An Architecture for a Rational Reactive Agent

Pierangelo Dell'Acqua*†, Mattias Engberg*, and Luís Moniz Pereira†

* Department of Science and Technology - ITN
Linköping University, 601 74 Norrköping, Sweden
{pier,maten}@itn.liu.se

† Centro de Inteligência Artificial - CENTRIA
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
lmp@di.fct.unl.pt

Abstract. We present an architecture for a rational, reactive agent and describe its implementation. The paper addresses issues raised by the interaction of the rational and reactive behaviour of the agent, and its updating mechanism. We relate it with the work of others.

1 Introduction

In previous work we defined a logical formalization of a framework for multi-agent systems and we defined its semantics [13]. In such a framework, we embedded a flexible and powerful kind of agent that are rational, reactive, abductive, able to prefer and they can update the knowledge base of other agents.

The knowledge state of each agent is represented by an an abductive logic program in which it is possible to express rules, integrity constraints, active rules and priorities among rules. This allows the agents to reason, to react to the environment, to prefer among several alternatives, to update both beliefs and reactions, and to abduce hypotheses to explain observations. We defined a declarative and procedural semantics of this kind of agent [4, 14]. These agents are then embedded into a multi-agent system in such a way that the only form of interaction among them is based on the notions of project and update.

The semantics of the agents depends on the query that each agent has to prove at a certain moment in time. In fact, in proving a query G the agent may abduce hypotheses that explain G, and in turn these hypotheses may trigger active rules, and so on. Hypotheses abduced in proving G are not permanent knowledge, rather they only hold during the proof of G. To make them permanent knowledge, an agent can issue an internal project and update its own knowledge base with those hypotheses.

Our approach to agents is an enabling technology for sophisticated Web applications due to their deliberative, reactive and updating capabilities. See [15] for a discussion on the use of preference reasoning in Web applications, and [1] for a discussion on the use of Logic Programming for the Semantic Web.

In this paper we present an architecture for such a kind of agents and outline its implementation. Our aim is to have a simple architecture and to test its behaviour in different application domains. In particular, we are interested in testing the interaction between the rational/deliberative (D) and reactive (R) behaviour of the agent, and its updating mechanism (U). In fact, the interaction of these components raises a number of issues that are addressed in Section 3.

D-R Can (should?) the deliberative behaviour of the agent influence the reactive behaviour? and vice versa? This is an important issue. For example, if the deliberative mechanism of the agent is performing some planning, and some exogenous event occurs, then the reactive mechanism of the agent can detect the anomaly, suspend the planning and require to start a replanning phase.

D-U If α has a goal to prove and contemporarily some updates to consider, which of the two tasks shall α perform first? The policy of giving priority to goals may decrease the agent performance when a quick reaction is required. In contrast, prioritizing updates can cause unwanted delays of the proof of a goal only to consider unrelevant updates.

Another issue concerns the behaviour of the agent when is proving a goal G (i.e., it is deliberative) and it receives an update. Should α complete the execution of G, and then consider the update? or instead should it suspend the execution of G, consider the update and then relaunch G ?

R-U If α receives several updates, should α consider each update and then trigger its active rules? or should it consider all the updates and only then trigger the active rules?

Two other interesting features of our architecture are the ability (i) to declaratively interrupt/suspend the execution of a query, and (ii) to declaratively self-modify its control parameters. These features are important for example in the context of reactive planning when the agents are situated in dynamic environments. Here, there is often a need to suspend the current execution of a plan due to unexpected exogenous events, and to replan accordingly. Also, the agent must be able to tune its behaviour according to the environment's conditions. For example, in a dangerous situation the agent may decide to become more reactive to quickly respond to changes of the environment. In contrast, in other situations the agent is required to be more deliberative, like when it enters into a planning phase.

The remainder of the paper is structured as follows. Section 2 presents the logical framework, the notion of abductive agents, and the agent cycle. The architecture of the agent is presented in Section 3. Finally, Sections 6 and 7 compare our work with the literature and presents future lines of research work.

2 Preliminaries

In this section we present the language of agents and our conception of abductive agents. The reader is referred to [14] for more details and examples on the language, including its declarative semantics, [4] for the procedural semantics, and [5] for the declarative semantics of update and preference reasoning.

2.1 Logic Programming Framework

In this exposition, we syntactically represent the theories of agents as propositional Horn theories. In particular, we represent default negation *not A* as a standard propositional variable. Propositional variables whose names do not begin with "*not*" and do not contain the symbols ":", "÷" and "<" are called *domain atoms.* For each domain atom A we assume a complementary propositional variable of the form *not A*. Domain atoms and negated domain atoms are called *domain literals.*

Communication is a form of interaction among agents. The aim of an agent β when communicating a message C to an agent α, is to make α update its current theory with C (i.e., to make α accept some desired mental state). In turn, when α receives the message C from β, it is up to α whether or not incorporate C. This form of communication is formalized through the notion of projects and updates. Propositional variables of the form $\alpha{:}C$ (where C is defined below) are called *projects.* $\alpha{:}C$ denotes the intention (of some agent β) of proposing the updating of the theory of agent α with C. Projects can be negated. A negated project of the form *not* $\alpha{:}C$ denotes the intention of the agent of not proposing the updating of the theory of agent α with C. Projects and negated projects are generically called *project literals.*

