

Reflective Agents for Adaptive Workflows

Uwe M. Borghoff¹, Paolo Bottoni², Piero Mussio² and Remo Pareschi¹

¹ Rank Xerox Research Centre, Grenoble Laboratory
6, chemin de Maupertuis. F-38240 Meylan, France
{borghoff, pareschi}@grenoble.rxc.xerox.com

² Dipartimento di Scienze dell'Informazione, Universita' La Sapienza di Roma
Via Salaria 113. I-00198 Roma, Italy
{bottoni, mussio}@dsi.uniroma1.it

Abstract

Adaptation to changes in organizational procedures and business rules is a *sine qua non* for workflow management systems, if they have to be useful to organizations. This paper describes an approach based on workflow agents capable of managing dynamic changes in business policies. The two key aspects of our approach are (i) the agentification of the process engine, in the sense that this is viewed as controlled and executed by autonomous workflow agents capable of reacting and adapting to external changes, and (ii) the fact that these agents are reflective, in the sense that they can observe and thus modify their own behavior. The model is described in the context of an agent-based framework for coordination with reflective capabilities, and is evaluated against a well-known case of dynamic change.

1 Introduction

Workflow management systems are supposed to streamline the execution of business operations by automating the running of corresponding business policies. However, business policies are bound to change, either because of the arising of exceptions and of the need of re-planning during their execution, or because the management of an organization decides to re-design the organization's processes [HC93]. Thus, workflow systems should be able to cope with *dynamic changes*, where a dynamic change is a change in a procedure that is performed while the procedure is in the course of being executed. Managing a dynamic change requires not only the procedure to be consistent before and after the change, but also that those processes, which have already performed a portion of the affected procedure, can safely complete it. As a consequence, when changes occur, a workflow system has to adapt the affected parts of the workflow to the changes, without disrupting ongoing activities, and without affecting operations in those parts of the process which are not touched by the change.

We take as an example changes in a workflow for managing orders in a company (from receiving the order to either archiving or rejecting it), as described in [EKR95]. An approval process for the order comprises first an inventory check and then a credit check. It ends with approval (leading to shipping and billing) or rejection. A workflow instance is completed either by an archiving task, or by notifying the rejection to the customer. In a first version, shipping and billing are performed concurrently, while the rest of the activities is performed sequentially (see Fig. 1a).

Now, suppose that the workflow administrator wants:

- to sequentialize billing and shipping, so as to ensure that shipping is performed after the billing process has completed, while archiving remains the final action for an approved order (Fig. 1b). Then care must be taken to archive orders that have already been shipped, but are still unbilled at the time the change in the procedure is enforced – otherwise we may end up with a bunch of unarchived orders.
- to parallelize execution of the rules for inventory and credit checks. If we do not distinguish which orders have to be processed by which procedure, orders which have completed the credit check will be approved without going through the inventory check (Fig. 1c).

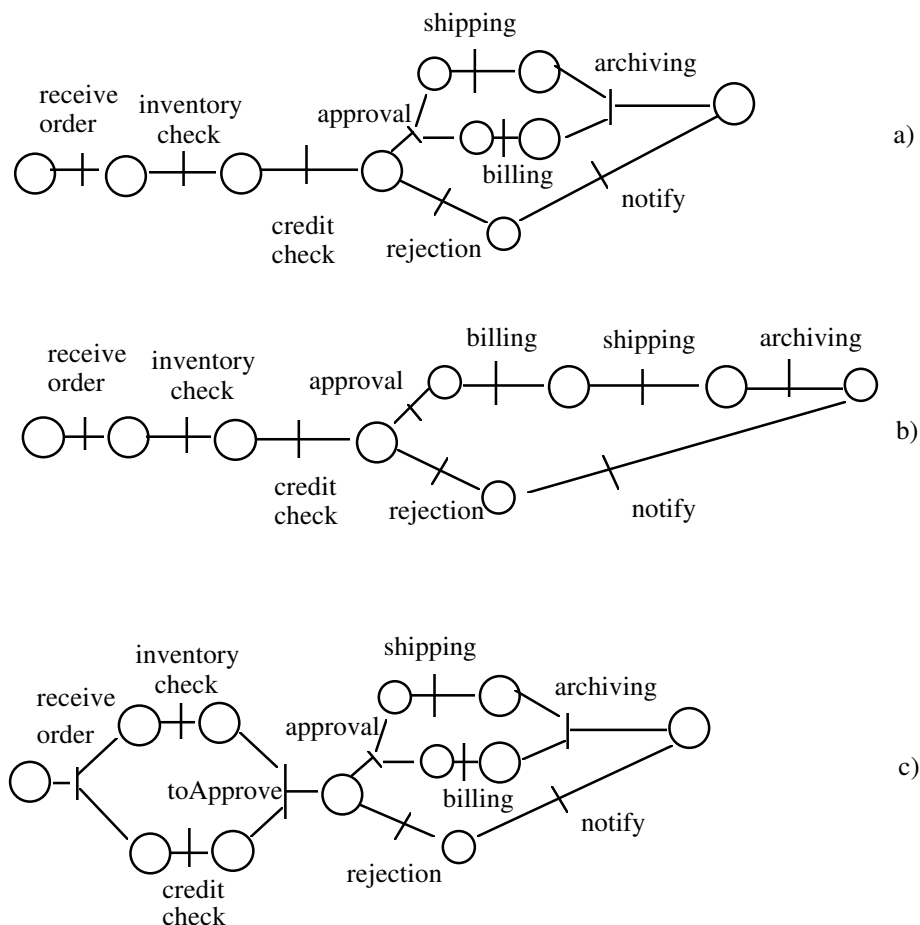


Figure 1. a) original situation, b) sequentializing shipping and billing, c) parallelizing the checks

