Coordinating Mobile Actors in Pervasive and Mobile Scenarios: An AI-based Approach

Massimiliano de Leoni, Andrea Marrella,
Massimo Mecella, Stefano Valentini
Dipartimento di Informatica e Sistemistica
SAPIENZA – Università di Roma, ITALY
<lastname>@dis.uniroma1.it

Sebastian Sardina
Department of Computer Science
RMIT University, AUSTRALIA
sebastian.sardina@rmit.edu.au

Abstract

Process Management Systems (PMSs) can be used not only in classical business scenarios, but also in highly dynamic and uncertain environments, for example, in supporting operators during Emergency Management for coordinating their activities. In such challenging situations, processes should be adapted in order to cope with anomalous situations, including connection anomalies and task faults. This requires the provision of intelligent support for the planning and enactment of complex processes, that allows to capture the knowledge about the dynamic context of a process. In this paper, we show how this knowledge, together with information about the capabilities of the available actors, may be specified and used to not only to support the selection of an appropriate set of agents to fill the roles in a given task, but also to solve the problem of adaptivity. The paper describes a first prototype of a PMS based on well-known Artificial Intelligence Techniques and how it can be extended to tackle adaptation.

Keywords Process Management Systems, AI-based coordination, Emergency Management

1 Introduction

Nowadays, process management systems (PMSs, [11, 16]) are widely used for the management of "administrative" processes characterized by clear and well-defined structure. Besides such scenarios, which present mainly static characteristics (i.e., deviations are not the rule, but the exception), PMSs can also be used in mobile and highly dynamic situations, for instance, to coordinate operators/devices/robots/sensors [1].

Let us consider just for example a scenario for emergency management. There, a PMS can be used to coordinate the activities of a emergency operators within teams. There, the members of a team are equipped with PDAs and are coordinated through a PMS residing on a leader device (usually an ultra-mobile laptop). In such a PMS, process schemas (in the form of Activity Diagrams) are defined, describing different aspects, such as tasks/activities, control and data flow, tasks assignment to services, etc. Every task gets associated to a set of conditions which ought to be true for the task to be performed; conditions are defined on control and data flow (e.g., a previous task has to be completed or a variable needs to be assigned a specific range of values). Devices communicate among themselves through ad hoc networks; and in order to carry on the whole process, such devices need to be continually connected to the PMS. However, this cannot be guaranteed: the environment is highly dynamic and the movement of nodes (that is, devices and the related operators) around the affected area to carry out assigned tasks can cause disconnections and, thus, unavailability of nodes. This means that in highly dynamic scenarios processes can be easily invalidated, since the execution environment may change continuously because of frequent unforeseeable events and, thus, the process cannot carried on. Because of all this, some type of *process adaptivity* is desirable in such scenarios. But what does "adaptivity" mean? Adaptivity can be seen as the ability of the PMS to reduce the gap from the *virtual reality*, the (idealized) model of reality that is used by the PMS to deliberate, and the *physical reality*, the real world with the actual values of conditions and outcomes. For instance in scenarios of emergency management, PMS may restructure by including a *follow* X task to be assigned to a certain service in order to avoid the X's disconnection.

The reduction of such gap requires sufficient knowledge of both kinds of realities (virtual and physical). Such knowledge, determined and harvested by the services performing the process' tasks, would allow the sensing of deviations and the adaptation of the process for the final goal.

In [5], a general framework has been proposed to address the task of automatically adapting a process when a gap is sensed between the virtual and physical realities. As discussed there, other systems aim at addressing the critical issue of automatic adaptation. Here, we report on our ongo-

ing efforts to operationalize such framework by making use of the INDIGOLOG agent architecture [2, 3, 15] developed at the University of Toronto's Cognitive Robotics Group.[1] In particular, in this paper we describe the architecture of a preliminary version and show its applicability through an example stemming from emergency management. We note that while the paper develops all the required features for carrying out processes, it does not handle adaptivity as of yet, although it is arranged for this purpose.