Propositional variables of the form $\beta \div C$ are called *updates.* $\beta \div C$ denotes an update with C in the current theory (of some agent α), that has been proposed by β. Updates can be negated. A negated update of the form *not* $\beta{\div}C$ in the theory of some agent α indicates that agent β does not have the intention to update the theory of agent α with C. Updates and negated updates are called *update literals.*

Preference information is used along with incomplete knowledge. In such a setting, due to the incompleteness of the knowledge, several models of a program may be possible. Preference reasoning is enacted by choosing among those possible models, through the expression of priorities amongst the rules of the program. Preference information is formalized through the notion of priority atoms. Propositional variables of the form $n_r < n_u$ are called *priority atoms.* $n_r < n_u$ means that rule r (whose name is n_r) is preferred to rule u (whose name is n_u). Priority atoms can be negated. *not* $n_r < n_u$ means that rule r is not preferred to rule u. Priority atoms and negated priority atoms are called *priority literals.*

Domain atoms, projects, updates, and priority atoms are generically called *atoms.* Domain literals, project literals, update literals, and priority literals are generically called *literals.*

Definition 1. A *generalized rule* is a rule of the form $L_0 \leftarrow L_1, \ldots, L_n$ with $n \geq 0$ where every L_i $(0 \leq i \leq n)$ is a literal.

Definition 2. A *domain rule* is a generalized rule $L_0 \leftarrow L_1, \ldots, L_n$ whose head L_0 is a domain literal distinct from *false* and *not false*, and every literal L_i $(1 \leq i \leq n)$ is a domain literal or an update literal.

Definition 3. An *integrity constraint* is a generalized rule whose head is the literal *false* or *not false*.

Integrity constraints are rules that enforce some condition on states, and they take the form of denials. To make integrity constraints updatable, we allow the domain literal *not false* to occur in the head of an integrity constraint. For example, updating the theory of an agent α with $\textit{not false} \leftarrow \textit{relaxConstraints}$ has the effect to turn off the integrity constraints of α if *relaxConstraints* holds. Note that the body of an integrity constraint can contain any literal. The following definition introduces rules that are executed bottom-up. To emphasize this aspect we employ a different notation for them.

Definition 4. An *active rule* is a generalized rule whose head Z is a project literal and every literal L_i $(1 \leq i \leq n)$ in its body is a domain literal or an update literal. We write active rules as $L_1, \ldots, L_n \Rightarrow Z$.

Active rules can modify the current state, to produce a new state, when triggered. If the body $L_1, \ldots, L_n$ of the active rule is satisfied, then the project (fluent) Z can be selected and executed. The head of an active rule is a project, either internal or external. An *internal project* operates on the state of the agent itself (self-update), e.g., if an agent gets an observation, then it updates its knowledge. *External projects* instead are performed on other agents, e.g., when an agent wants to update the theory of another agent.

To express preference information in logic programs we introduce the notion of priority rule.

Definition 5. A *priority rule* is a generalized rule $L_0 \leftarrow L_1, \ldots, L_n$ whose head L_0 is a priority literal and every L_i $(1 \leq i \leq n)$ is a domain literal, an update literal, or a priority literal.

Priority rules are also subject to updating.

Definition 6. A *query* takes the form $?\!-\ L_1, \ldots, L_n$ with $n \geq 0$, where every L_i $(1 \leq i \leq n)$ is a domain literal, an update literal, or priority literal.

We assume that for every project $\alpha{:}C$, C is either a domain rule, an integrity constraint, an active rule, a priority rule or a query. Thus, a project can take one of the forms:

$$\begin{array}{ll}
\alpha{:}(L_0 \leftarrow L_1, \ldots, L_n) & \alpha{:}(L_1, \ldots, L_n \Rightarrow Z) \\
\alpha{:}(\textit{false} \leftarrow L_1, \ldots, L_n, Z_1, \ldots, Z_m) & \alpha{:}(?\!-L_1, \ldots, L_n) \\
\alpha{:}(\textit{not false} \leftarrow L_1, \ldots, L_n, Z_1, \ldots, Z_m) &
\end{array}$$

2.2 Abductive Agents

The knowledge of an agent can dynamically evolve when it receives new knowledge or when it abduces new hypotheses to explain observations. The new knowledge is presented in the form of an updating program, and the new hypotheses in the form of a (finite) set of domain atoms (abducibles).

An *updating program* U is a finite set of updates. An updating program contains the updates that will be performed on the current knowledge state of the agent. As negated updates are not performed by any agent, negated updates cannot occur in any updating program. To characterize the evolution of the knowledge of an agent we need to introduce the notion of sequence of updating programs. In the remaining, let $S = \{1, \ldots, s, \ldots\}$ be a set of natural numbers. We call the elements $i \in S \cup \{0\}$ *states*. A *sequence of updating programs* $\mathcal{U} = \{U^s \mid s \in S\}$ is a set of updating programs U^s superscripted by the states $s \in S$.

