Common-sense reasoning as proto-scientific agent activity

Pierangelo Dell'Acqua*† and Luís Moniz Pereira†

* Department of Science and Technology - ITN
Linköping University, 601 74 Norrköping, Sweden
pier@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 wish to model common-sense reasoning in situations where it contains some of the ingredients typical of proto-scientific reasoning, with a view to future elaboration and proof of concept. To model this proto-scientific narrative, we employ the integrative formal computational machinery we have been developing and implementing for rational cooperative epistemic agents. In our logic-based framework, agents can update their own and each other's theories, which are comprised of knowledge, active rules, integrity constraints, queries, abducibles, and preferences; they can engage in abductive reasoning involving updatable preferences; set each other queries; react to circumstances; plan and carry out actions; and revise their theories and preferences by means of concurrent updates on self and others.

1 Introduction

"The scientific method is nothing more than a refinement of our everyday thinking."

Albert Einstein [14, page 9].

In this paper we take up the perspective quoted above, enjoining to it the computer modelling of common-sense reasoning. Indeed, recent developments in the logical modelling of the rational functionalities of agents, and the intertwined combination of these functionalities, enable us to formally model and implement, for the first time and in an integrated fashion, common-sense reasoning and common-sense acting of agents when solving everyday practical problems. To which we add the distributed problem solving cooperation of similar agents [21].

Our goal has been to provide a trail blazing inroad into the use of computational logic formalizations and logic programming techniques to address non trivial common-sensical behaviour, as a means to provide a valid proof of concept. In the longer term, the problems jointly addressed by agents should become more and more scientific-like, in fulfillment of Einstein's dictum above,

and involve the automation of the processes of discovery, argumentation, etc, i.e. all that it takes to build and consolidate scientific knowledge, towards an AI epistemology not separate but symbiotic with human epistemology [29].

The paper's structure is as follows. First, the computational logic background and framework is recapitulated, involving a knowledge representation language, combined abductive preferential and update reasoning, and an agent's reasoning and activity cycle. Second, reactive planning is introduced, and combined with the previous functionalities. At last, the stage is set for the modelling, with the presented tools, of an elaborate example of a proto-scientific collaborative agent common-sensical behaviour. This is enacted on the basis of formalizing a doctor, patient, and third party, situated and developing interaction, as described by an ongoing natural language narrative. An appreciative conclusion brings the paper to a close.

2 Framework

2.1 Language

It is convenient to 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 ":", "$\div$" 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 for mental state by β). In turn, when α receives the message C from β, it is up to α whether or not to 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*[1]. $n_r < n_u$ 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 *not false* $\leftarrow$ *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 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 a 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.*

[1] In the remaining of the paper, we implicitly assume that the relation induced by $<$ is a strict partial order.

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 α: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:

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

Let $\mathcal{A}$ be a set of domain literals distinct from *false*, we call the members of $\mathcal{A}$ *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.

The reader can refer to [11, 12] for the declarative and procedural semantics of our framework of abductive logic-based agents, to [9] for a logic-based agent architecture, to [2] for a proof procedure of updating plus preferring reasoning, and to [10] for an asynchronous multi-agent system in which the interaction among agents is characterized by a transition rule system. In these references, the reader can find theorems, properties, and examples of our framework.

N.B.: In the sequel, rules with variables stand for the set of all their ground instances with respect to the Herbrand universe of the program.

2.2 Abductive Agents

The knowledge of an agent can dynamically evolve when the agent receives new knowledge, albeit by self-updating rules, or when it abduces new hypotheses to explain observations. The new knowledge is represented in the form of an updating program, and the new hypotheses in the form of a finite set of abducibles.

Definition 7. *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. To characterize the evolution of the knowledge of an agent we need to introduce the notion of a 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 $s \in S$.

Definition 8. *An* agent α at state s*, written as* Ψ_α^s*, 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 $\Psi_\alpha^0 = (\mathcal{A}, \{\})$. 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^i is the updating program containing the updates that α has received at state $i-1$.

2.3 Abductive stable models

In the remainder of the paper, by (2-valued) *interpretation* M we mean any consistent[2] set of literals. Given a generalized rule r of the form $L_0 \leftarrow L_1, \ldots, L_n$, we write $head(r)$ to indicate L_0 and $body(r)$ to indicate $L_1, \ldots, L_n$.

Definition 9. *Let P be a set of generalized rules and M an interpretation. The set of* default assumptions *is:*

$$Default(P, M) = \{not\ A \mid \nexists r \in P \text{ such that } head(r) = A \text{ and } M \models body(r)\}.$$

The knowledge of an agent α is characterized at the start (at state 0) by the set of all default assumptions *not* A (that is, by $Default(\{\}, M)$), for evry atom A. Its knowledge can dynamically evolve when α receives new knowledge, via a sequence of updating programs $\mathcal{U} = \{U^1, \ldots, U^s\}$. Intuitively, the evolution of knowledge may be viewed as the result of, starting with the set of all default assumptions, updating it with U^1, updating next with U^2, and so on. The role of updating is to ensure that the rules contained in a new updating program override the rules contained in a less recent updating programs, provided that their heads contradict one another. If this is not the case, then that older rules are still valid by inertia. This rationale is at the basis of the notion of rejected rules, spelled out below.

A rule r proposed via an update in U^i by an agent β is rejected at state s by an interpretation M if there exists a rule r' proposed via a subsequent update in U^j by an agent α, such that the head of r' is the complement of the head of r, the body of r' is true in M and β does not distrust the update proposed by α. In contrast, if β does distrust α, then the rule r' proposed by α cannot be used to reject older rules of β.