The paper is organized as follows. Section 2 provides a background overview of the machinery to be used for representing and reasoning about processes and exogenous events, namely the Situation Calculus and the INDIGOLOG high-level programming language. Section 3 presents the general conceptual framework to address adaptivity in dynamic scenarios; whereas Section 4 shows how these conceptual models can be formalized within the Situation Calculus. Section 5 describes the conceptual architecture of the PMS component inside the INDIGOLOG module. Section 6 shows its applicability in the specific domain of emergency management. Finally, Section 7 concludes the paper by explaining how we intend to extend the current version in order to handle adaptivity.

2 Preliminaries

In this section we briefly introduce the framework used to to formalize the adaptivity in PMSs. The situation calculus is a logic formalism designed for representing and reasoning about dynamical domains [13]. In the situation calculus, a dynamic world is modeled as progressing through a series of *situations* as a result of various *actions* being performed. A situation represents a history of actions occurrences. The constant S_0 denotes the initial situation, and a special binary function symbol $do(\alpha, s)$ denotes the next situation after performing the action α in the situation s. A special binary relation $Poss(\alpha, s)$ is used to denote that action α is executable in situation s. Statements whose truth value may change are modeled by means of *relational fluents* and *functional fluents*, predicates and functions, respectively, which take a situation as their final argument (e.g., $Holding(x, s)$ and $Color(x, s)$). So, fluents may be thought of as "properties" or "features" of the world whose values may vary across situations. Changes in fluents (resulting from actions occurrences) are specified through the so-called *successor state axioms*. The successor state axiom for a particular fluent F captures the effects and non-effect of actions on F and has the following form:

$$F(\vec{x}, do(\alpha, s)) \equiv \Phi_F(\vec{x}, \alpha, s),$$

where $\Phi_F(\vec{x}, \alpha, s)$ is a formula fully capturing the truth-value of fluent F on objects $\vec{x}$ when action α is performed in situation s ($\vec{x}$, α, and s are all free-variables in Φ_F).

[1] INDIGOLOG is freely available `www.cs.toronto.edu/cogrobo`.

Within this language, one can formulate action theories that describe how the world changes as the result of the available actions. For example, basic action theories [13], include domain-independent foundational axioms that describe the structure of the situations, one successor state axiom per fluent, one precondition axiom per action, and initial state axioms that describe what is true initially.

On top of these theories of actions, one can define complex control behavior by means of high-level programs expressed in GOLOG-like programming languages [13]. In particular, we shall use here INDIGOLOG [14], an agent architecture completely implemented in Prolog. We chose INDIGOLOG for several reasons. First, it includes primitives for expressing concurrency. Second, it allows agent planning to be *interleaved* with agent acting, giving rise to the so-called incremental executions. GOLOG, in contrast, searches for a whole legal execution before executing even the first action, something unfeasible for our application where agents execute complex and long tasks. In fact, our agents must do some planning, then execute some of the plan constructed, then engage in some more planning, and so on. Third, the execution of INDIGOLOG programs allows for the agent to perform *sensing* of the environment it is acting on, and adapt their executions accordingly. Finally, INDIGOLOG's execution scheme accommodates the occurrence of exogenous events-actions. In our domain, other agents or entities may perform actions that are outside the control of our agents, who in turn need to diagnose, incorporate, and reason about such exogenous actions. In fact, when exogenous actions occur, plans may no longer be valid. It may hence be necessary to monitor the execution of plans, and perform re-planning or plan repair when exogenous events turn these plans unsuccessful.

In Section 4, thus, we shall show how to represent processes in our PMS by means of INDIGOLOG programs.

3 General Framework

The general framework which we shall introduce in this paper is based on the *execution monitoring* scheme as described in [10, 4] for situation calculus agents. As we will later describe in more details, when using INDIGOLOG for process management, we take tasks to be predefined sequences of actions (see later) and processes to be INDIGOLOG programs. After each action, the PMS may need to align the internal world representation (i.e., the virtual reality) with the external one (i.e., the physical reality).