These two cases show that a process management model has to deal with two different kinds of partial orders induced on the activities (notice how these partial orders also define a *vertical* evolution of the work): *causal dependencies* among activities, and *prioritization schemes* deriving from different policies within an organization. Strictly speaking, the notion of workflow management as generally understood applies to the first kind of dependencies (e.g., in the example above, detecting that in c) inventory check and credit check are independent and thus can be executed in parallel), while the second kind of dependencies is captured at the meta-level in the form of a “policy management” responsible both for data consistency,

by making sure that changes in procedures do not corrupt the output of already running processes, and organizational consistency, by making sure that changes in procedures do get applied as soon as possible.

A workflow management model should therefore meet the following criteria:

1. It has to allow the specification of the different types of dependencies among activities.
2. It has to be consistent with respect to data and procedures.
3. It has to be adaptive, namely it should be able to deal with changes in the environment and in the structure of the organization.

In this paper we argue that reflective agents provide an effective way to satisfy these three requirements. We represent the behavior of reflective agents through production rules and meta-rules. Our approach, however, is general and compatible with other representation frameworks for reflective behavior.

The paper is organized as follows. In Section 2, we introduce and define reflective agents. In Section 3, we discuss reflective behavior. Section 4 illustrates the management of dynamic changes in a reflective workflow environment, and Section 5 draws some conclusions.

2 Reflective Agents

We define a *reflective agent* as an agent that can use meta-level activities to observe and modify its own behavior so as to adapt it to changes in the environment. In the literature, the integration of meta-level deliberative behaviors [BT95] and of reactive behaviors has been approached through reactive planning (i.e. interleaving generation and execution phases of a plan [Osa94]), or by allowing agents to exhibit purely reactive behaviors along with deliberative ones [BR93, Kow96].

We adopt the second approach: agents have a representation of their own abilities, and can accordingly define and modify plans. *Meta-level* attributes control the deliberative definition and modification of plans, while *object-level* attributes control their reactive execution. To get a bit more into details, we define the internal control on the agent's behavior via representations of

- the agent's current state,
- the agent's abilities, i.e. the actions it can perform,
- the trace of the past actions performed by the agent, and
- the goals (or lines of actions) the agent is pursuing.

Hence, the meta-level attributes include an explicit description of the current state, stored in an atom *reportStatus*; a description of the available state and output maps, in an atom *codeText*; an explicit description of the already executed computations, both at object- and meta-level, through an atom *pastActions*; an explicit description of the already planned actions, both at object and meta-level, in an atom *futureActions*. The atoms *pastActions* and *codeText* define the past dynamics of the agent, while *reportStatus*, *codeText*, and *futureActions* allow an (incomplete) foresight on the possible evolution of the agent. This definition of the meta-level is not specific to workflow and can be adopted for other situations requiring reflective behavior. For instance it was used in [BMP94] to describe a reflective system for image interpretation.

In the model defined so far, all actions performed by an agent cause the production or consumption of internal resources. Meta-level attributes define special kinds of resources which can be examined (and consumed) in order a) to assess what the agent can perform in a given situation, b) to find out which lines of actions to commit to, c) to reach a given goal, d) to enable or disable certain actions, and e) to define responsiveness to influences from the environment. Since an agent has access to representations of its own current state and of its own current capabilities, it may determine how to react to modifications perceived in the environment, and how to manage partial failures in plan execution.

2.1 A Rules Based Model of Reflective Agents

Reflective agents as described above can be implemented in a formalism with reflective capabilities. We adopt here *rewriting rules* as they are understood in the planning tradition (see [McH95] for an overview). This notation has been adopted for the CLF (Coordination Language Facility), a program library for the design and implementation of agents capable of coordinating multiple system components [AGP94, AFP96]. In [AMP96], Andreoli *et al.* discuss the use of the CLF notation and environment for the flexible reconfiguration of workflow architectures. Grasso *et al.* [GMPP96] show how this approach can be generalized to workflow architectures for geographically distributed organizational settings.

Rules specify, in a declarative way, the transformations of the agent's state. In particular, rules specify what resources are needed (and consumed) to perform a transformation, and what resources are produced. We employ rules uniformly, both at the object-level and the meta-level. Hence, control knowledge is expressed in a declarative way too. The *state* of an agent is encoded as a multiset $\{a,a,b,c,c,\dots\}$ of atoms which represent the resources available to the agent. The state is also called the pool of resources.

A rule has the form (we print generic resources in bold face)

head $\langle\rangle$ - **body**.

The lhs resources in **head** and the rhs resources in **body** represent non-empty multisets of atoms connected by the symbol @.

The lhs resources constitute the *trigger* of the rule. Two mechanisms control a rule application:

1. A rule may apply only if all of the trigger's atoms occur in the agent state.
2. If a rule applies and is selected for application, the trigger's atoms are removed from the agent state and the resource **body** of the rule is executed in the agent.

In other words, a rule

$a @ b \langle\rangle$ - **body**

applies to any agent containing in its state both atoms *a* and *b*.

The rhs resources, also called a *transformer*, have the following syntax, defined recursively. A transformer **tr** can be either:

1. An atom: in this case, this atom is added to the state of the agent; or
2. **tr1 @ tr2** where **tr1** and **tr2** are transformers: in this case, both **tr1** and **tr2** are put into the agent.