Definition 10. *Let $\mathcal{U} = \{U^i \mid i \in S\}$ be a sequence of updating programs and M an interpretation. The set of* rejected rules *at state s is:*

$$\begin{aligned} Reject(\mathcal{U}, s, M) = \{r \mid \exists\, (\beta \div r) \in U^i \text{ and } \exists\, (\alpha \div r') \in U^j \text{ such that } i < j \leq s, \\ head(r) = not\ head(r'), M \models body(r') \text{ and } M \not\models distrust(\alpha \div r')\} \end{aligned}$$

The idea behind the updating process is that newer rules reject older ones in such a way that contradictions can never arise between them. Thus, contradictions could only ever arise between rules introduced at the same state. Furthermore, an agent can prevent any type of updates from an agent α via the use of the atom $distrust(\alpha \div r')$. For instance, if the theory of an agent α contains the rule

[2] A set M is *consistent* iff there exists no atom X such that $X \in M$ and $not\ X \in M$.

distrust($\beta \div r$) $\leftarrow$ *liar*(β), and α believes that agent β is a liar, then α distrusts the updates proposed by β.

As the head of an active rule is a project (and not a domain atom), active rules can only be rejected by active rules. Rejecting an active rule r makes r not triggerable even if its body is true in the model. Thus, by rejecting active rules, we make the agent less reactive.

Let $\Psi^s_\alpha = (\mathcal{A}, \mathcal{U})$ be an agent α at state s and $La \subseteq \mathcal{A}$ a set of abducibles. We write $\mathcal{U} + La$ to indicate the sequence of updating programs $\mathcal{U} \cup \{U^{s+1}\}$, where $U^{s+1} = \{\alpha \div L \mid \text{for every } L \in La\}$. That is, $\mathcal{U} + La = \{U^1, \ldots, U^s, U^{s+1}\}$.

Definition 11. *Let $\Psi^s_\alpha = (\mathcal{A}, \mathcal{U})$ be an agent α at state s and M an interpretation. Let $La \subseteq \mathcal{A}$ be a set of abducibles and $\mathcal{U}' = \mathcal{U} + La$ a sequence of updating programs. M is an* abductive stable model of agent α at state s with hypotheses *La iff:*

- *false* $\notin M$
- $M = least(\mathcal{X} \cup Default(T, M) \cup \bigcup_{1 \leq i \leq s} U^i)$, *where:*

$$T = \{r \mid \exists\, (\beta \div r) \text{ in } \bigcup_{1 \leq i \leq s+1} U^i \text{ such that } M \not\models distrust(\beta \div r)\}$$
$$R = Reject(\mathcal{U}', s+1, M)$$
$$\mathcal{X} = T - R$$

In the definition above, T is a set containing all the rules r proposed by β via an update $\beta \div r$ that is not distrusted bt α. R is the set of all rejected rules at state $s+1$. Note that the abducibles are treated as a *virtual update*. That is, to compute the abductive stable models the abducibles abduced by α at state s are treated as if they were internal updates of α at state $s+1$. The virtual update is only used to compute the abductive stable models, and there is no commitment to what α will receive as update at the next state $s+1$. Finally, note that M contains all the updates (both trusted and not) that α has received.

2.4 Preferred abductive stable models

While updates allow us to deal with a dynamically evolving world, where rules change in time, preferences allow us to choose among various possible models of the world and among possible incompatible reactions. In [3], two criteria are established to remove unpreferred generalized rules in a program: removing unsupported generalized rules, and removing less preferred generalized rules defeated by the head of some more preferred one. Unsupported generalized rules are rules whose head is true in the model and whose body is defeated by the model. Below we write $body^+(r)$ (resp. $body^-(r)$) to indicate the atoms (resp. the negated atoms) in the body of a rule r.

Definition 12. *Let P be a set of generalized rules and M an interpretation. The set of* unsupported generalized rules *of P and M is:*

$$Unsup(P, M) = \{r \in P \mid M \models head(r), M \models body^+(r) \text{ and } M \not\models body^-(r)\}.$$

Unpref(P, M) *is a set of* unpreferred generalized rules *of* P *and* M *iff:*

$$Unpref(P, M) = least(Unsup(P, M) \cup \mathcal{X})$$

where $\mathcal{X} = \{\ r \in P \mid\ \exists u \in (P - Unpref(P, M))$ *such that:*
$M \models n_u < n_r,\ M \models body^+(u)$ *and* [*not* $head(u) \in body^-(r)$ *or*
$(not\ head(r) \in body^-(u),\ M \models body(r))$] $\}$.

In other words, a generalized rule is unpreferred if it is unsupported or defeated by a more preferred generalized rule (which is not itself unpreferred), or if it attacks (i.e., attempts to defeat) a more preferred generalized rule. The following definition introduces the notion of a preferred abductive stable model of an agent α at a state s with a set of hypotheses *La*. Given a sequence of updating programs $\mathcal{U}$ and the hypotheses *La* assumed at state s by α, a preferred abductive stable model of α at state s with hypotheses *La* is a stable model of the program $\mathcal{X}$ that extends P to contain all the updates in $\mathcal{U}$, all the hypotheses in *La*, and all those rules whose updates are not distrusted but that are neither rejected nor unpreferred. The preferred abductive stable model contains also the selected projects.