Before a process starts, PMS takes the initial context from the real environment and builds the corresponding initial situation S_0, by means of first-order logic formulas. It also builds the program δ_0 corresponding to the process to be carried on. Then, at each execution step, PMS, which has a complete knowledge of the internal world (i.e., its virtual reality), assigns a task to a service. The only "assignable" tasks are those whose preconditions are fulfilled. A service

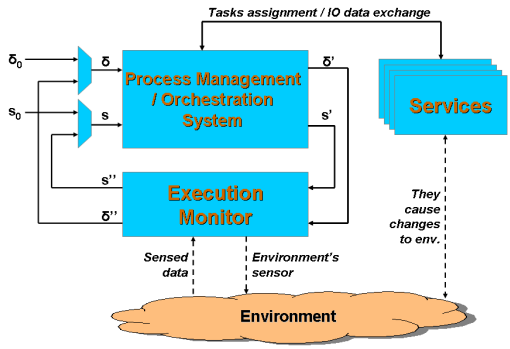

Figure 1. Execution Monitoring.

can collect data required needed to execute the task assigned from PMS. When a service finishes executing a task, it alerts PMS of that.

The execution of the PMS can be interrupted by the *monitor* module when a misalignment between the virtual and the physical realities is discovered. In that case, the monitor *adapts* the (current) program to deal with such discrepancy.

In Figure 1, the overall framework is depicted. At each step, the PMS advances the process δ in situation s by executing an action, resulting then in a new situation s' with the process δ' remaining to be executed. Both s' and δ' are given as input to the monitor, which also collects data from the environment through *sensors*.[2] If a discrepancy between the virtual reality as represented by s' and the physical reality is sensed, then the monitor changes s' to s'', by generating a sequence of actions that explains the changes perceived in the environment, thus re-aligning the virtual and physical realities. Notice, however, that the process δ' may *fail* to execute successfully (i.e., assign all tasks as required) in the new (unexpected) situation s''. If so, the monitor adapts also the (current) process by performing suitable recovery changes and generating then a new process δ''. At this point, the PMS is resumed and the execution continues with program-process δ'' in situation s''.

4 Formalization

In this section, we show how to formalize the general framework proposed above in the Situation Calculus-based INDIGOLOG language. First of all, we shall use some *domain-independent* and *situation-independent* predicates to denote the various objects of interest present in our framework, namely:

- $Service(a)$: a is a service able to execute tasks;
- $Task(x)$: x is a task;
- $Capability(b)$: b is a capability;
- $Require(x, b)$: the task x requires the capability b;
- $Provide(a, b)$: the service a provides the capability b;

Every task realization is the sequence of four actions, all performed by PMS. We formalize as follows:

- $assign(a, x)$: task x is assigned to service a;
- $start(a, x, p)$: a should start executing task x with variable p as support information;
- $stop(a, x, q)$: service a has successfully finished task x with output q;
- $release(a, x)$: task x is realeased from service a.

The value p and q denote arbitrary sets of input/output, which depend on the specific task; if no input or output is needed, p and q are $\emptyset$.

Then, for each specific domain, we have several fluents representing the relevant properties of world. The framework assumes the presence of a process designer responsible for defining the INDIGOLOG program and the corresponding action theory. In particular, a specific definition of domain-dependent fluent $Available(a, s)$ needs to be provided, capturing the fact that the PMS can assign a task to service a in situation s. Though necessary, it is generally not enough for a service to be "free" for the PMS to be able to assign a task to it—other domain properties must also hold. For instance, in the emergency management on MANET scenario, a service has to not only be free, but also be connected to the device holding the PMS. Because of this, we take relation $Available(a, s)$ not to be a fluent perse, but an abbreviation of the following form:

$$\forall a, s. Available(a, s) \stackrel{\text{def}}{=} Free(a, s) \wedge \psi(a, s). \quad (1)$$

where $\psi(a, s)$ is meant to capture the (extra) domain constraints that need to be met for a to be *available* to the PMS, besides being "free."