Let $\mathcal{A}$ be a set of domain atoms distinct from *false*. We call the domain atoms in $\mathcal{A}$ *abducibles*. Abducibles can be thought of as hypotheses that can be used to extend the current theory of the agent in order to provide an "explanation" for given queries. Explanations are required to meet all the integrity constraints. Abducibles may also be defined by domain rules as the result of a self-update to adopt an abducible as a fact rule.

Definition 7. Let $s \in S \cup \{0\}$ be a state. An *agent α at state s*, written as Ψ^s_α, is a pair $(\mathcal{A}, \mathcal{U})$, where $\mathcal{A}$ is the set of abducibles and $\mathcal{U}$ is a sequence of updating programs $\{U^1, \ldots, U^s\}$. If $s = 0$, then $\mathcal{U} = \{\}$.

An agent α at state 0 is defined by a set of abducibles $\mathcal{A}$ and an empty sequence of updating programs, that is $\Psi^0_\alpha = (\mathcal{A}, \{\})$. At state 1 α is defined by $(\mathcal{A}, \{U^1\})$, where U^1 is the updating program containing all the updates that α has received at state 0 either from other agents or as self-updates. In general, an agent α at state s is defined by $\Psi^s_\alpha = (\mathcal{A}, \{U^1, \ldots, U^s\})$, where each U^s is the updating program containing the updates that α has received at state $s - 1$.

Example 1. Consider a situation where an agent Elizabeth (represented by e) is told by Maria (represented by m) to abduce fire to explain the observation that there is smoke, and to sound the alarm to the fire brigade (represented by f) in case of fire. Later on, Elizabeth is told by Robert (represented by r) that a possible explanation for the presence of smoke is that John is smoking, and in such a case she should scream at him. The theory of Elizabeth can be formalized as $\Psi^2_e = \{\mathcal{A}, \mathcal{U}\}$ where $\mathcal{A} = \{\mathit{fire}, \mathit{smoking}(\mathit{john})\}$ and $\mathcal{U} = \{U^1, U^2\}$:

$$U^1 = \begin{Bmatrix} m{\div}(\mathit{smoke} \leftarrow \mathit{fire}) \\ m{\div}(\mathit{fire} \Rightarrow f{:}\mathit{alarm}) \end{Bmatrix} \qquad U^2 = \begin{Bmatrix} r{\div}(\mathit{smoke} \leftarrow \mathit{smoking}(\mathit{john})) \\ r{\div}(\mathit{smoking}(\mathit{john}) \Rightarrow j{:}\mathit{scream}) \end{Bmatrix}$$

Suppose that at state 2 there is smoke. Then, Elizabeth has two hypotheses equally plausible to explain it: *fire* and *smoking*(*john*). Things change if later on Elizabeth is told by Maria to prefer the hypothesis *fire* to *smoking*(*john*) when John is at the pub to explain the presence of smoke:

$$U^3 = \begin{Bmatrix} m{\div}(\mathit{fire} < \mathit{smoking}(\mathit{john}) \leftarrow \mathit{pub}(\mathit{john})) \\ m{\div}(\mathit{pub}(\mathit{john})) \end{Bmatrix}$$

Now, Elizabeth has one preferred hypothesis to explain smoke, i.e., *fire*.

2.3 Agent Cycle

In this section we briefly sketch the behaviour of an agent. This is carried out from the perspective of an agent α. The basic "engine" of α is an abductive logic programming proof procedure, executed via the cycle represented in Fig. 1.

Cycle(α,s,Ψ^s_α,G), where $\Psi^s_\alpha = (\mathcal{A},\mathcal{U})$ and $\mathcal{U} = \{U^1, \ldots, U^s\}$.

1. Observe and record any input in the updating program U^{s+1}.
2. Select a query $?{-}g$ in $G \cup \mathit{Queries}(\alpha, U^{s+1})$ and execute $?{-}g$ wrt. $P = \Gamma(s+1, \mathcal{U} \cup \{U^{s+1}\})$. Let $U^{s+2} = \{\alpha \div \mathit{ans}(g, g\theta, \mathit{La})\}$ if g is provable with substitution θ and abduced hypotheses $\mathit{La} \subseteq \mathcal{A}$, otherwise (i.e., if g is not provable) let $U^{s+2} = \{\alpha \div \mathit{ans}(g, \mathit{fail}, [\,])\}$ and $\mathit{La} = [\,]$.
3. Execute all the projects in $\mathit{ExecProj}(\mathit{La}, Q, s+2)$, where $Q = \Gamma(s+2, \mathcal{U} \cup \{U^{s+1}, U^{s+2}\})$.
4. Cycle with (α,$s+2$,Ψ^{s+2}_α,G'), where $\Psi^{s+2}_\alpha = (\mathcal{A}, \mathcal{U} \cup \{U^{s+1}, U^{s+2}\})$ and $G' = G \cup \mathit{Queries}(\alpha, U^{s+1}) - \{?{-}g\}$.