Definition 13. *Let* $\Psi^s_\alpha = (\mathcal{A}, \mathcal{U})$ *be an agent* α *at state* s *and* M *an abductive stable model of* α *at state* s *with hypotheses La. Let* $\mathcal{U}' = \mathcal{U} + La$ *be a sequence of updating programs.* M *is a* preferred abductive stable model of agent α at state s with hypotheses *La iff:*

- $\forall r_1, r_2 :$ *if* $(n_{r_1} < n_{r_2}) \in M$, *then* $(n_{r_2} < n_{r_1}) \notin M$
- $\forall r_1, r_2, r_3 :$ *if* $(n_{r_1} < n_{r_2}) \in M$ *and* $(n_{r_2} < n_{r_3}) \in M$, *then* $(n_{r_1} < n_{r_3}) \in M$
- $M = least(\mathcal{X} \cup Default(T, M) \cup \bigcup_{1 \le i \le s} U^i)$, *where:*

$$T = \{r \mid \exists\, (\beta \div r) \text{ in } \bigcup_{1 \le i \le s+1} U^i \text{ such that } M \not\models distrust(\beta \div r)\}$$
$$R = Reject(\mathcal{U}', s + 1, M)$$
$$\mathcal{X} = (T - R) - Unpref(T - R, M)$$

T is the set containing all the rules in updates that are trusted from α according to M, and R is the set of all the rules that are rejected at state s. $\mathcal{X}$ is the set of all the trusted rules that are neither rejected nor unpreferred.

Definition 14. *An* abductive explanation *for a query* Q *is any set* $La \subseteq \mathcal{A}$ *of hypotheses such that there exists a preferred abductive stable model* M *with hypotheses La and* $M \models Q$.

Note that at state s an agent α may have several abductive explanations for a query Q.

2.5 Agent Cycle

Every agent α can be thought of as a pair $\Psi_\alpha = (\mathcal{A},\mathcal{U})$, where $\mathcal{A}$ is a set of abducibles and $\mathcal{U}$ is a sequence of updating programs. The abducibles are used as abductive explanations for queries. The basic "engine" of an agent α is an abductive logic programming proof procedure executed via the cycle represented in Fig. 1. Below let G be a set of queries that α has to prove.

Cycle(α,s,Ψ_α^{s-1},G), where $\Psi_\alpha^{s-1} = (\mathcal{A},\mathcal{U})$ and $\mathcal{U} = \{U^1,\ldots,U^{s-1}\}$.

1. Observe and record any input in the updating program U^s.
2. Select a query g in $G \cup \mathit{Queries}(\alpha, U^s)$ and execute g. Let $\mathit{La} \subseteq \mathcal{A}$ be an abductive explanation of g, if g is provable; otherwise, let $\mathit{La} = \{\}$.
   Let $G' = G \cup \mathit{Queries}(\alpha, U^s) - \{g\}$.
3. Execute all the projects in $\mathit{ExecProj}(\mathit{La})$.
4. Cycle with (α,$s+1$,Ψ_α^s,G'), where $\Psi_\alpha^s = (\mathcal{A},\mathcal{U} \cup \{U^s\})$.

Fig. 1. *The agent cycle*

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

Step 2: α selects and executes a query g is selected from $G \cup \mathit{Queries}(\alpha, U^s)$, where

$$\mathit{Queries}(\alpha, U^s) = \{?{-}g \mid \alpha \div (?{-}g) \in U^s\}.$$

Note that only the queries issued by the agent α itself are executed. The queries issued by other agents are treated as normal updates[3]. Here, to execute g one can use any abductive proof procedure, such as ABDUAL [5, 4].

Step 3: α executes all the executable projects. The set *ExecProj* of executable projects of an agent depends on the kind of agent we want to model. For instance, in case of a cautious agent, the set of executable projects is:

$$\mathit{ExecProj}(\mathit{La}) = \{\, \beta{:}C \mid \text{for every preferred abductive stable model } M \text{ at state } s \text{ with hypotheses } \mathit{La}, \text{ it holds that } \beta{:}C \in M\}.$$

If an executable project takes the form $\beta : C$ (meaning that agent α intends to update the theory of agent β with C at state s), then (once executed) the update $\alpha \div C$ will be available as input to the cycle of the agent β.

Step 4: Finally, α cycles by increasing its state, by incorporating the updating program U^s into $\mathcal{U}$, and with the new set G' of queries.

Initially, the cycle of α is Cycle(α,1,Ψ_α^0,{}) with $\Psi_\alpha^0 = (\mathcal{A}, \{\})$.

[3] In this way, α retains control upon deciding on 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 $?{-}g$.

2.6 Temporary updates

Until now we have considered permanent updates: whenever an agent α updates its theory, the update persists by inertia until it is contradicted by a counter-update. Often, for example in applications based on planning, it is desirable to update the theory of an agent with updates that hold for a limited period of time. Consider that case where we want to make the update of the theory of an agent valid only for the next state. We employ projects of the form $\alpha : once(\ldots)$ to achieve that. Consider the active rule:

$$cond \Rightarrow \alpha : once(C)$$

in the theory of an agent α. It states that if the condition *cond* holds at the current state in the theory of α, then α must update its theory with C with the restriction that C holds only at the next state in what this update is concerned. This ability can be coded in our framework as follows. We can first formalize a counter that allows an agent to count its states.

$$\begin{array}{l} counter(s(0)) \\ counter(X) \Rightarrow \alpha : not\ counter(X) \\ counter(X) \Rightarrow \alpha : counter(s(X)) \end{array}$$

Initially, the counter is set to 1. At every cycle of α, the two active rules above are triggered. By executing the corresponding projects, at the next cycle of α the value of the counter is increased by one unit.