Finally, if services can only handle at most one task at the time, then we would also include the following successor state axiom for fluent $Free(a, s)$:

$$\begin{aligned} &\forall a, s. Free(a, do(\alpha, s)) \equiv \\ &\qquad (\exists x)\alpha = release(a, x) \ \vee \\ &\qquad Free(a, s) \wedge (\forall x)\alpha \neq assign(a, x). \end{aligned} \quad (2)$$

In words, service a is considered *free* after action α has been executed in situation s *if and only if* α is the action of releasing service a or service a was already free in the previous situation and α does not assign any task to a.

[2] Here, we refer as *sensors* not only proper sensors (e.g., the ones deployed in sensor networks), but also any software or hardware component enabling to retrieve contextual information. For instance, it may range from GIS clients to specific hardware that makes available the communication distance of a device to its neighbors. [6]

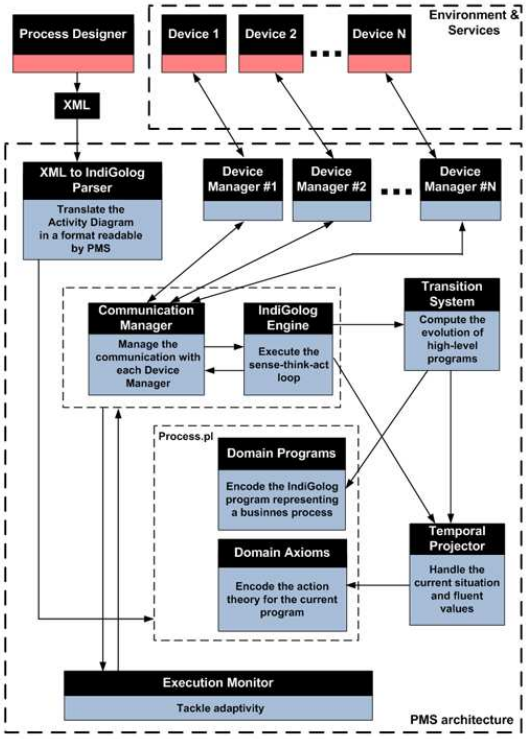

Figure 2. Architecture of the PMS.

5 Architecture

This section aims at describing the internal structure of PMS. Figure 2 shows its conceptual architecture. At the beginning, a responsible person designs an Activity Diagram through a *Process Designer* tool. Such a tool translates the Activity Diagram in a XML format file. Then, such a XML file is loaded into PMS. The *XML-to-*INDIGOLOG *Parser* component translates this specification in a *Domain Program*, the INDIGOLOG program corresponding to the designed process, and a set of *Domain Axioms*, which is the action theory comprising the initial situation, the set of available actions with their pre- and post-conditions.

When the program is translated in the Domain Program and Axioms, a component named *Communication Manager* (CM) starts up all of *device managers*, which are basically some drivers for making communicate PMS with the services and sensors installed on devices. For each real world device PMS holds a device manager. After this initialization process, CM activates the INDIGOLOG *Engine*, which is in charge of executing INDIGOLOG programs. Then, CM enters into a passive mode where it is is listening for messages arriving from the devices through the device managers. In general, a message can be a exogenous event harvested by a certain sensor installed on a given device as well as a message notifying the beginning or the completion of a certain task. When CM judges a message as significant, it forwards it to INDIGOLOG, such as, the signalling of the completion of a certain task or the sudden unavailability of a given device.

The Communication Manager can be invoked by the INDIGOLOG Engine whenever it produces an action for execution. CM picks a service judged as the best for the execution and forwards the request to the appropriate device holding that service.