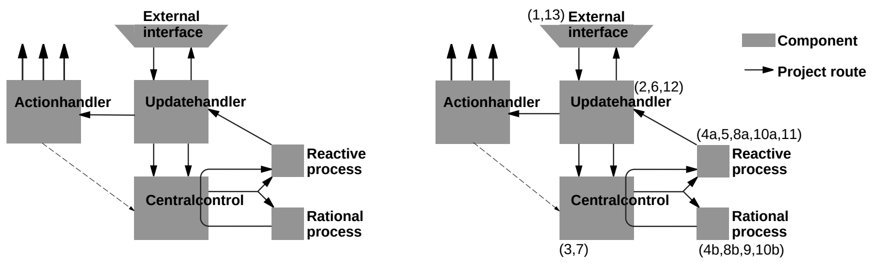
Fig. 1. *The agent cycle*

Step 1: The cycle of an agent α starts at state s by observing any inputs, i.e. updates from other agents or from the environment, and by recording them in updating program U^{s+1}.

Step 2: A query $?{-}g$ is selected from $G \cup \mathit{Queries}(\alpha, U^{s+1})$, where G is the set of queries still to be executed and $\mathit{Queries}(\alpha, U^{s+1}) = \{?{-}g \mid \alpha{\div}(?{-}g) \in U^{s+1}\}$ is the set of queries requested to be proved by α. Then, g is executed with respect to the generalized logic program P obtained via a syntactic transformation Γ presented in [4]. Basically, Γ takes into consideration the entire sequence of updating programs $\mathcal{U}$ at state $s+1$ and codes it into an object level generalized logic program P. Thus, any abductive proof procedure, such as ABDUAL [7], can be used in proving from P. The answer $\mathit{ans}(g, g\theta, \mathit{La})$ of the query $?{-}g$ is then recorded in the updating program U^{s+2}. Note that only the queries issued by the agent α itself are executed. The queries issued by other agents are treated as normal updates[1].

Step 3: The set $\mathit{ExecProj}(\mathit{La}, Q, s+2)$ of projects that are executable at state $s+2$ is computed and all the projects executed. A project is executable at state $s+2$ if it is defined by an active rule in P whose body contains literals that are true at $s+2$ or are abducibles in La. If an executable project takes the form $\beta : C$ (meaning that agent α intends to update the theory of agent β with C),

[1] This way α retains control on deciding which queries (requested by other agents) to execute. For example, the theory of α may contain the active rule: $\beta{\div}(?{-}g), \mathit{Cond} \Rightarrow \alpha{:}(?{-}g)$ which states that if α has been requested to prove a query $?{-}g$ by β and some condition Cond holds, then α will issue the internal project to prove the query $?{-}g$.



(a) Agent base structure (b) Project flows

Fig. 2.

then (once executed) the update $\alpha \div C$ will be available as input to the cycle of the agent β.

Step 4: Finally, the agent cycles with state $s+2$, by incorporating the updating programs U^{s+1} and U^{s+2} into $\mathcal{U}$, and with the new list G' of queries.

Initially, the cycle of α is Cycle(α,0,Ψ^0_α,{}) with $\Psi^0_\alpha = (\mathcal{A}, \{\})$.

3 Agent Architecture

The agent framework is implemented as follows: its logical parts (e.g., logical reasoning, preferring, updating, etc.) are implemented in XSB Prolog [11], while its non-logical parts (e.g., agent communication, user interface, etc.) are implemented in Java. We then employ InterProlog [10] to interface Java and XSB Prolog. The next subsections present an overview of the architecture, its components and the flow of projects (for more details consult [16]).

3.1 Overview

The architecture consists of six components, as illustrated in Fig. 2. Each one harbours its own specific task and is implemented in Java to enhance flexibility. Since every component is implemented via a Java thread, the components can run concurrently. Therefore, the behaviour of the agent is not sequential like in Kowalski and Sadri's agent cycle [18]. Rather, the system has the ability to execute at the same time, asynchronously, several tasks differently located in the agent cycle (defined in Section 2.3) . The agent in our implementation is therefore enabled to exhibit both rational and reactive behaviours concurrently. The architecture consists of the following components that form the agent's base structure:

– The *central control*: it controls the behaviour of the entire architecture via a number of control parameters. For example, it determines when a query or an update should be sent to the reactive and/or to the rational processes.

– The *reactive* and the *rational processes*: they characterize the reactive and rational behaviours of the agent, respectively. The rational and reactive processes are implemented by two corresponding XSB Prolog processes whose knowledge bases are kept identical[2].
– The *update handler*: it sorts and forwards the information received from the reactive process and from the surrounding environment.
– The *action handler*: it handles the different tasks (actions) the agent wants executed. It has the ability to affect the environment.
– The *external interface*: it handles the communication between the agent and it's environment, including other agents.

The solid arrows between the components in Fig. 2 (a) illustrate all the possible paths of projects. The dashed arrow indicates that the action handler can change the value of some parameters in the central control. The three arrows exiting the action handler indicate that it can perform several actions to affect the environment,which includes other agents, depending on the situation and tasks at hand.

