often humans). On the contrary, our approach adapts a running process instance by modifying only those parts of the process that need to be changed/adapted and keeps other parts stable.

The works [26] and [46] provide a formalization and a regression-based approach to identify when a given plan, developed for an ultimate specified goal, does not work anymore and replanning may be required. Such works adopt a traditional notion of plan, either sequential [26] or partial-order [46]. Conversely, in SmartPM plans are seen as recovery processes and we do not assume the presence of pre-specified goals associated to the main process under execution. This makes the regression-based approach of [26] and [46] not directly applicable to our approach.

In the work [6] the authors propose a goal-driven approach for service-based applications to automatically adapt business processes to run-time context changes. Process models include service annotations describing how services contribute to the intended goal. Contextual properties are modeled as state transition systems capturing possible values and evolutions in the case of precondition violations or external events. Process and context evolution are continuously monitored and context changes that prevent goal achievement are managed through an adaptation mechanism based on service composition via automated planning techniques. However, this work requires that the process designer explicitly defines the policies for detecting the exceptions at design-time, while in SmartPM the recovery procedure is synthesized at run-time, without the need to define any recovery policy at design-time.

A work dealing with process interference is that of [68]. Process interference is a situation that happens when several concurrent business processes depending on some common data are executed in a highly distributed environment. During the processes execution, it may happen that some of these data are modified causing unanticipated or wrong business outcomes. To overcome this limitation, the work [68] proposes a run-time mechanism which uses *(i) Dependency Scopes* for identifying critical parts of the processes whose correct execution depends on some shared variables; and *(ii) Intervention Processes* for solving the potential inconsistencies generated from the interference, which are automatically synthesised through a domain independent planner based on CSP techniques. While closely related to van Beest's work, our account deals with changes in a more abstract and domain-independent way, by just checking misalignment between expected/physical realities. Conversely, van Beest's work requires specification of a (domain-dependent) adaptation policy, based on volatile variables and when changes to them become relevant.

8. Concluding Remarks

We are at the beginning of a profound transformation of BPM due to advances in AI and Cognitive Computing [33]. Cognitive systems offer computational capabilities typically based on large amount of data, which provide cognition power that augment and scale human expertise. The aim of the emergent field of cognitive BPM is to offer the computational capability of a cognitive system to provide analytical support for processes over structured and unstructured information sources. The target is to provide proactivity and self-adaptation of the running processes against the evolving conditions of the application domains in which they are enacted.

In this direction, our paper has been devoted to define a general approach, a concrete framework and a CPMS implementation, called SmartPM, for automated adaptation of CPPs. Our purpose was to demonstrate that the combination of procedural and imperative models with cognitive BPM constructs such as data-driven activities and declarative elements, along with the exploitation of techniques from the field of AI such as situation calculus, IndiGolog and classical planning, can increase the ability of existing PMSs of supporting and adapting CPPs in case of unanticipated exceptions.

Existing approaches dealing with unanticipated exceptions typically rely on the involvement of process participants at run-time, so that authorized users are allowed to manually perform structural process model adaptation and ad-hoc changes at the instance level. However, CPPs demand a more flexible approach recognizing the fact that in real-world environments process models quickly become outdated and hence require closer interweaving of modeling and execution. To this end, the adaptation mechanism provided by SmartPM is based on execution monitoring for detecting failures and context changes at run-time, without requiring to predefine any specific adaptation policy or exception handler at design-time (as most of the current approaches do).

From a general perspective, our planning-based automated exception handling approach should be considered as complementary with respect to existing techniques, acting as a “bridge” between approaches dealing with anticipated exceptions and approaches dealing with unanticipated exceptions. When an exception is detected, the run-time engine may first check the availability of a predefined exception handler, and if no handler was defined it can rely on an automated synthesis of the recovery process. In the case that our planning-based approach fails in synthesizing a suitable handler (or an handler is generated but its execution does not solve the exception), other adaptation techniques need to be used. For example, if the running process provides a well-defined intended goal associated to its execution, we could resort to the van Beest's work [68] and do planning from first-principle to achieve such a goal. Conversely, if no intended goal is associated to the process, a human participant can be involved, leaving her/him the task of manually adapting the process instance.

We notice that the SmartPM approach is predicated on two (strong) assumptions:

- postconditions or intended goals of programs are not explicitly available/specified. This comes from the traditional BPM domain where all the focus is on the structure of the business process. While the process itself intends to achieve some goal, this remains implicit and non-specified. Moreover, the process specified generally encodes non-functional requirements that go beyond the goal being achieved, which means that the goal needs to be achieved by executing the process as specified (cf. [19]);

– the exogenous events are always considered as "dangerous". This also comes from the BPM domain, where what is expected is modeled in the process and everything else is considered "dangerous" (or erroneous). Of course, there exist real world cases where an exogenous event could be "helpful" for the final goal.