Suppose that C can be a generalized rule or an active rule. Then, active rules of the form:

$$\begin{array}{l} cond \Rightarrow \alpha : once(a) \\ cond \Rightarrow \alpha : once(a \leftarrow a_1, \ldots, a_n) \\ cond \Rightarrow \alpha : once(a_1, \ldots, a_n \Rightarrow z) \end{array}$$

can be expressed in our framework as:

$$\begin{array}{l} cond, counter(X) \Rightarrow \alpha : (a \leftarrow counter(s(X))) \\ cond, counter(X) \Rightarrow \alpha : (a \leftarrow a_1, \ldots, a_n, counter(s(X))) \\ cond, counter(X) \Rightarrow \alpha : (counter(s(X)), a_1, \ldots, a_n \Rightarrow z) \end{array}$$

3 Reactive Planning

In *classical planning* an agent typically performs the following steps: (*i*) it makes observations about the initial situation of the world, (*ii*) it constructs a plan that achieves the desired goal, and finally (*iii*) it executes the plan. During the planning process (step (*ii*)), the world must be frozen. In fact, if some condition relevant to the plan changes its value, it is possible that some precondition crucial to the plan is not satisfied any longer. The same applies to step (*iii*). This assumption is not realistic in real-world applications where for instance the world is inaccessible, non-deterministic and open (i.e., the agent may have

incomplete information), exogenous events are possible (e.g., actions by other agents or natural events), and the world may not match the agent's model of it (i.e., the agent may have incorrect information). To overcome the restrictions of classical planning, research in planning (see e.g. [7, 15, 17, 19, 28]) is focusing more and more on *reactive planning systems*, i.e., systems able to plan and to control the execution of plans. In reactive planning, planning itself is not necessarily a priori reasoning about the preconditions and the effects of actions. Rather, planning (deliberation) and execution (action) are interleaved, plans can be modified, abandoned, and substituted with alternative plans dynamically at run-time. During the execution of this kind of plans, the agent can therefore interleave deliberation and action to perform run-time decision making.

This approach to planning may involve also the ability to generate revisable plans, that is, plans that are generated in stages. After having executed the actions of each stage, we test whether the performed actions have been successful in order to move to the next stage. If some action failed, then we must revise our plan. A successful plan is a revisable plan that has been completely executed such that the obtained final state satisfies the original goal.

3.1 Plans

We introduce here the notion of a *plan* as a sequence of actions. In the following, given a set $\mathcal{N}$ of action names, we represent an action whose name is $a \in \mathcal{N}$ and its preconditions and effects are *precA* and *effectA* with an active rule of the form:

$$a, precA \Rightarrow \alpha : effectA$$

We write $a_1 \diamond \ldots \diamond a_k$ to indicate the sequence $a_1, \ldots, a_k$ of the action names.

Definition 15. *Let $\mathcal{N}$ be a set of action names. Then, a* plan *is any sequence $a_1 \diamond \ldots \diamond a_k$ $(k \geq 0)$ of action names in $\mathcal{N}$.*

Intuitively, a plan $a \diamond b \diamond c$ states to execute first a, then b, and finally c. This capability allows an agent to execute actions in sequence. For instance, if the actions a, b, and c are defined by three distinct active rules r_a, r_b, and r_c, then the sequence $a \diamond b \diamond c$ permits executing the projects occurring in the heads of r_a, r_b, and r_c sequentially.

Consider an agent α that has a plan consisting of two actions, a followed by b. Suppose that the plan is executable if some condition *cond* holds. Such a planning problem can be expressed as:

$$p \leftarrow cond, a \diamond b$$
$$a, precA \Rightarrow \alpha : effectA$$
$$b, precB \Rightarrow \alpha : effectB$$

The execution of the plan $a \diamond b$ is launched by means of the query $?{-}p$ if the conditions *cond* of the plan are fulfilled. When the execution of the plan $a \diamond b$ starts, the action a is executed first, provided that *precA* holds. The effects of

executing a are expressed via the project $\alpha : \mathit{effectA}$. If instead precA does not hold, then the plan p cannot be accomplished and it is therefore abandoned. Once the execution of a is terminated, the action b is executed.

This planning problem can be coded in our framework as follows:

$$
\begin{array}{ll}
p \leftarrow \mathit{cond}, \mathit{start} & (1) \\
\mathit{exec}(a), \mathit{precA} \Rightarrow \alpha : \mathit{effectA} & (2) \\
\mathit{exec}(b), \mathit{precB} \Rightarrow \alpha : \mathit{effectB} & (3) \\
\mathit{start} \Rightarrow \alpha : \mathit{once}(\mathit{exec}(a)) & (4) \\
\mathit{exec}(a), \mathit{precA} \Rightarrow \alpha : \mathit{once}(\mathit{exec}(ta)) & (5) \\
\mathit{exec}(ta) \Rightarrow \alpha : \mathit{once}(\mathit{exec}(b)) & (6) \\
\mathit{exec}(b), \mathit{precB} \Rightarrow \alpha : \mathit{once}(\mathit{exec}(tb)) & (7)
\end{array}
$$

with the set of abducibles $\mathcal{A} = \{\mathit{start}\}$. Suppose that cond is true. Then the query $?{-}p$, by means of the rule (1), has the effect of abducing start which in turn will trigger the active rule (4). When the agent α executes the project $\alpha : \mathit{once}(\mathit{exec}(a))$, at the next agent cycle it will update its theory with $\mathit{once}(\mathit{exec}(a))$ indicating that it must execute the action a of the plan. This update will hold only at the next state of α (temporary update). (For the coding of $\mathit{once}(\ldots)$ see Section 2.6.)