Between every two components connected by a solid arrow there exists a queue. This is necessary for handling asynchronicity between the threads of the connected components when exchanging information. The queues are not shown in Fig. 2 (a). If a component receives a large workload, the queue collects the information to be exchanged and awaits till the receiving component can handle the new incoming information.

3.2 Project Flow

This section illustrates the flow of projects in the agent architecture. This is narrated from the perspective of an agent α, depicted in Fig. 2 (b). Suppose that an agent β asks agent α to prove a query $?{-}g$ via the external project $\alpha : ?{-}g$. At the next state, the external interface of α receives (step 1) the update $\beta \div ?{-}g$, indicating that β has requested α to prove g.[3] In turn, this update is sent to the update handler (step 2) whose task is to sort the different types of updates. For example, if the update contains an action to be performed, then the update will be sent to the action handler, otherwise to the central control. Since the update contains a query requested by another agent, the update handler sends $\beta \div ?{-}g$ to central control (step 3). Now, central control updates the database of the reactive process (step 4a) and of the rational process (step 4b) with $\beta \div ?{-}g$. Note that when α receives a request to prove a query $?{-}g$ from another agent β (via the update $\beta \div ?{-}g$ in the theory of α), g is not proven directly. Instead, α has the ability to decide whether or not to prove g. Indeed, α proves g only

[2] The use of identical copies of the knowledge base is required by the present unavailability of XSB Prolog threads sharing the same knowledge program. This will be remedied in the near future by ongoing XSB implementation work by colleagues.

[3] An asynchronous transition rule system that characterizes the interactions among agents is presented in [13].

if α itself posts this request via the internal project $\alpha : ?{-}g$. Suppose that the knowledge base of the reactive process contains an active rule of the form:

$$\beta \div ?{-}g \Rightarrow \alpha : ?{-}g$$

saying that, if α receives a request to prove a query $?{-}g$ from an agent β, then α assumes the project to prove g (i.e., $\alpha : ?{-}g$). At the next state of α, the internal project $\alpha : ?{-}g$ will be executed (step 5). This implies that the update handler will receive (step 6) the update $\alpha \div ?{-}g$ from the reactive process, which in turn sends it to central control (step 7). Central control updates again the knowledge bases of the reactive and rational process with $\alpha \div ?{-}g$ (steps 8a and 8b). Since the query is issued by the agent itself, central control will launch the query $?{-}g$ (step 9). Notice that the updating and reasoning phases are considered to be atomic. That is, it is not possible while proving a query to make an update of the knowledge base (unless by relaunching the query itself). If g is provable with substitution θ and list of abduced hypotheses La, central control will update the knowledge bases of the reactive and rational process with the answer $\alpha \div \mathit{ans}(g, g\theta, \mathit{La})$ of the query $?{-}g$ (steps 10a and 10b). This allows α to eventually send the answer back to β in case its knowledge base contains an active rule of the form:

$$\beta \div ?{-}g, \alpha \div \mathit{ans}(g, \mathit{ANS}, \mathit{ABD}) \Rightarrow \beta : \mathit{ans}(g, \mathit{ANS}, \mathit{ABD})$$

When this active rule triggers (step 11), the update handler receives the project $\beta : \mathit{ans}(g, \mathit{ANS}, \mathit{ABD})$ (step 12) which is sent to the external interface (step 13) it being an external project.

3.3 Rational Process

The rational process models the rational ability of agents. It is implemented by an XSB Prolog process. If an agent α is at state s and $\Psi^s_\alpha = \{\mathcal{A}, \mathcal{U}\}$, then the knowledge base of the rational process is the generalized logic program $P = \Gamma(s, \mathcal{U})$ obtained via a syntactic transformation Γ presented in [4]. The ABDUAL procedure [6, 7] is employed wrt. P to compute well-founded abductive answers to goals. (This procedure is capable of computing generalized stable models as well.) The rational process receives the queries to be proved from central control. When the rational process proves a query $?{-}g$ with substitution θ and list of abduced hypotheses La, it returns the answer as $\mathit{ans}(g, g\theta, \mathit{La})$ to central control which in turn updates the knowledge bases of the rational and reactive processes with $\alpha \div \mathit{ans}(g, g\theta, \mathit{La})$.

3.4 Reactive Process

The reactive process models the reactive behaviour of the agent. It is implemented by an XSB Prolog process whose underlying theory is the generalized logic program $P = \Gamma(s, \mathcal{U})$.[4] The task of the reactive process is to trigger any

[4] Recall that the rational and reactive process have the same knowledge base.

active rule whose body is satisfied by P at the current state s. Triggering an active rule means executing the project occurring in its head. To test which active rules can be triggered central control launches the query $?-exec(La, L)$ to the reactive process, where La is the list of hypotheses abduced by the rational process at state s. Executing $?-exec(La, L)$ returns a list L of executable projects to central control. Basically, the call to $exec(\ldots)$ checks the active rules whose body holds in $\Gamma(s, \mathcal{U})$ and returns their heads in the list L. $exec(\ldots)$ is defined by a generalized logic program implementing the behaviour of $ExecProj(\ldots)$. The definition of $exec(\ldots)$ can vary depending on the kind of agent behaviour we want to characterize. The intuition is that P can have several well-founded abductive models, each of which may trigger distinct active rules (and therefore each model will contain distinct projects). If we want to characterize a cautious behaviour for agents, then the executable projects are those occurring in every model. An alternative definition for would be to execute all the projects that occur in some model of P: brave behaviour.