However, the fact that the SmartPM approach relies on well founded KR&R formalisms opens the door for many advanced reasoning tasks upon failure and amounts to very promising future work, e.g., to associate goals with processes and subprocesses and checks that such goals are fulfilled, to exploit "helpful" exogenous events that make smarter the synthesis of recovery procedures, or to investigate what parts of the process can not be repaired or abduce what has gone wrong in the past, in order to assists the user in the manual definition of the recovery plan.

Future work will include an extension of our approach to "stress" the above assumptions and all those one imposed by the usage of automated classical planning techniques for the synthesis of the recovery procedure, which frame the scope of applicability of the approach for addressing more expressive problems, including incomplete information, preferences and multiple task effects.

We also notice that, even if the SmartPM approach is able to adapt a process instance at run-time, it does not allow either to support hierarchical processes or to evolve the original process model on the basis of exceptions captured. Therefore, a second future direction of this work is to provide support for executing hierarchical processes, with high-level processes achieving more general goals that can invoke simpler processes to achieve some of their subgoals. We argue that agent-technology (for example, BDI [54], which stands for "beliefe-desire-intensions") and hierarchical planners [48] can provide promising approaches and methods to address this challenge. In addition, a third main future work concerns to avoid to consider all deviations from the process as errors, but as a natural and valuable part of the work activity, which provides the opportunity for learning and thus evolving the process model for future instantiations. Finally, a further interesting future work is to devise a set of design-time guidelines that may help the process designer in choosing what fluents/abbreviations should be (or not be) monitored for misalignment.

The current implementation of SmartPM is developed to be effectively used by process designers and practitioners.[9] Users define processes in the well-known BPMN language, enriched with semantic annotations for expressing properties of tasks, which allow our interpreter to derive the IndiGolog program representing the process. Interfaces with human actors (such as specific graphical user applications in Java) and software services (through Web service technologies) allow the core system to be effectively used for enacting processes. Although the need to explicitly model process execution context and annotate tasks with preconditions and effects may require some extra modeling effort at design-time (also considering that traditional process modeling efforts are often mainly directed to the sole control flow perspective), the overhead is compensated at run-time by the possibility of automating exception handling procedures. While, in general, such modeling effort may seem significant, in practice it is comparable to the effort needed to encode the adaptation logic using alternative methodologies like happens, for example, in rule-based approaches.

Acknowledgements

This work has been partly supported over the years by the following projects: EU FP-6 WORKPAD, EU FP-7 SM4All, Italian Sapienza grants DAKIP, TESTMED and SUPER, Italian Sapienza award SPIRITLETS, Italian projects Social Museum and Smart Tourism (CTN01_00034_23154), NEPTIS (PON03PE_00214_3), and RoMA - Resilence of Metropolitan Areas (SCN_00064). The authors would like to thanks the many persons involved over the years in the SmartPM conception and development, namely Giuseppe De Giacomo, Massimiliano de Leoni, Patris Halapuu, Arthur H.M. ter Hofstede, Alessandro Russo.

References

[1] M. J. Adams. *Facilitating dynamic flexibility and exception handling for workflows.* PhD thesis, Queensland University of Technology Brisbane, Australia, 2007.

[2] V. Agarwal, G. Chafle, K. Dasgupta, N. M. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *Journal on Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):311–339, 2005.

[9] More information about the system is available at `http://www.dis.uniroma1.it/~smartpm`

[3] C. Beckstein and J. Klausner. A Meta Level Architecture for Workflow Management. *Journal of Integrated Design and Process Science*, 3(1):15–26, 1999.

[4] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proceedings of the First International Conference on Service-Oriented Computing*, ICSOC 2003, pages 43–58. Springer, 2003.

[5] R. Brachman and H. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004.

[6] A. Bucchiarone, M. Pistore, H. Raik, and R. Kazhamiakin. Adaptation of service-based business processes by context-aware replanning. In *Proceedings of the 4th International Conference on Service-Oriented Computing and Applications*, SOCA 2011, pages 1–8. IEEE, 2011.

[7] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems (TODS)*, 24(3):405–451, 1999.

[8] F. Casati and G. Cugola. Error Handling in Process Support Systems. In *Advances in Exception Handling Techniques*, pages 251–270. Springer-Verlag, 2001.