Notice that atoms in an agent state can be structured terms, and atoms in rules may contain variables. Upon application of a rule, the variables in the rule trigger are instantiated by pattern matching with corresponding atoms in the agent state. Furthermore, it is assumed that all agents contain in their state a set of resources representing the underlying computational environment. Such resources are assumed to be immutable (they cannot be removed via rule application) and are called *computational resources*. A rule can access them if they are specified in the trigger, embedded in curly brackets.

For example, the rule

$$p(X) @ \{sum(X,1,Y)\} \leftarrow p(Y)$$

applies to any agent containing an atom of the form $p(a)$ where a is a number, and, if selected, removes this atom and replaces it with the atom $p(b)$ where b is the number $a+1$.

2.2 Levels of Operations and Types of Dependencies

The use of a declarative mechanism allows the expression of coordination of object-level activities in a simple way. The same mechanism is uniformly exploited to define meta-level control on the realization of these activities. The different types of dependencies among activities are reflected by types of resources and rules.

Causal dependency, in which the achievement of a task provides the resources necessary to another task, is expressed by rules at the object-level, while partial orders deriving from *prioritization schemes* are expressed at the meta-level, without the need to insert fictitious synchronization resources in the object-level rules. Thus, the object-level is completely relieved of the necessity of procedural knowledge. Procedural knowledge defining prioritization schemes is expressed, again in declarative terms, at the meta-level.

We now characterize (Section 3) the management of meta-level resources, and will focus then (Section 4) on the management of dynamic changes through a reflective behavior.

3 Meta-level Management

Under our approach, agents change state via the production and consumption of resources at the meta-level and at the object-level. Object-level consumption-production processes involve resources through which causal dependencies may be expressed. These resources are of the form $p(x1, \dots, xn)$ and are defined by rules of the type discussed in the previous section. Meta-level processes monitor the available object-level resources, and determine the plans to pursue. They involve production, consumption, and update of special types of resources.

We characterize these consumption-production processes according to three categories of rules, as shown in Tab. 1. Sub-categories are shown in Tabs. 2-3.

Table 1. A classification of rules according to their format and use

Classification	Description
reified rules	contain their name in their trigger
recording rules	keep consistent representations of evolution
deliberating rules	(partially) determine agent's future evolution

Table 2. Sub-categories of recording rules

tracing rules	modify the <i>pastActions</i> component with a representation of the applied rule
reporting rules	update meta-level representations of available (object-level) resources in <i>currentReport</i>

Table 3. Sub-categories of deliberating rules

meta-rules	enable reified rules by providing the rights for their execution
planning meta-rules	modify the <i>futureActions</i> component with representations of rules to be applied

3.1 Deliberating on Activities

A *reified* rule contains a resource used as a guard on the application of a rule. *Deliberating* rules have the role of producing such resources, which are then consumed upon application of the rule. Deliberating rules can be reified in turn, so that their application is also guarded by a name.

A reified rule has the form

rule-id @ **restOfHead** <>- **body**.

For each identifier rule-id, we assume that there exists an implicit rule of the form

select(rule-id) <>- rule-id.

In this way, the insertion of the atom *select(X)* in the pool of an agent, where *X* is instantiated with a rule-id, makes the rule identified by *X* available to this agent for execution. Identifiers can be extended to define partitions in the set of rules, so that any rule in the set *S* (i.e. sharing a same partition-id *s*) can be employed when *s* is present in the pool. A further extension leads to the definition of theories. A *theory* is a set of rules sharing the same theory-id. A reified rule in a theory has thus the form

theory-id @ rule-id @ **restOfHead** <>- theory-id @ **restOfBody**.

Note that the rule-id is consumed by the application of the rule but the theory-id remains in the pool. Different theories can contain replicas of the same rule, if there is need to apply them in different contexts. A connection is established between the object-level (execution of rules) and the meta-level (control of rules) for determining and controlling the behavior of the agents. This is achieved via *meta-rules*, i.e. rules which contain in their body either atoms of the form *select(X)* or rule-id atoms. Once a meta-rule has placed a multiset of rule-ids in the pool, the execution mechanism becomes responsible of the order in which enabled rules are applied. But it is also desirable to let the agent choose among alternative (partial) sequentializations on the execution of rules. To this end, we introduce the notion of *planning rule*, which is a special kind of metarule that, rather than directly placing the names of reified rules in the pool, modifies the plan an agent is pursuing, as represented in the designated atom *futureActions*. Planning rules are therefore defined as

restOfHead @ futureActions(P) @ {**pcr**(P,**plan_modifier**,P')} <>-
restOfBody @ futureActions(P')

where **pcr** is a suitable *planning computational resource*. The plan P contains a representation of the rules to be applied as well as a partial order of their application. **pcr** can be used either to add rules to a plan, or to remove them. Depending on the chosen representation for the plan P , a suitable set of meta-rules playing the role of *enactment rules* is defined. These rules extract rule identifiers from P and enable the associated rules for application, and then proceed with the plan after given conditions on the already enabled activities are satisfied. In a first simple approach, plans are viewed as sequences of modules, which are themselves sets of rule-ids, meaning that the execution of the entire plan corresponds to the sequential execution of each of its modules, and a module allows concurrent execution of its rules. More refined notions of plans can be devised.