The hypotheses in La assumed to prove a query g remain abduced only during the current cycle of the agent. They can be made permanent knowledge through an internal update. For instance, the active rule $a \Rightarrow \alpha : a$ where $a \in \mathcal{A}$, states to execute the internal update $\alpha : a$ when the hypothesis a is abduced.

Distinguished projects are those defining the predefined predicates *sendInterrupt*, *doAct*, and *stateModify*. For example, the active rule in the theory of an agent α:

$$\beta \div urgentRequest(G) \Rightarrow \alpha : sendIterrupt(G, true)$$

instructs α to interrupt the execution of its current query G', to launch the query G, and to resume the execution of G' once G is proved (since the flag of *sendInterrupt* is true), just in case an agent β has urgently requested it. The active rule:

$$meeting \Rightarrow \alpha : doAct(displayMessage)$$

instructs α to graphically display a message to remind itself of the meeting. An interesting feature of our architecture is that an agent can declaratively change the value of its control parameters. This is achieved through the predicate *stateModify*. The active rule

$$alarmState \Rightarrow \alpha : stateModify(reactiveDelayTime, 100)$$

once triggered allows α to assign the value 100 to the control parameter *reactiveDelayTime*, in case a dangerous situation supervenes. *ReactiveDelayTime* is a parameter in central control that defines the time interval between two distinct tests of which active rules of α that can be triggered. Basically, its value defines the level of reactiveness of α. The list of parameters that control the agent behaviour can be found in [16].

The ability to declaratively interrupt the rational process and to modify the values of the control parameters of the agent architecture is essential to tailor the global behaviour of the agent to its actual needs (cf. [3] for a discussion).

4 Update Handler

The update handler collects both the executable projects coming from the reactive process and the updates coming from the external interface. The task of the update handler is to sort out the projects and updates and to send them to the right destination. Note that to maintain the semantics of updates, all the executable projects of an agent α must be executed at the same time.

A project can be either (i) an external project, (ii) an internal project, or (iii) an internal project containing a predefined atom *sendInterrupt*, *doAct*, or *stateModify*. Updates and external projects are sent to central control and to the external interface, respectively. Regarding internal projects we distinguish among three cases. Those not containing any predefined atom are sent to central control. Those containing the atoms *doAct* or *stateModify* are sent to the action handler, while those containing the atom *sendInterrupt* are sent to central control via a prioritized queue in order to make them executed before regular updates.

5 Central Control

Central control handles all the other components of the architecture by means if its control parameters. Central control has also the ability to interrupt/suspend the execution of a query. To do so, it sends an interrupt $\mathit{sendInterrupt}(g', F)$ to the rational process commanding it to interrupt the execution of the current query g and to execute the query g'. The flag F indicates whether or not the execution of g must be resumed after the proof of g' is completed. The interruption/suspension commands are implemented by XSB Prolog, and can originate externally. They call upon an interrupt handler, whose behaviour is user definable by logic program rules. Its workings are similar in spirit to break interrupts during debugging and, like them, can be embedded.

It comprises two incoming queues: a prioritized queue with updates containing interrupts, and a queue with normal updates. Central control works in cycles. First, it executes the updates with interrupts one by one until the queue is empty. To do so, it suspends the execution of the current query and launches the query associated with the interrupt. Once terminated, the current query can be relaunched or resumed, depending on the flag of the interrupt. Since interrupts cannot contain updates (only queries), the knowledge base remains unchanged after performing an interrupt and therefore the execution of the suspended goal can be resumed. Then the queue with normal updates is given attention by selecting the next update in the queue and by updating the knowledge bases of the rational and reactive process. If it is an internal update of the form $\alpha \div ?{-}g$ (indicating that the agent has posted an update to itself by requiring the execution of a query $?{-}g$), then central control launches the query $?{-}g$ to the rational process, it collects its answer $\mathit{ans}(g, g\theta, La)$ and it updates the knowledge bases of the rational and reactive process with it. Then, it launches the query $\mathit{exec}(La, L)$ to the reactive process and sends the list L of executable projects to the update handler. Finally, central control modifies the value of its control parameters if so requested by the action handler through a *stateModify* command, and cycles again.

Example 2. Consider the situation presented in Example 1. Suppose that at state 3 Elizabeth is proving the query $?- smoke$. This task is carried out as follows. First, central control posts the query to the rational process which proves it by abducing *fire*.[5] After receiving the answer $ans(smoke, smoke, [fire])$ from the rational process, central control posts the update $e \div ans(smoke, smoke, [fire])$ to both the rational and reactive process, and launches the query $?- exec([fire], L)$ to the reactive process. Since the active rule $fire \Rightarrow f{:}alarm$ in U^1 is triggered, the query succeeds with substitution $L = [f{:}alarm]$. Finally, central control posts the list L of executable projects to the update handler to sort them out and send them to the final destination.