The active rules (2) and (3) characterize the actions a and b by stating to update the theory of α with the effect of the action provided that that action must be executed and its preconditions hold. Finally, the active rules (5)–(7) model the sequencing of the actions a and b. If the action a must be executed and its preconditions hold, then the active rule (5) is triggered. Doing so has the effect of making α (at the next cycle) to temporarily update its theory with $\mathit{once}(\mathit{exec}(ta))$ indicating that the execution of a is terminated. In turn, $\mathit{exec}(ta)$ will trigger the active rule (6) which temporarily updates the theory of α with $\mathit{once}(\mathit{exec}(b))$. This means that at the next cycle α must execute the action b. This proceeds until α has terminated the execution of b (i.e., $\mathit{once}(\mathit{exec}(tb))$, the execution of the entire plan is then terminated.

3.2 Executing plans

Plans that can be executed in parallel and plans containing actions that can start a subplanning process can also be expressed. The parallel execution of two plans p_1 and p_2 can be expressed as:

$$
\begin{array}{l}
p \leftarrow p_1, p_2 \\
p_1 \leftarrow \mathit{cond}_1, a_1 \diamond \ldots \diamond a_n \\
p_2 \leftarrow \mathit{cond}_2, b_1 \diamond \ldots \diamond b_m
\end{array}
$$

Executing the plan p gives rise to the parallel execution of the subplans p_1 and p_2. One can also formalize a plan p containing an action a_1 whose effect is to

start the execution of a subplan q.

$$p \leftarrow cond_1, a_1 \diamond a_2 \diamond a_3$$
$$q \leftarrow cond_2, b_1 \diamond b_2$$
$$a_2 \Rightarrow \alpha : ?{-}q$$

In a similar way, one can formalize reactive plans, that is, plans whose planning phase and execution phase are interleaved.

$$p \leftarrow cond_1, a_1 \diamond a_2 \diamond a_3$$
$$a_3 \Rightarrow \alpha : ?{-}planner(\ldots)$$

Executing the last action of the plan $a_1 \diamond a_2 \diamond a_3$ starts a deliberation phase: a planner is invoked to generate the next actions to be performed.

Often in reactive planning it is needed to interleave the planning phase with an abductive phase. During the abductive phase, the agent can make abductions to be employed later in its decision making. Let $\mathcal{A} = \{b\}$ a set of abducibles. Consider the planning problem:

$$p \leftarrow cond_1, a_1 \diamond a_2 \diamond a_3$$
$$a_2 \Rightarrow \alpha : ?{-}b$$

When α executes the action a_2, it will issue the project $\alpha : ?{-}b$, and the query $?{-}b$ will be executed at the next cycle of α. Since $b \in \mathcal{A}$, b will be abduced.

3.3 Conditional plans with sensing actions

In the presence of incomplete information, the notion of a plan as a "fixed" sequence of actions is no longer adequate. In response to this, a notion of conditional plan, that combines sensing actions and conditional constructs such as if-then-else, has been proposed [27, 28]. In this context, a sensing action is an action that determines the value of some proposition. For example, "looking at the door" is a sensing action that determines whether or not the door is closed.

In this section we illustrate how to express conditional plans in our framework. We consider conditional plans formalized as follows, where we assume that every action in $\mathcal{N}$ is either a sensing action or a non-sensing action.

Definition 16. *Let $\mathcal{N}$ be a set of action names.*

1. *A sequence $a_1 \diamond \ldots \diamond a_k$ $(k \geq 0)$ of non-sensing action names in $\mathcal{N}$ is a* conditional plan.
2. *If $a_1 \diamond \ldots \diamond a_k$ $(k \geq 0)$ is a sequence of non-sensing action names in $\mathcal{N}$, a_{k+1} is a sensing action that determines f, and b and c are conditional plans, then $a_1 \diamond \ldots \diamond a_k \diamond a_{k+1} \diamond if(f, b, c)$ is a* conditional plan.
3. *Nothing else is a conditional plan.*

To execute a plan $a_1 \diamond \ldots \diamond a_k \diamond a_{k+1} \diamond if(f, b, c)$, an agent α first executes $a_1 \diamond \ldots \diamond a_k \diamond a_{k+1}$. Then it evaluates f: if f is true, it executes b else c.

Consider the following planning problem:

$$p \leftarrow cond, a_1 \diamond a_2 \diamond if(f, b_1 \diamond b_2 \diamond b_3, c_1)$$

where a_2 is a sensing action determining f, and $b_1 \diamond b_2 \diamond b_3$ and c_1 are conditional plans. Suppose that every action x in the conditional plan (including sensing actions) is expressed as:

$$x, precX \Rightarrow \alpha : effectX$$

The conditional plan above can be first translated into the (ordinary) plan:

$$\begin{array}{l} p \leftarrow cond, a_1 \diamond a_2 \diamond if(f, p_1, p_2) \\ p_1 \leftarrow b_1 \diamond b_2 \diamond b_3 \\ p_2 \leftarrow c_1 \\ if(f, p_1, p_2), f \Rightarrow \alpha : ?{-}p_1 \\ if(f, p_1, p_2), not\ f \Rightarrow \alpha : ?{-}p_2 \end{array}$$

and then coded into our framework as shown in Section 3.1. Note that the sensing action a_2 and the if-then-else construct $if(f, p_1, p_2)$ are treated here as ordinary actions.