We provide the following basic planning computational resources, from which more complex ones can be built:

- $plan(P, M, P')$ states that plan P' is obtained by concatenating module M to plan P . This resource can be used either as a constructor (given P and M it returns P') or as a destructor (e.g., given P' and M , it returns P).
- $plan0(P)$ states that plan P is an empty sequence of modules.
- $module(M, R, M')$ states that module M' is obtained by adding rule-id R to module M . This resource can also be used either as a constructor or as a destructor.
- $module0(M)$ states that module M is an empty set of rules.

3.2 Recording History

Planning rules can use the “raw information” provided by object-level atoms to take decisions that modify *futureActions*. However, it is also useful to make planning decisions on the basis of information synthesized at the meta-level about the state of the agent and its evolution. Hence, we introduce *recording rules* which realize the upward connection necessary to reflect agent's activities at the meta-level. Recording rules are object-level rules which modify a designated meta-level atom. A recording rule has the form

$$\begin{array}{l} \mathbf{restOfHead} @ \mathbf{currentRecord}(R) @ \{ \mathbf{rcr}(R, \mathbf{rec_modifier}, R') \} \langle - \\ \mathbf{restOfBody} @ \mathbf{currentRecord}(R') \end{array}$$

where **rcr** is a suitable *recording computational resource* and **currentRecord** is some generic meta-level atom whose argument contains a representation of some aspect of the agent and its evolution. Planning rules can then analyze **currentRecord**, so as to devise plans on the basis of recording information about the agent. Several aspects of an agent can be recorded, corresponding to different atoms in place of **currentRecord**. Two aspects are worth remarking: traces of the actions of the agent (atom *pastActions*), and synthetic reports over the current state of the agent (atom *reportStatus*).

Tracing rules are recording rules which maintain a history of the agent's past actions. For simplicity sake, a trace can be viewed as a simple sequence of rule-ids.

Reporting rules are recording rules which maintain synthetic reports about the current state of the agent. We consider here simple reports representing the content of the agent state along some (or all) object-level atoms. Thus, reports are simple multisets of atoms. Two basic computational resources for reporting can be used to build more complex ones: *re-*

$port(R,A,R')$ states that R' is the report obtained by adding the atom A to report R , while $report0(R)$ states that R is the empty report.

A reporting rule has the form

restOfHead @ reportStatus(R) @ {reportRules(R, [**headT**, **bodyT**], R')} <>-
restOfBody @ reportStatus(R')

where **headT** describes the recorded head atoms, **bodyT** describes the recorded body atoms (which must be added to *reportStatus*), and where *reportRules* is constructed on top of the computational resource *report*.

4 Adaptation through Reflection: The Case of Dynamic Changes

In this section, we show how reflective agents can model workflow systems able to handle dynamic changes in the processes defining the workflow. In particular, we address the problem of changes in prioritization schemes.

As discussed in the introduction, problems arise if two activities, originally scheduled sequentially, are executed in parallel in a new procedure. A process having completed the first activity, but not yet entered the second, will never perform the second activity in the new procedure. For this problem, Ellis *et al.* [EKR95] introduce the so-called *synthetic cut-over change*. In this kind of change, the old version of the affected part of the procedure, called the *old region*, is still active, so that process instances which were performing activities in this region can complete them according to the old procedure. Process instances which have not entered the old region before the change occurred will never enter it. Instead, they will perform the activity according to the new definition of the procedure, where the old region is replaced by the new region. Ellis *et al.* propose to model the workflow through a special type of Petri net, but they do not specifically address the management of the individual instances of the workflow. Researchers from the Cooperative Information Systems community [CoopIS96] have recently proposed the notion of *coach agent* to manage this aspect of dynamic change [DW96]. A coach agent enables a cooperation between the different layers of a cooperative information system in order to manage dynamic change: the system layer, where a workflow gets executed, the collaboration layer, where work is (collaboratively) done, and the organization layer, where changes in business policies take place. Our approach is an attempt to provide a practical implementation of this form of cooperation.

A fundamental assumption that we make is that workflows are managed and enacted by reflective agents. We restrict our treatment to the case where there is a single change at a time, each change affecting a connected region in the lattice of activities. The agent in charge of workflow execution is then able to detect the change region, and to execute the single instances according to the procedure which guarantees their termination.

Thus, *workflow agents* are implemented as collections of object-rules and meta-rules. Workflow agents will control the execution of *workflow plans*, namely collections of rule or meta-rule modules with one initial and one final meta-rule module. Each meta-rule module contains either a single meta-rule or a set of mutually exclusive and exhaustive meta-rules. These modules are used to define branching points of the workflow plan, by offering options for conditional routing of workflow tasks on the basis of conditions which are verified at run time. Each rule specifies an *activity*, whose state is encoded in a set of attributes accessed through a specific computational resource, while pre-conditions and post-conditions of the activity are represented as simple (non-computational) resources. A special *instantiation*

meta-rule provides an instance of the workflow with an initial plan, defining sequences of meta-rule modules. Each *instance* of a workflow is characterized by a unique identifier and a collection of attributes modeling the outcomes of the executed activities. All instances of the workflow are collected in a *process* resource, where a representation of the execution plan is also maintained. A *procedure* is a sequence of activities defined by a meta-rule. A resource *execAccord*(**mr**,**tspec**) is associated with each meta-rule **mr**, where **tspec** is a theory-id that indicates the theory according to which the meta-rule has to be executed.

The firing of (meta-) rules defines the progress of the individual workflow instances. A representation of the active instances is maintained in *reportStatus*.