In sum, CM is responsible of deciding which device should be performing certain actions, instructing the appropriate device managers to communicate with the device services and collecting the corresponding sensing outcome. The INDIGOLOG Engine is intended to execute a *sense-think-act* interleaved loop [8]. The cycle repeats at all times the following three steps:

1. check for exogenous events that have occurred;
2. calculate the next program step; and
3. if the step involves an action, *execute* the action, instructing the Communication Manager.

The INDIGOLOG Engine relies on two further modules named *Transition System* and *Temporal Projector*. The former is used to compute the evolution of INDIGOLOG programs according to the statements' semantic., whereas the latter is in charge of holding the current situations throughout the execution, making possible to evaluate the fluent values

The last module that is worth mentioning is the *Execution Monitor* (MON), which get notifications of exogenous events from the Communication Manager. It decides whether adaptation is needed and adapts accordingly the process. Clearly, in this preliminary version, MON is just a stub, since adaptation is not implemented. Section 7 illustrates how MON is envisioned in the final version.

6 A Running Example

Figure 3 depicts a process example stemming from [1] as an informal Activity Diagram. The process consists of two concurrent branches; the final task is *Send data with gprs* which can be executed only when both of branches are successfully completed. The left branch comprises three concurrent execution of the *rescue* tasks followed by three concurrent execution of the sequence of task *evacuation* and *census*. When a *evacuation* task terminates, the following task *census* can be started. Similarly, the right branch begins with the concurrent execution of three sequences of tasks *photo* and *survey*. When every survey task is terminated, task *evaluate photo* shall be executed. Then, a condition is evaluated on the current state at a decision point. If the condition holds, the right branch is considered as concluded. Otherwise, the whole branch is repeated, including the testing condition at the decision point.