3.4 Executing, suspending and resuming plans

In reactive planning the ability to modify, abandon, and substitute plans with alternative plans at run time is fundamental. In this section we illustrate how plans can be suspended, resumed and stopped. Consider the following planning program:

$$\begin{array}{l} p \leftarrow cond, a \diamond b \\ a, precA \Rightarrow \alpha : effectA \\ b, precB \Rightarrow \alpha : effectB \\ cond1 \Rightarrow \alpha : ?{-}stop(p) \\ cond2 \Rightarrow \alpha : ?{-}suspend(p) \\ cond3 \Rightarrow \alpha : ?{-}resume(p) \end{array}$$

where the last three active rules state the conditions to stop, suspend, and resume the plan. This planning problem can be coded in our framework as follows:

$$p \leftarrow cond, not\ block, start \quad (1)$$
$$block \leftarrow cond1 \quad (2)$$
$$block \leftarrow cond2 \quad (3)$$
$$not\ block, exec(a), precA \Rightarrow \alpha : effectA \quad (4)$$
$$not\ block, exec(b), precB \Rightarrow \alpha : effectB \quad (5)$$
$$start \Rightarrow \alpha : once(exec(a)) \quad (6)$$
$$not\ block, exec(a), precA \Rightarrow \alpha : once(exec(ta)) \quad (7)$$
$$not\ block, exec(ta) \Rightarrow \alpha : once(exec(b)) \quad (8)$$
$$not\ block, exec(b), precB \Rightarrow \alpha : once(exec(tb)) \quad (9)$$
$$cond2, exec(X) \Rightarrow \alpha : suspended(X) \quad (10)$$
$$cond3, suspended(X) \Rightarrow \alpha : once(exec(X)) \quad (11)$$
$$cond3, suspended(X) \Rightarrow \alpha : not\ suspended(X) \quad (12)$$

where $\mathcal{A} = \{start\}$. The query $?{-}p$ has the effect to start the execution of the plan provided that the preconditions *cond* of p hold and that the plan is not blocked, i.e., both *cond1* and *cond2* do not hold. This is achieved by abducing *start*, that in turn triggers the active rule (6).

Rules (4) and (5) characterize the actions a and b. Those rules are not triggerable in the case the plan is blocked. The active rules (7)-(9) model the sequential execution of the plan. Being *not block* one of their preconditions, as soon as one of the conditions *cond1* or *cond2* become true, none of these 3 active rules is triggerable, and the plan is consequently stopped or suspended.

Rule (10) characterizes the suspension of the plan. If *cond2* is true, then *block* holds and the agent α updates its theory with the action suspended (i.e., with *suspended*(X)). This is needed to allow α to resume its plan. If the condition *cond3* becomes true, then the plan is resumed by triggering the last two active rules (11) and (12). α updates its theory with *once*$(exec(X))$ (rule (11)) indicating that the suspended action X must be executed and with *not suspended*(X) (rule (12)) X being no longer suspended.

4 Modelling Proto-scientific Reasoning by Rational Agents

Next we illustrate how to model, with the above instruments, common-sense reasoning in situations where it contains some of the ingredients typical of proto-scientific reasoning, with a view to future elaboration, proof of concept, and extension of the approach to scientific reasoning itself. To do so, we construe an exemplificative narrative of a doctor/patient cooperative diagnostic situation development, involving a combination of a number of common rational abilities illustrative of proto-scientific reasoning and acting, which demand their joint exercise, both in an individual and a cooperative fashion, akin to scientific theory

refinement. To model this proto-scientific narrative, we employ the integrative formal computational machinery we have been developing and implementing for rational cooperative epistemic agents, and recapitulated above. Indeed, in our logic-based framework, agents can update their own and each other's theories, which are comprised of knowledge, active rules, integrity constraints, queries, abducibles, and preferences; they can engage in abductive reasoning involving updatable preferences; set each other queries; react to circumstances; plan and carry out actions; and revise their theories and preferences by means of concurrent updates on self and others.

The narrative below involves an initial patient situation requiring causal explanation; plus his interactive recourse to a doctor, whose initial therapeutic theory, diagnoses, and diagnostic preferences, are conducive to his advising the patient; and furthermore, initiative is required by the patient about courses of action to obtain prescribed medicine, and experimentation and observation of its effect; but meanwhile, unforeseen circumstances provide unexpected new information and action from a third agent, become pertinent for the problem at hand; as a result, the doctor's original theory is revised, in what regards his diagnostic preferences, in the light of the patient's experimentation, and the unexpected triggering of an unforeseen action by the third party. The example has been fully tested with our implementation.

4.1 Requiring causal explanation

John runs a small software house and likes working until late when needed. He drinks coffee and has been a heavy smoker for a long time. Recently, he's having problems with sleeping. He would like to have a break from his work, perhaps a vacation, but he does not have any company. Thus, he keeps on working. John does not know what the cause is and decides to visit a doctor. He tells the doctor about his sleeping problems and asks him what is the cause. John answers all of the doctor's questions.

To abbreviate the presentation of the example, we assume that every agent is equipped with an initial theory. This theory can be understood as an initial updating program whose updates are performed by the agent itself. Thus, when we write that the theory of an agent α contains a rule r, we intend that we have the update $\alpha \div r$ in one of the updating program of α. Recall that we write generalized rules containing variables as a shorthand for all their ground instances.