6 Related Work

The use of computational logic for modelling multi-agent systems has been widely investigated (e.g., see [2, 22] for a roadmap). One approach close to our own is the agent-based architecture proposed by Kowalski and Sadri [18], which aims at reconciling rationality and reactivity. Agents are logic programs that continuously perform the *observe-think-act* cycle. The thinking or deliberative component consists in explaining the observations, generating actions in response to observations, and planning to achieve its goals. The reacting component is defined via a proof-procedure which exploits integrity constraints.

Another approach close to ours is the Dali multi-agent system proposed by Costantini [12]. Dali is a language and environment for developing logic agents and multi-agent systems. Dali agents are rational agents that are capable of reactive and proactive behaviour. These abilities rely on and are implemented over the notion of event.

The Impact system [8] represents the beliefs of an agent by a logic-based program and integrity constraints. Agents are equipped with an action base describing the actions they can perform. Rules in the program generalize condition-action rules by including deontic modalities to indicate, for instance, that actions are permitted or forbidden. Integrity constraints, as in our approach, specify situations that cannot arise and actions that cannot be performed concurrently. Alternative actions can be executed in reaction to messages.

3APL [9] is a programming language for implementing cognitive agents. It provides programming constructs for modelling agent's beliefs and goals, and a number of basic capabilities to revise them.

The BDI approach [21] is a logic-based formalism to represent agents. In it, an agent is characterized by its *beliefs*, *desires* (i.e., objectives it aims at), and *intentions* (i.e., plans it commits to). Beliefs, desires, and intentions are represented via modal operators with a possible world semantics.

[5] Recall that due to the preference expressed on abducibles by Maria, *fire* is preferred to *smoking*(*john*) when John is at the pub.

Another logic-based formalism proposed for representing agents is Agent0 [23]. In this approach, an agent is characterized by its *beliefs* and *commitments*. Commitments are production rules that can refer to non-consecutive states of the world in which the agent operates. Both the BDI and the Agent0 approach use logic as a tool for representing agents, but rely upon a non-logic-based execution model. This causes a wide gap between theory and practice in these approaches [22].

An example of a BDI architecture is Interrap [20]. It is a hybrid architecture consisting of two vertical layers: one containing layers of knowledge bases, the other containing various control components that interact with the knowledge bases at their level. The lowest control component is the *world interface* that manages the interactions between the agent and its environment. Above the world interface there is the *behaviour-based component*, whose task it is to model the basic reactive capabilities of the agent. Above this component there is a *local planning component* able to generate single-agent plans in response to requests from the behaviour-based component. On top of the control layer there is a *social planning component*. The latter is able to satisfy the goals of several agents by generating their joint plans. A formal foundation of the Interrap architecture is presented in [17].

Sloman et al. [24, 25] proposed a hybrid agent whose architecture consists of three layers: the reactive, the deliberative, and the meta-management layer. The layers operate concurrently and influence each other. The deliberative layer, for example, can be automatically invoked to reschedule tasks that the reactive layer cannot cope with. The *meta-management* (reflective) layer provides the agent with capabilities of self-monitoring, self-evaluation, and self-control.

A hybrid architecture, named Minerva, that includes, among others, deliberative and reactive behaviour was proposed by Leite et al. [19]. This architecture consists of several components sharing a common knowledge base and performing various tasks, like deliberation, reactiveness, planning, etc. All the architectural components share a common representation mechanism to capture knowledge and state transitions.

7 Conclusion and Future Work

We have presented an architecture for a type of agent that is able to reason, to react to the environment, to update its knowledge to model the dynamic aspects of the world where it is situated, is able to prefer, and also to abduce hypotheses to explain its observations.

Our aim is to develop a simple architecture and to test its behaviour in different application domains. Currently we are carrying out on an experimental implementation to test our agent in a simple Web site management application.

An interesting line of research is to investigate how to formally describe the behaviour of our agent architecture. This will allow us (i) to formally compare our architecture with other hybrid architectures proposed in the literature, (ii) to verify its logical properties, and (iii) to analyze and expand its architectural design to accommodate other components. Currently, we are investigating whether it is possible to employ/extend the COOP calculus [17] to achieve this goal. In the future, we plan to add more modules to the basic structure, such as a planner, a learning module, etc., modelling other rational abilities and their functionalities, and to test their interactions within the basic architecture.

The design of our architecture allows an agent to tune its behaviour to the environmental needs. Thus, the agent has self-monitoring and self-control capabilities. Regarding self-evaluation, we are exploring the benefits of integrating a meta-management component into the architecture along the lines proposed by Sloman [24]. Our aim is to design an approach to agents where self-evaluation capabilities allow an agent to dynamically reconfigure its architectural components.