A rule for an activity **someAction** has the form

```
someAction-id @ instance(Id,Atts) @ {someAction(Id,Atts,NewAtts)} <>-
instance(Id,NewAtts)
```

while a meta-rule has the form

```
mrId @ instance(Id,Atts) @ execAccord(mrId,theoryId) @ futureActions(Id,P)
@ {condition(Atts)} @ {plan(P,someModule,NewP)} <>-
futureActions(Id,NewP) @ instance(Id,Atts) @ execAccord(mrId,theoryId).
```

We do not consider change in the implementation of a computational action as a change in the theory. For this reason, object-level rules are only guarded by the corresponding rule-id and not by a theory-id. The activation of rules and meta-rules is also guarded by a resource *susp(false)*. At any time, a process can be suspended by inserting a resource **interrupt**, with the effect of triggering the rule

```
susp(_) @ interrupt <>- susp(true).
```

4.1 Synthetic Cut-over Change

A synthetic cut-over change is defined by the addition of a new meta-rule, replacing an existing meta-rule. This new meta-rule has the same name as the original one, but a different theory-id. The fact that the meta-rule maintains the same name means that “what” it does is the same as before, while changing the theory determines a change in “how” the “what” is achieved. Specifically, changes can be of the following types: a) parallelization of activities which were to be performed in sequence (e.g. from a plan with [a][b] to one with [ab]), or b) sequentialization of activities which were to be performed in parallel (e.g. from [ab] to [a][b]), or c) insertion of a new sequence of activities (e.g. from [a][b] to [a][c][b]), or d) deletion of an old sequence of activities (e.g. from [a][c][b] to [a][b]).

After suspending the process, modifying the rule base and, accordingly, the representation of the resulting workflow plan, as maintained in *codeText(Rls)*, the user places a meta-level resource *chgdThry*(**mrId**,**nthId**) to inform the process that a new version of the meta-rule **mrId** has been inserted. This meta-rule belongs to a new theory **nthId**.

The process assigns the new theory-id to the resource *execAccord*. Instances for which the modified meta-rule has already been evaluated, but that have still to execute the modules prior to the modified ones, will be executed according to the new procedure. Let us consider the following case. A meta-rule *mr* placing the modules [r1][r2][r3][r4] in the plan is replaced by a rule which, in the same conditions, places [r1][r2 r3][r4]. The modification to *execAccord*(**mrId**,**nthId**) ensures that any workflow instance, for which the old version of

mrId has not been evaluated yet, will be processed according to the new version, while any instance for which **mrId** has already been evaluated will proceed in the plan. Now, consider an instance which is proceeding according to the old plan, but which has only completed the module [r1] when it was suspended to change the rules. It is required that it will proceed with the new procedure. The meta-level can now modify the associated plan, by replacing [r2][r3][r4] with [r2 r3] [r4]. For those instances that have already completed the module [r2] no change is performed. By making use of its reflective capabilities, the agent is able to explore *codeText(Rls)* to assess a) which ones of the old meta-rules share the same **mrId**, and b) which modifications in the procedure are induced by the change. In particular, the process can assess that the sub-procedure *ToErase* is replaced by the sub-procedure *ToInsert*. This replacement must occur in the plan for each instance which has not yet entered *ToErase*. The set *Ids* of such instances is evaluated by inspecting the representations of instances and plans in *reportStatus* and *futureActions*.

- 1) `susp(true) @ reportStatus(Ids) @ chgdThry(MrId,NwTh) @ codeText(Rls) @ execAccord(MrId,_) @ {lookDiff(Rls,MrId,ToErase,ToInsert)} @ {reverse(ToInsert,RevToIns)} <>- correctPlan(ToErase,RevToIns) @ execAccord(MrId,NwTh) @ reportStatus(Ids) @ toChk(Ids) @ toRmv(0) @ toIns(0).`
- 2) `correctPlan(ToErase,ToInsert) @ futureActions(Id,P) @ toChk(Ids) @ toRmv(Num) @ toIns(Num) @ {present(ToErase,P)} @ {diff(Id,Ids,LeftIds)} @ {NewNum = Num + 1} <>- correctPlan(ToErase,ToInsert) @ removeFromPlan(ToErase,Id) @ addToPlan(ToInsert,Id) @ toChk(LeftIds) @ futureActions(Id,P) @ toRmv(NewNum) @ toIns(NewNum).`
- 3) `correctPlan(ToErase,ToInsert) @ futureActions(Id,P) @ toChk(Ids) @ {not(present(ToErase,P))} @ {diff(Id,Ids,LeftIds)} <>- correctPlan(ToErase,ToInsert) @ toChk(LeftIds) @ futureActions(Id,P).`
- 4) `removeFromPlan(ToErs,Id) @ futureActions(Id,P) @ {plan(LeftToErs,M,ToErs)} @ {not(plan0(ToErs))} @ {plan(P,M,LeftP)} <>- removeFromPlan(LeftToErs,Id) @ futureActions(Id,LeftP).`
- 5) `removeFromPlan(ToErs,Id) @ toRmv(Num) @ {plan0(ToErs)} @ {New=Num-1} <>- toRmv(New).`
- 6) `addToPlan(ToIns,Id) @ futureActions(Id,P) @ {plan(LeftToIns,M,ToIns)} @ {plan(P,M,NewP)} @ {not(plan0(ToIns))} <>- addToPlan(LeftToIns,Id) @ futureActions(Id,NewP).`
- 7) `addToPlan(ToIns,Id) @ toIns(Num) @ {plan0(ToIns)} @ {New = Num - 1} <>- toIns(New).`
- 8) `toChk(nil) @ correctPlan(,_) @ toIns(0) @ toRmv(0) <>- susp(false).`