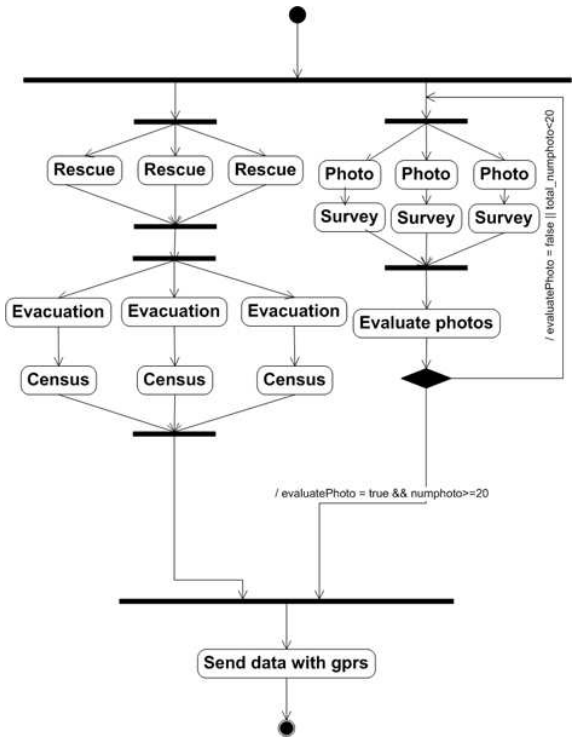

Figure 3. The Activity Diagram representing a process to be executed.

```
proc(manageTask(X,D,I,O),
  pick(b,[?(Require(X,b) = TRUE),
    pick(a,[?(and(Provided(a,b) = TRUE,
         Available(a) = TRUE)),
            assign(X,D,a),
            <wait for the notification of the task X start>,
            start(X,D,a,I,O),
            <wait for the notification of the task X end>,
            stop(X,D,a),
            release(X,D,a)
         ])
       ])
      ).

proc(mainControl(1), [
    itconc([
        [itconc([                   % Left branch
            [manageTask(rescue,id_8,loc,[])],
            [manageTask(rescue,id_9,loc,[])],
            [manageTask(rescue,id_10,loc,[])]
        ]),
        itconc([
            [manageTask(evacuation,id_11,loc,[]),
            manageTask(census,id_12,loc,text)],
            [manageTask(evacuation,id_13,loc,[]),
            manageTask(census,id_14,loc,text)],
            [manageTask(evacuation,id_15,loc,[]),
            manageTask(census,id_16,loc,text)]
        ])
    ],
    [while(or(nophoto(id_2)+        % Right branch
        nophoto(id_3)+nophoto(id_4)<20,
        evaluation(id_7)=false),
        [itconc([
            [manageTask(photo,id_2,loc,numphoto),
            manageTask(survey,id_5,loc,questionnaire)],
            [manageTask(photo,id_4,loc,numphoto),
            manageTask(survey,id_18,loc,questionnaire)],
            [manageTask(photo,id_3,loc,numphoto),
            manageTask(survey,id_6,loc,questionnaire)]
            ]),
            manageTask(evaluatephoto,id_7,loc,evaluation)]
        )]
    ]),
    manageTask(senddata,id_17,info,sendOK) % Last Step
]).
```

Figure 4. The INDIGOLOG program of the example in Figure 3

Figure 6 describes the program as computed by the XML-to-INDIGOLOG parser.

The program defines a sub-routine `manageTask` which handles the whole cycle of a task, since it is assigned until it is realized. The `manageTask` routine picks a service a providing every capability required by task X and assigns X to a, which is notified about that. Then, the INDIGOLOG engine waits for the notification coming from Communication Manager that a is willing to begin the X's execution. Upon arriving of such a notification, it executes `start` internally and acknowledges a so that a can begin the execution. Finally, it waits for the notification that n terminated the execution. Upon receiving it, it executes the `stop` and `release` statements to change fluents to make a available again in order that a can receive another task assignment.

In order to model the action assignment, the INDIGOLOG implementation introduces the statement `pick(serv, prog)`, which is a concrete implementation of the $\pi x.\delta(x)$ construct. We use such construct to "select" a service `serv` such that program `prog` can successfully execute. At this stage, it uses a trivial approach by taking the first (any) available service. Nevertheless, we aim at implementing a smarter approach by considering logging information of past executions of similar tasks and by balancing the load among all available services [12, 9].

Notice that if the `pick` construct is not able to find any service, then the INDIGOLOG program remains "blocked", waiting for a proper service selection that would allow program `prog` to be executed.

Procedure `mainControl` is the starting point of the process. Every process task execution corresponds to an invocation of the `manageTask` routine. The concurrency in the execution of tasks is granted by the special construct `itconc(A,B)` that executes programs `A` and `B` concurrently in a (fair) round-robin manner. The branch on the right in the activity diagram of the examples basically a `while` cycle. The cycle body is repeated while either the number of photos taken by the `photo` task instances whose identifier is id_2, id_4 and id_3 is less than 20 or the photos quality is not judged good enough. Predicates $nophoto(\cdot)$ and $evaluation(\cdot)$ are automatically updated by the Communication Manager upon receiving such an information from services through the respective Device Managers.

7 Conclusions

In this paper, we have introduced a novel approach to tackle the issue of automatically adapting business processes in highly dynamic environments. Other PMSs rely on the existence of a domain expert who is responsible of changing the process at hand in order to deal with exceptional events. In highly dynamic scenarios, exogenous unforeseen events are quite frequent and certainly not exceptions. As a consequence, the task of manually handling the changes would become too difficult, if not unfeasible. The solution we are proposing and working on, which has already been partially operationalized, relies on well-known techniques and frameworks in Artificial Intelligence, such as the Situation Calculus and automated planning.

Currently, we are working on the adaptation process. Recall that the *Execution Monitor* meant to sense the *real* values of fluents to recognize any *gap* between the expected and the real states. If a gap is indeed sensed (i.e., at least one has changed to an unexpected value), the monitor has to "adapt" the INDIGOLOG program that is currently representing the process.

Adaptation amounts to finding a linear program (i.e., one without concurrency) that is meant to be "appended" before the current INDIGOLOG program remaining to be executed. This new linear program is meant to resolve the *gap* that was just sensed, by restoring the value of the affected fluents. In that way, the remaining of the original program is again able to execute.