At the moment, our agent implementation runs locally on the host machine, but the entire application can be made distributed over internet. This can be achieved via the remote method invocation of Java and by the fact that our XSB subagents keep a mirror copy of the knowledge base of the agent. In this view, the central control component plays the role of a conductor that orchestrates the integration of all the architectural components, and thereby offers an high-level interface for Web services. The advantage of a distributed implementation is that our agents can have their parts spread on the Web. Thus, for instance if a deliberative agent is being developed at one place, several reactive agents may use its deliberations.

Acknowledgements

This work was partially supported by POCTI project 40958 "FLUX - FleXible Logical Updates". We thank David S. Warren for the implementation of break like interrupts in XSB-Prolog, and Miguel Calejo for InterProlog adaptations.

References

1. J. J. Alferes, C. Damásio, and L. M. Pereira. Semantic web logic programming tools. Workshop on Principles and Practice of Semantic Web Reasoning, at 19th Int. Conf. on Logic Programming (ICLP03). To appear in LNAI, 2003.
2. J. J. Alferes, P. Dell'Acqua, E. Lamma, J. A. Leite, L. M. Pereira, and F. Riguzzi. A logic based approach to multi-agent systems. Invited paper in The Association for Logic Programming Newsletter, 14(3), August 2001. Available at http://www.cwi.nl/projects/alp/newsletter/aug01/.
3. J. J. Alferes, P. Dell'Acqua, and L. M. Pereira. Updating, Preferring, and Acting with Abductive Agents – theory and implementation (work in progress).
4. J. J. Alferes, P. Dell'Acqua, and L. M. Pereira. A compilation of updates plus preferences. In S. Flesca, S. Greco, N. Leone, and G. Ianni (eds.), *Logics in Artificial Intelligence*, LNAI 2424, pp. 62–74, 2002.
5. J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego, I. P. de Guzmn, G. Brewka, and L. M. Pereira (eds.), *Logics in AI, Procs. JELIA'00*, LNAI 1919, pp. 345–360, 2000.

6. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. To appear in: J. of Theory and Practice of Logic Programming, 2003.
7. J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabled dual programs. In D. De Schreye (ed.), *ICLP'99*. MIT Press, 1999.
8. IMPACT system. Available at www.cs.umd.edu/projects/impact.
9. 3ALP Web site. Available at www.cs.uu.nl/3apl.
10. InterProlog. Available at www.declarativa.com/InterProlog/default.htm.
11. XSB-Prolog. Available at xsb.sourceforge.net.
12. S. Costantini. Towards active logic programming. Elect. Proc. 2nd Int. Workshop on Component-Based Software Development in Computational Logic, 1999.
13. P. Dell'Acqua, U. Nilsson, and L. M. Pereira. A logic based asynchronous multi-agent system. Computational Logic in Multi-Agent Systems (CLIMA02). Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 70, Issue 5, 2002.
14. P. Dell'Acqua and L. M. Pereira. Preferring and updating in abductive multi-agent systems. In A. Omicini, P. Petta, and R. Tolksdorf (eds.), *Engineering Societies in the Agents' World (ESAW 2001)*, LNAI 2203, pp. 57–73. Springer-Verlag, 2001.
15. P. Dell'Acqua, L. M. Pereira, and A. Vitória. User preference information in query answering. In T. Andreasen, A. Motro, H. Christiansen, and H. L. Larsen (eds.), *5th Int. Conf. on Flexible Query Answering Systems*, LNCS 2522, pp. 163–173, 2002.
16. M. Engberg. *An implementation of a rational, reactive agent.* M. Sc. No LITH-ITN-KTS-EX03/021–SE, Dept. of Science and Technology (ITN), Linköping University, Norrköping, Sweden, 2003.
17. K. Fischer and C. G. Jung. A layered agent calculus with concurrent, continuous processes. In M. P. Singh, A. Rao, and M. J. Wooldridge (eds.), *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, LNCS 1365, pp. 245–258, 1997.
18. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo (eds.), *Logic in Databases, Int. Workshop LID'96*, LNCS 1154, pp. 137–149, 1996.
19. J. A. Leite, J. J. Alferes, and L. M. Pereira. MINERVA - A Dynamic Logic Programming Agent Architecture. In J. J. Meyer and M. Tambe (eds.), *Intelligent Agents VIII*, LNAI 2333, pp. 141–157, 2002.
20. J. P. Müller. *The Design of Intelligent Agents: A Layered Approach.* LNCS 1177, 1997.
21. A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall (eds.), *Proc. of Knowledge Representation and Reasoning (KR&R-92)*, pp. 473–484. Morgan Kaufmann, 1991.
22. F. Sadri and F. Toni. Computational logic and multiagent systems: a roadmap, August 2001. Available at http://www2.ags.uni-sb.de/net/Forum/Supportdocs.html.
23. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
24. A. Sloman. What sort of architecture is required for a human-like agent. In M. Wooldridge and A. Rao (eds.), *Foundations of Rational Agency*, pp. 35–52. Kluwer Academic Publishers, 1999.
25. A. Sloman and B. Logan. Architectures and tools for human-like agents. In *Proc. 2nd European Conf. on Cognitive Modelling (ECCM98)*, pp. 58–65, 1998.