Figure 3. The rules for realizing the synthetic cut-over dynamic change

To sum up, the management of synthetic cut-over change is performed as illustrated in Fig. 3, where the reporting mechanism is limited to maintaining a record of the active instances. First the process is started. The resource *toChk* is created to memorize the instances whose plan remains to be checked, and *correctPlan* is created to memorize the changes to be performed (Rule 1 in Fig. 3). Note that the modules in *RevToIns* are in reversed order, because of the definition of the plan constructor. For each instance having *ToErase* in the plan, the system memorizes that its plan has to be corrected (Rule 2), while instances with nothing to

erase (either because the process has not yet executed the meta-rule **mrId**, or because it has already started the sub-procedure) are left untouched (Rule 3). The resources *toIns* and *toRmv* maintain the count of the instances for which a replacement has to be performed. The modules in *ToErase* are erased one at a time using *plan* as destructor (Rule 4) until *plan0* checks that there are no more modules to be erased (Rule 5). Similarly, the modules in *RevToIns* are inserted one at a time until completion (Rules 6 and 7). Finally, it is checked that the required substitutions are performed for all instances; the process is then restarted (Rule 8).

On resumption, the workflow instances which had not entered yet the old region will be processed according to the new plan, while those already inside will not be affected. When exiting either the new or the old region, instances will be processed according to the theory-ids for the subsequent meta-rules. This mechanism also accounts for the case in which no instance was active at the time of suspension, or no instance had still to enter the old region. In these cases, the only effect is the modification of the *execAccord* resource.

The correctness of the described procedure is summarized by the following three propositions, holding true at the completion of the procedure:

- P1:** There is no instance containing *ToErase* in its plan.
- P2:** A plan contains a subpart *P* of *ToErase-ToInsert*, if and only if it has already enabled the modules in *ToErase - P*.
- P3:** A plan contains a module from *ToInsert - ToErase*, if and only if it contains all of *ToInsert*.

4.2 Revisiting the Order Management Example

As an example, we consider the workflow for managing orders as described in [EKR95]. It is assumed that the process starts in the following state:

```
{ workflow(0),execAccord(mrInit,th1), execAccord(mrAp,th1),
  execAccord(mrNotAp,th1), codeText(codingOfRules), reportStatus(nil) }.
```

The rule for creating new instances of the workflow, at the reception of an order *order(Atts)*, characterized by some attributes *Atts*, is:

```
order(Atts) @ workflow(Curr) @ reportStatus(Inst) @ { Id = Curr + 1 }
@ { globalPlan([], [[mrInit(Id)], [mrAp(Id), mrNotAp(Id)], [end(Id)]], NewP) }
@ { report(Inst,Id,NewInst) } <-
workflow(Id) @ reportStatus(NewInst) @ instance(Id,Atts)
@ futureActions(Id,NewP).
```

For brevity, we do not detail the whole plan construction and use instead a constructor *globalPlan*, built on top of *plan*. The initial theory *th1* is composed of rules and meta-rules as illustrated in Fig. 4 (in order to link the rules to the order management example, we print the relevant computational resources in bold face).

Actions are here assumed to modify the attributes. Note that *mrAp* and *mrNotAp* constitute a module of mutually exclusive meta-rules, hence only one of the actions prescribed by this module can succeed. By contrast, rules *shp* and *bil* from the second module must both be executed.

We maintain two enactment policies in the process: a) in the policy for meta-rule modules, completion of a single action from a module, as signaled by the resource *mrDone(Id)*, leads

to enabling actions from the successive module, b) in the policy for rule modules, all actions have to be performed as signaled by the count of resources *ruleDone(Id)* (before rules from the next module are enabled). The enactment mechanism is also specialized to the different identifiers, so that plans for different instances can progress independently. Rule 11 in Fig. 4 allows the updating of the record of active instances. The use of identifiers to restrict rule and metarule action to a single instance may be seen as a case of partial evaluation where an action is specialized prior to its execution [Har91].

<p>1) <code>mrInit(Id) @ instance(Id,Atts) @ futureActions(Id,P) @ execAccord(mrInit,th1) @ {true} @ {plan(P, [[crChk(Id)], [inChk(Id)]],NewP)} <>- instance(Id,Atts) @ futureActions(Id,NewP) @ execAccord(mrInit,th1) @ mrDone(Id).</code></p> <p>2) <code>crChk(Id) @ instance(Id,Atts) @ {creditCheck(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>3) <code>inChk(Id) @ instance(Id,Atts) @ {inventoryCheck(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>4) <code>mrAp(Id) @ instance(Id,Atts) @ futureActions(Id,P) @ execAccord(mrAp,th1) @ {approved(Atts)} @ {plan(P,[[shp(Id),bil(Id)], [arch(Id)]],NewP)} <>- instance(Id,Atts) @ futureActions(Id,NewP) @ execAccord(mrAp,th1) @ mrDone(Id).</code></p> <p>5) <code>mrNotAp(Id) @ instance(Id,Atts) @ futureActions(Id,P) @ execAccord(mrNotAp,th1) @ {not(approved(Atts))} @ {plan(P,[[rej(Id)][ntf(Id)]],NewP)} <>- instance(Id,Atts) @ futureActions(Id,NewP) @ execAccord(mrNotAp,th1) @ mrDone(Id).</code></p> <p>6) <code>shp(Id) @ instance(Id,Atts) @ {shipping(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>7) <code>bil(Id) @ instance(Id,Atts) @ {billing(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>8) <code>arch(Id) @ instance(Id,Atts) @ {archiving(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>9) <code>rej(Id) @ instance(Id,Atts) @ {rejection(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>10) <code>ntf(Id) @ instance(Id,Atts) @ {notify(Atts,NewAtts)} <>- instance(Id,NewAtts) @ ruleDone(Id).</code></p> <p>11) <code>end(Id) @ reportStatus(Inst) @ instance(Id,_) @ futureActions(Id,_) @ {report(LeftInst,Id,Inst)} <>- reportStatus(LeftInst).</code></p>
--