John

$work$
$likeWork$
$sProblems$
$badHabits$
$longTimeBadHabits$
$explanation \leftarrow cause(P, X, sProblems)$

$sProblems, company \Rightarrow john : takeVacation$
$sProblems, not\ company, not\ explanation \Rightarrow doctor : sProblems$
$sProblems, not\ company, not\ explanation \Rightarrow doctor : ?{-}askReason(sProblems)$

$doctor{\div}(?{-}Q), Q \Rightarrow doctor : Q$
$doctor{\div}(?{-}Q), not\ Q \Rightarrow doctor : not\ Q$

Since the theory of John does not contain any priority rules, the preferred stable model of John at the current state is equivalent to his abductive stable model M_1 whose positive part consists of $M_1^+ = \{work, likeWork, sProblems, badHabits, longTimeBadHabits, doctor : sProblems, doctor : ?{-}askReason(sProblems)\}$. According to the definition of agent cycle, John will execute the two projects in M_1^+ (step 3 of the agent cycle), and then he will cycle (step 4).

When the doctor shall observe his inputs (step 1), he will receive the two updates:

$$john{\div}sProblems$$
$$john{\div}(?{-}askReason(sProblems))$$

Any time the doctor is asked a reason for a medical problem by a patient, the doctor must make a diagnosis. To do so, he must first collect the relevant information about the medical problem from the patient, then diagnose the cause of the problem and tell it to the patient together with the suggested treatment.

doctor

$\mathcal{A} = \{cause(\ldots), treatment(\ldots)\}$
$\mathcal{N} = \{collectRelInfo(\ldots), findCause(\ldots), findTreatment(\ldots), answer(\ldots)\}$

$$diagnosis(P, X, Y) \leftarrow collectRelInfo(P, Y) \diamond findCause(P, X, Y) \diamond answer(P, cause(P, X, Y)) \diamond findTreatment(X, T) \diamond answer(P, treatment(X, T))$$

$relevant(work, sProblems)$
$relevant(likeWork, sProblems)$
$relevant(badHabits, sProblems)$
$relevant(longTimeBadHabits, sProblems)$

$P{\div}(?{-}askReason(Y)) \Rightarrow doctor : ?{-}diagnosis(P, X, Y)$
$collectRelInfo(P, Y), relevant(R, Y) \Rightarrow P : ?{-}R$
$answer(P, A) \Rightarrow P : A$

$findCause(P, X, Y) \Rightarrow doctor : ?{-}cause(P, X, Y)$
$findTreatment(X, T) \Rightarrow doctor : ?{-}treatment(X, T)$

Since the doctor does not have any goal to execute at step 2, he starts executing his projects (step 3). At this step, the unique project in his preferred

abductive stable model is $doctor : ?{-}diagnosis(john, X, sProblems)$. Thus, at the next cycle of the doctor, the plan to make a diagnosis will start (step 2) by collecting all the information relevant for the sleeping problems. This will make the doctor ask John the following questions $john : ?{-}work$, $john : ?{-}likeWork$, $john : ?{-}badHabits$ and $john : ?{-}longTimeBadHabits$.

After the replies of John (recall that John answers every question of the doctor), the theory of the doctor will be updated with:

$$john \div work$$
$$john \div likeWork$$
$$john \div badHabits$$
$$john \div longTimeBadHabits$$

After having executed the first action of the diagnosis plan, the doctor must find out a cause and the corresponding treatment of the problem, and tell them to John. To find a cause, the doctor employs abduction. This is achieved by means of the active rule

$$findCause(P, X, Y) \Rightarrow doctor : ?{-}cause(P, X, Y)$$

The doctor has three hypotheses that may explain John's sleeping problems: bad habits (like drinking coffee and smoking), stress, or insomnia. The doctor evaluates John with the help of his medical history, and he diagnoses a chronic insomnia. The doctor discards bad habits since John has been drinking coffee and smoking for many years without the attending sleeping problems. The doctor prefers to diagnose chronic insomnia attributable to stress since John's stress may be positive stress due to the fact that John likes his work.

doctor

$cause(P, insomnia, sProblems) < cause(P, stress, sProblems) \leftarrow pStress(P)$
$cause(P, stress, sProblems) < cause(P, insomnia, sProblems) \leftarrow stress(P), not\ pStress(P)$
$pStress(P) \leftarrow stress(P), P \div likeWork$
$stress(P) \leftarrow P \div work$
$false \leftarrow cause(P, badHabits, sProblems), P \div longTimeBadHabits$

Being $pStress(john)$ true in the theory of the doctor, the doctor prefers the abductive explanation $cause(john, insomnia, sProblems)$ to the abductive explanation $cause(john, stress, sProblems)$. Since it does not satisfy the integrity constraints, the abductive explanation $cause(john, badHabits, sProblems)$ is excluded by the doctor.