More concretely, let δ' be the process/program still to be carried on, and let ϕ be the formula representing the state that has to be restored for the correct execution of δ'. For example, for simplicity, formula ϕ may state the *expected* value of each fluent. Then, within INDIGOLOG, we can formalize the adaptation process by making the PMS change the current program δ' to program $\delta'' = \Sigma[(\pi\, a.a)^*; ?\phi]; \delta'$. The INDIGOLOG *search operator* $\Sigma\delta$ provides a mechanism for off-line lookahead so as to find a complete execution of program δ. Though δ can be any INDIGOLOG program, we can always consider the case $\delta = (\pi\, a.a)^*; ?\phi$. It would mean we are requiring INDIGOLOG to perform first-principle planning: find a sequence of actions that would make ϕ true.

While the current implementation of INDIGOLOG does not handle such off-line task in any specialized way, one can imagine using any of the current state-of-the-art classical planners [7] to implement our particular adaptive process. This is indeed the direction we are currently following.

Acknowledgements. The work at SAPIENZA has been supported by the European Commission through the project FP6-2005-IST-5-034749 WORKPAD. The last author was supported by the Australian Research Council and AOS under the grant LP0560702, and the National Science and Engineering Research Council of Canada under a PDF fellowship.

References

[1] T. Catarci, M. de Leoni, A. Marrella, M. Mecella, B. Salvatore, G. Vetere, S. Dustdar, L. Juszczyk, A. Manzoor, and H. Truong. Pervasive Software Environments for Supporting Disaster Responses. *IEEE Internet Computing*, 12:26–37, 2008.

[2] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri, editors, *Logical foundation for cognitive agents: Contributions in honor of Ray Reiter*, pages 86–102. Springer, Berlin, 1999.

[3] G. De Giacomo, H. J. Levesque, and S. Sardina. Incremental execution of guarded theories. *ACM Transactions on Computational Logic (TOCL)*, 2(4):495–525, October 2001.

[4] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *Proc. of KR-98*, pages 453–465, 1998.

[5] M. de Leoni, M. Mecella, and G. De Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. In *BPM*, pages 182–197, 2007.

[6] M. de Leoni, M. Mecella, and R. Russo. A Bayesian Approach for Disconnection Management in Mobile Ad-hoc Networks. In *Proc. 4th International Workshop on Interdisciplinary Aspects of Coordination Applied to Pervasive Environments: Models and Applications (CoMA) (at WETICE 2007)*, 2007.

[7] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[8] R. Kowalski. Using Meta-logic to Reconcile Reactive with Rational Agents. pages 227–242, 1995.

[9] A. Kumar, W. van der Aalst, and H. Verbeek. Dynamic work distribution in workflow management systems: How to balance quality and performance. *Journal of Management Information Systems*, 18(3), 2002.

[10] Y. Lespéerance and H. Ng. Integrating Planning into Reactive High-level Robot Programs, 2000.

[11] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 1999.

[12] H. A. Reijers, M. H. Jansen-Vullers, M. zur Muehlen, and W. Appl. Workflow management systems + swarm intelligence = dynamic task assignment for emergency management applications. In *BPM*, pages 125–140, 2007.

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

[14] S. Sardina, G. De Giacomo, Y. Lespénce, and H. Levesque. On the semantics of deliberation in IndiGolog – from theory to implementation. 41(2–4):259–299, August 2004.

[15] S. Sardina and S. Vassos. The Wumpus World in IndiGolog: A preliminary report. In L. Morgenstern and M. Pagnucco, editors, *Proceedings of the Workshop on Non-monotonic Reasoning, Action and Change at IJCAI (NRAC-05)*, pages 90–95, 2005.

[16] W. van der Aalst and K. van Hee. *Workflow Management. Models, Methods, and Systems*. MIT Press, 2004.