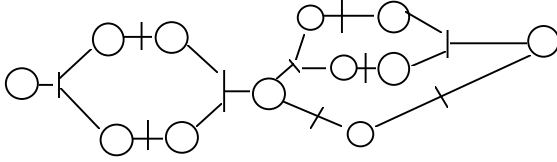
Figure 4. The initial theory for the order management example

Now, consider the case in which one wants to have the rules for inventory and credit checks performed in parallel. A new theory, say *th2*, is introduced, containing the meta-rule

```

mrInit(Id) @ instance(Id,Atts) @ futureActions(Id,P)
@ execAccord(mrInit,th2) @ {true} @ {plan(P, [[crChk(Id), inChk(Id)]],NewP)}
<>- instance(Id,Atts) @ futureActions(Id,NewP)
@ execAccord(mrInit,th2) @ mrDone(Id).

```



The process is notified of the introduction of the new theory by placing in its pool an atom $chgDThry(mrInit, th2)$. After completing the modifications of the $execAccord$ resource and of the affected plans in $futureActions(Id, P)$, the process will contain the atom $execAccord(mrInit, th2)$.

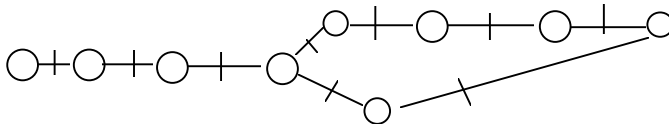
The correctness of the modification of the plans is summarized by specializing the propositions P1 and P2 as follows:

- In1:** There is no instance such that its plan contains $[[crChk], [inChk]]$.
- In2:** If the plan for an instance contains $[inChk]$, this is the first module in the plan.

Since the meta-level identifies the sequence of modules $[[crChk], [inChk]]$ as the value of the variable $ToErase$, the described mechanism ensures that the two propositions hold true.

Now, consider the case when one wants to sequentialize billing and shipping, so as to ensure that shipping is performed after the billing process has completed, while archiving remains the final action for an approved order. This is performed by modifying the meta-rule $mrAp$ (and communicating the resource $chgDThry(mrAp, th2)$ to the process) as follows:

```
mrAp(Id) @ instance(Id,Atts) @ futureActions(Id,P) @ execAccord(mrAp,th2)
@ {approved(Atts)} @ {plan(P,[[bil(Id)], [shp(Id)], [arch(Id)]],NewP)} <-
instance(Id,Atts) @ futureActions(Id,NewP) @ execAccord(mrAp,th2)
@ mrDone(Id).
```



Note that we do not need to create a fresh theory-id for this meta-rule, since it only matters that it differs from any theory-id for any meta-rule with the same name. Moreover, the modification of this meta-rule does not affect the meta-rule for rejection, used in the same module, since we have not modified the computational resource $approved(Th, Atts)$.

The correctness of the modification of the plan is now expressed by the specializing propositions P1 and P3.

- Ap1:** There is no instance such that its plan contains $[shp, bil]$.
- Ap3:** If the plan for an instance contains $[shp]$ or $[bil]$, the plan contains $[[bil], [shp], [arch]]$.

The two propositions hold true due to the identification of $ToErase$ as $[shp, bil]$, and of $ToInsert$ as $[bil] [shp]$.

5 Conclusions

The paper has shown how the distinction between prioritization schemes, managed at the meta-level, and other types of dependencies, managed at the object level, allows a reflective management of dynamic changes in a workflow system.

In the proposed solution, changes involving the sequentialization or the parallelization of workflow tasks are automatically and safely enacted simply by inserting the specification of the changed region of the workflow, and by informing the process that a change has occurred. Thus, no modification of the pre- or post-conditions of any specific activity is required, and the identification of the parts of the plan affected by the change is performed by the agent in charge of the process, while the only action performed by the user is specifying the change.

In future work we plan to exploit reflective abilities, in particular tracing rules, to cache know-how developed during the execution of the process. By implementing *persistent* agents, this know-how could be re-used. Caching of this kind has already been used for applying agent technology to the brokerage of distributed knowledge [ABPS95, ABP96]. Further synergy can derive from combining our approach, based essentially on forward reasoning through production rules, with backward-reasoning goal-oriented approaches to the generation of plans for process executions [GPP96]. Finally, we have not discussed the interaction between multiple workflow agents. Inter-organizational workflow, and workflow in complex geographically distributed organizations [GMPP96, JFJOW96] provide the appropriate scenario for this kind of interaction [TaVe95].