[9] S. Chand and J. Davis. What is smart manufacturing. *Time Magazine Wrapper*, pages 28–33, 2010.

[10] D. K. W. Chiu, Q. Li, and K. Karlapalem. A Logical Framework for Exception Handling in ADOME Workflow Management System. In *Proccedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 110–125. Springer-Verlag, 2000.

[11] J. Claßen, P. Eyerich, G. Lakemeyer, and B. Nebel. Towards an Integration of Golog and Planning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, IJCAI 2007, pages 1846–1851, 2007.

[12] J. Claßen, Y. Hu, and G. Lakemeyer. A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence*, AAAI-07, pages 956–961, 2007.

[13] F. Cossu, A. Marrella, M. Mecella, A. Russo, G. Bertazzoni, M. Suppa, and F. Grasso. Improving operational support in hospital wards through vocal interfaces and process-awareness. In *25th Int. Symp. on Computer-Based Medical Systems (CBMS)*. IEEE, 2012.

[14] G. De Giacomo, Y. Lespérance, H. Levesque, and S. Sardina. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. *Multi-Agent Programming: Languages, Tools and Applications*, pages 31–72, 2009.

[15] G. De Giacomo, M. Mecella, and F. Patrizi. Automated Service Composition Based on Behaviors: The Roman Model. In *Web Services Foundations*, pages 189–214. Springer, 2014.

[16] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution Monitoring of High-Level Robot Programs. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, KR'98, pages 453–465, 1998.

[17] M. de Leoni, A. Marrella, and A. Russo. Process-aware information systems for emergency management. In *European Conference on a Service-Based Internet*, pages 50–58. Springer, 2010.

[18] A. Dix, J. Finlay, G. Abowd, and R. Beale. Human-Computer Interaction. *Pearson*, 2004.

[19] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers. *Fundamentals of Business Process Management*. Springer-Verlag Berlin Heidelberg, 1st edition, 2013.

[20] M. Dumas, W. M. van der Aalst, and A. H. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. John Wiley & Sons, 1st edition, 2005.

[21] S. Edelkamp and J. Hoffmann. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical report, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.

[22] J. Eder and W. Liebhart. The Workflow Activity Model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems*, CoopIS-95, pages 87–98, 1995.

[23] J. Eder and W. Liebhart. Workflow recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems*, CoopIS'96, pages 124–134. IEEE Computer Society, 1996.

[24] H. M. Ferreira and D. R. Ferreira. An Integrated Life Cycle for Workflow Management Based on Learning and Planning. *International Journal on Cooperative Information Systems*, 15, 2006.

[25] C. Fritz, J. A. Baier, and S. A. McIlraith. ConGolog, Sin Trans: Compiling ConGolog into Basic Action Theories for Planning and Beyond. In *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, KR'08, pages 600–610, 2008.

[26] C. Fritz and S. A. McIlraith. Monitoring Plan Optimality During Execution. In *ICAPS*, pages 144–151, 2007.

[27] 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 2005, pages 982–986. IEEE Computer Society Press, 2005.

[28] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers, 2013.

[29] A. Gerevini, A. Saetti, I. Serina, and P. Toninelli. LPG-TD: a Fully Automated Planner for PDDL2.2 Domains. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, ICAPS-04, 2004.

[30] D. Ghosh, R. Sharman, H. R. Rao, and S. J. Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[31] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.

[32] O. Hanteer, A. Marrella, M. Mecella, and T. Catarci. A petri-net based approach to measure the learnability of interactive systems. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI 2016, Bari, Italy, June 7-10, 2016*, pages 312–313, 2016.

[33] R. Hull and H. R. Motahari Nezhad. Rethinking BPM in a cognitive world: Transforming how we learn and perform business processes. In *Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings*, volume 9850 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2016.

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

[35] W. D. Kingery. Introduction to ceramics. 1960.

[36] M. Klein and C. Dellarocas. A Knowledge-based Approach to Handling Exceptions in Workflow Systems. *Computer Supported Cooperative Work (CSCW)*, 9(3-4):399–412, 2000.

[37] A. Lanz, M. Reichert, and P. Dadam. Robust and Flexible Error Handling in the AristaFlow BPM Suite. In *Information Systems Evolution: CAiSE Forum 2010*, pages 174–189. Springer Berlin Heidelberg, 2011.

[38] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. Industry 4.0. *Business & Information Systems Engineering*, 6(4):239–242, 2014.

[39] E. A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC 2008, pages 363–369. IEEE, 2008.

[40] B. S. Lerner, S. Christov, L. J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise. Exception Handling Patterns for Process Modeling. *IEEE Transactions on Software Engineering*, 36(2):162–183, 2010.

[41] Z. Luo, A. Sheth, K. Kochut, and J. Miller. Exception Handling in Workflow Systems. *Applied Intelligence*, 13(2):125–147, 2000.

[42] A. Marrella, M. Mecella, and S. Sardina. SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, 2014.

[43] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - The Planning Domain Definition Language. Technical report, 1998.

[44] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*, pages 482–496, 2002.

[45] M. Minor, R. Bergmann, and S. Görg. Case-based adaptation of workflows. *Information Systems*, 40:142–152, 2014.

[46] C. Muise, S. A. McIlraith, and J. C. Beck. Monitoring the execution of partial-order plans via regression. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, 2011.

[47] R. Müller, U. Greiner, and E. Rahm. AGENT WORK: A Workflow System Supporting Rule-based Workflow Adaptation. *Data & Knowledge Engineering*, 51(2):223–256, Nov. 2004.

[48] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research(JAIR)*, 20:379–404, 2003.

[49] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[50] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proceedings of the 2005 IEEE International Conference on Web Services*, ICWS'05, pages 293–301. IEEE Computer Society, 2005.

[51] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of executable web service compositions from BPEL4WS processes. In *Proceedings of the 14th International Conference on World Wide Web*, WWW 2005, pages 1186–1187. ACM, 2005.

[52] H. Psaier and S. Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, 2011.

[53] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-Physical Systems: The Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference*, DAC 2010, pages 731–736. IEEE, 2010.

[54] A. S. Rao and M. P. Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems*, ICMAS 95, pages 312–319, 1995.

[55] M. Reichert and P. Dadam. ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[56] M. Reichert, S. Rinderle, and P. Dadam. ADEPT Workflow Management System. In *Proceedings of the 1st International Conference on Business Process Management*, BPM 2003, pages 370–379. Springer Berlin Heidelberg, 2003.

[57] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive Process Management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 1113–1114, 2005.

[58] M. Reichert and B. Weber. *Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies*. Springer Berlin Heidelberg, 2012.

[59] H. Reichgelt. *Knowledge Representation: An AI perspective*. Greenwood Publishing Group Inc., 1991.

[60] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[61] S. Rinderle, M. Reichert, and P. Dadam. Correctness Criteria for Dynamic Changes in Workflow Systems: A Survey. *Data & Knowledge Engineering*, 50(1):9–34, 2004.

[62] S. Rinderle, B. Weber, M. Reichert, and W. Wild. Integrating Process Learning and Process Evolution – A Semantics Based Approach. In *Proccedings of the Third International Conference on Business Process Management*, BPM 2005, pages 252–267. Springer Berlin Heidelberg, 2005.

[63] N. Russell, W. M. van der Aalst, and A. H. ter Hofstede. Workflow exception patterns. In *Proceedings of the 18th International Conference on Advanced Information Systems Engineering*, CAiSE 2006, pages 288–302. Springer Berlin Heidelberg, 2006.

[64] R. Seiger, C. Keller, F. Niebling, and T. Schlegel. Modelling complex and flexible processes for smart cyber-physical environments. *Journal of Computational Science*, pages 137–148, 2014.

[65] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth International Conference on Artificial intelligence*, AAAI '94, pages 337–343. American Association for Artificial Intelligence, 1994.

[66] S. Sohrabi and S. A. McIlraith. Preference-based web service composition: A middle ground between execution and search. In *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I*, pages 713–729, 2010.

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

[68] N. R. van Beest, E. Kaldeli, P. Bulanov, J. C. Wortmann, and A. Lazovik. Automated runtime repair of business processes. *Information Systems*, 39:45–79, 2014.

[69] T. H. Van De Belt, L. J. Engelen, S. A. Berben, and L. Schoonhoven. Definition of Health 2.0 and Medicine 2.0: a systematic review. *Journal of medical Internet research*, 12(2), 2010.

[70] W. M. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed Parallel Databases*, 14(1), 2003.

[71] B. Weber, M. Reichert, and S. Rinderle. Change Patterns and Change Support Features - Enhancing Flexibility in Process-aware Information Systems. *Data & knowledge engineering*, 66(3):438–466, 2008.

[72] B. Weber, W. Wild, and R. Breu. CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-Based Reasoning. ECCBR 2004. Springer Berlin Heidelberg, 2004.

[73] 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. IEEE, 2001.

[74] D. E. Wilkins. *Practical planning: extending the classical AI planning paradigm*. Morgan Kaufmann, 1988.