Our proposal for a meta-level management has also some general features which are relevant from the point of view of software design of agent technology, i.e. the fact that the agent behavior is treated uniformly at any level. Hence, the same language is used to manage both the object- and the meta-level; this makes straightforward the application of partial evaluation techniques to lower the burden of meta-interpretation on the execution speed. Problems related to monolingual reflective systems [Har89] are overcome by the adoption of a suitable type system [HL89], where the different components of the meta-level are isolated in instances of well-defined types. Typing enforces the restriction of meta-level representations to object-level components, thus avoiding also the generation of an infinite name set to refer to meta-level components [Sub89]. Moreover, the model supports the option to have only a subset of the object-level attributes reflected at the meta-level. In this way, deliberative behavior is restricted to reified actions, while the existence of non-reified actions ensures the possibility of reactive behavior.

Acknowledgement

We would like to thank Jean-Marc Andreoli, Hervé Gallaire, Antonietta Grasso and the anonymous referees for helpful comments on earlier versions of this paper.

References

- [ABPS95] J.-M. Andreoli, U. M. Borghoff, R. Pareschi, J. H. Schlichter: Constraint Agents for the Information Age. *J. Universal Computer Science* 1:12, 1995, 762-789.

- [ABP96] J.-M. Andreoli, U. M. Borghoff, R. Pareschi: The Constraint-Based Knowledge Broker Model: Semantics, Implementation and Analysis. *J. Symbolic Computation* **21**:4, 1996, 635-667.
- [AFP96] J.-M. Andreoli, S. Freeman, R. Pareschi: The Coordination Language Facility: Coordination of Distributed Objects. *Theory and Practice of Object Systems* **2**:2, 1996, 1-18.
- [AGP94] J.-M. Andreoli, H. Gallaire, R. Pareschi: Rule-based Object Coordination, in *ECOOOP'94 Workshop on Object-based Models and Languages for Concurrent Systems*, Springer Verlag, LNCS 924, 1994.
- [AMP96] J.-M. Andreoli, J.-L. Meunier, D. Pagani: Process Enactement and Coordination, in *Proc. EWPST'95*, Nancy, France, 1996.
- [BMP94] P. Bottoni, P. Mussio, M. Protti: Metareasoning in the Determination of Image Interpretation Strategies, *Pattern Recognition Letters* **15**, 1994, 177-190.
- [BR93] J. Blythe, W. S. Reilly: Integrating Reactive and Deliberative Planning for Agents, *Carnegie Mellon Univ. Tech. Report CMU-CS-93-155*, May 1993.
- [BT95] F. Brazier, J. Treur: Formal Specification of Reflective Agents, in *IJCAI '95 Workshop on Reflection*, M. Ibrahim, ed. Montreal, 1995, pp. 103-112. Also in *KEML '96*, Paris, 1996.
- [CoopIS96] G. De Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, K. Pohl, J. Schmidt, C. Woo, E. Yu: Cooperative Information Systems: A Manifesto. *Technical report, University of Hamburg*, April 1996.
- [DW96] G. De Michelis, C. Woo: personal communication.
- [EKR95] C. Ellis, K. Kedara, G. Rozenberg: Dynamic Change Within Workflow Systems, *Proc. COOCS '95*, ACM Press, 1995, pp. 10-21.
- [GMPP96] A. Grasso, J.-L. Meunier, D. Pagani, R. Pareschi: Distributed Coordination and Workflow on the World-wide Web. *J. of CSCW, special issue on CSCW and the World-wide Web*, to appear.
- [GPP96] N. S. Glance, D. S. Pagani, R. Pareschi: Generalized Process Structure Grammars (GPSG) for Flexible Representations of Work, *Proc. CSCW '96*, Cambridge, Massachusetts, ACM Press, 1996.
- [HC93] M. Hammer, J. Champy: Reengineering the Corporation: a Manifesto for Business Revolution, HarperCollins, 1993.
- [Har89] F. van Harmelen: A Classification of Meta-level Architectures, in *Meta-Programming in Logic Programming*, H. Abramson, M. H. Rogers eds., MIT Press, 1989, pp. 104-122.
- [Har91] F. van Harmelen,: *Meta-level Inference Systems*, Pitman, 1991.
- [HL89] P. M. Hill, J. W. Lloyd: Analysis of Meta-Programs, in *Meta-Programming in Logic Programming*, H. Abramson, M. H. Rogers eds., MIT Press, 1989, pp.23-51.
- [JFJOW96] N. R. Jennings, P. Faratin, M. J. Johnson, P. O. O'Brien, M. E. Wiegand: Using Intelligent Agents to Manage Business Processes, *Proc. PAAM'96*, London, UK, PAP, 1996.
- [Kow96] R. A. Kowalski: Using Meta-logic to Reconcile Reactive with Rational Agents, *Proc. PAAM'96*, London, UK, PAP, 1996.
- [McH95] D. McDermott, J. Hendler eds.: *Artificial Intelligence, special volume on Planning and Scheduling* **76**, 1995.

- [Osa94] E.-I. Osawa: A Meta-level Coordination Strategy for Reactive Cooperative Planning, *Tech. Rep. SCSL-TR-94-014*, SONY Computer Science Laboratory, Tokyo, 1994.
- [Sub89] V. S. Subrahmanian: Theory of Metalogic Programming, in *Meta-Programming in Logic Programming*, H. Abramson, M. H. Rogers eds., MIT Press, 1989, pp. 65-101.
- [TaVe95] J. Tang, J. Veijalainen: Transaction-oriented Workflow Concepts in Inter-organizational Environments, *Proc. CIKM'95*, Baltimore, Maryland, ACM Press, 1995.