As treatment for insomnia, the doctor can either prescribe sleeping pills or suggest John have a rest. Sleeping pills being preferable to a vacation on the assumption that John can continue to work by having the pills, the doctor prescribes them.

doctor

$treatment(P, insomnia, sPills) < treatment(P, insomnia, rest) \leftarrow cWork(sPills), P \div work$
$cWork(sPills)$

According to the diagnosis plan, the doctor will tell John about the cause and the treatment for his sleeping problems. This is achieved by executing the actions $answer(john, cause(john, insomnia, sProblems))$ and $answer(john, treatment(john, insomnia, sPills))$ which make the doctor execute the projects $john : cause(john, insomnia, sProblems)$ and $john : treatment(john, insomnia, sPills)$. Once the two projects are executed, John updates his theory at the next cycle with:

$$doctor \div cause(john, insomnia, sProblems)$$
$$doctor \div treatment(john, insomnia, sPills)$$

4.2 Agent initiative to obtain prescribed medicines

John tries to get sleeping pills before going to bed. Since it is late, he thinks the pharmacy nearby is closed, and plans to go to another pharmacy, downtown.

John

$\mathcal{A} = \{\}$
$\mathcal{N} = \{goToPharmacy, askPills(\ldots), buyPills, havePills(\ldots), askAnotherPharmacy, go(\ldots), enterPharmacy(\ldots)\}$

$takePills \leftarrow doctor \div treatment(john, insomnia, sPills)$
$getPills \leftarrow goToPharmacy \diamond$
$\quad askPills(pharmacist) \diamond$
$\quad if(havePills(pharmacist), buyPills, askAnotherPharmacy)$
$choosePharmacy(f1) \leftarrow open(f1), not\ choosePharmacy(f2)$ $(r1)$
$choosePharmacy(f2) \leftarrow open(f2), not\ choosePharmacy(f1)$ $(r2)$
$r1 < r2 \leftarrow near(f1)$
$near(f1)$
$open(f2)$
$goTo(f1) \leftarrow choosePharmacy(f1), go(nearSquare) \diamond enterPharmacy(f_1)$
$goTo(f2) \leftarrow choosePharmacy(f2), go(square) \diamond go(center) \diamond enterPharmacy(f_2)$
$takePills, not\ havePills \Rightarrow john : ?{-}getPills$
$goToPharmacy \Rightarrow john : ?{-}goTo(X)$
$buyPills \Rightarrow john : havePills$
$go(X) \Rightarrow john : at(X)$
$enterPharmacy(F) \Rightarrow john : in(F)$

Since John must take the sleeping pills and he does not have them, by triggering the first active rule above, he will issue the internal project $john : ?{-}getPills$. This has the effect of launching the plan to get to a pharmacy and buy the pills. This plan is a conditional plan. In fact, $askPills(pharmacist)$ is a sensing action determining whether or not the pharmacist has the sleeping pills. The subsequent action $if(\ldots)$ states that if the pharmacist has the pills, John will buy them, otherwise he will ask the pharmacist for the address of another pharmacy.

The theory of John contains a priority rule stating to prefer r_1 to r_2 if the pharmacy f_1 is nearer. As f_1 is not open, it cannot be chosen (i.e., the body of r_1 is false). In fact, the rule r_2 does not belong to the set of unpreferred rules

(see Def. 12) since $body^{+}(r_1)$ is not true. $choosePharmacy(f1)$ being false and $open(f2)$ true, John chooses f_2. To get to f_2 John must get to the square, to the center, and finally enter f_2.

While he is going there, he notices lights on in the nearby pharmacy and he concludes that it is open. So he decides to interrupt his original plan and go to this nearby one. It being open, he buys the pills.

$$environment \div lightOn(f1)$$
$$lightOn(f1) \Rightarrow john : open(f1)$$
$$open(f1) \Rightarrow john : ?-stop(getPills)$$
$$open(f1) \Rightarrow john : ?-getPills$$

Since John believes that f_1 is open, he will trigger the last 2 active rules above. Thus, John will stop the execution of the plan and he will relaunch his goal to get to a pharmacy. Now, since f_1 is open, the rule r_1 is preferable to the rule r_2. Therefore John will go to the nearby pharmacy.

The ability to stop the execution of a plan and replanning characterize the reactive behaviour of John. Here we have coded those abilities directly into the theory of John. In general, we can express the relevant conditions under which a plan must be stopped and invoke a planner to revise the plan accordingly.

4.3 Patient's experimentation

John takes the sleeping pills. But his work implies coffee and stress, and the attempt fails. One day he meets his friend Pamela and tells her about his problems.

$$john \div sProblems$$
$$john \div takingPills$$
$$john \div stillsProblems$$

Pamela advises him to suspend the taking of sleeping pills, not to work so hard, and to have some rest. She invites him for an exciting vacation to one of the Caribbean islands.

Pamela

$$friend(john)$$
$$P \div sProblems \Rightarrow P : rest$$
$$P \div takingPills, P \div stillsProblems \Rightarrow P : not\ takePills$$
$$friend(P) \Rightarrow P : invite(vacation, carabbean)$$

John decides to follow her piece of advice and accepts her invitation.

John

$$pamela \div invite(vacation, carabbean)$$
$$A \div invite(X, Y), female(A) \Rightarrow A : accept(X, Y)$$
$$female(pamela)$$

4.4 Doctor's original theory revised

Subsequently John can sleep. One day he meets the doctor and tells him what happened.

$$john \div takingPills$$
$$john \div stillsProblems$$
$$john \div vacation$$
$$john \div not\ sProblemsAfterVacation$$

The doctor now revises his theory of preferences to suggest a vacation in the first place in the future.

doctor

$$P \div takingPills, P \div stillsProblems \Rightarrow$$
$$doctor : not\ (treatment(Q, insomnia, sPills) < treatment(Q, insomnia, rest))$$
$$P \div vacation, P \div not\ sProblemsAfterVacation \Rightarrow$$
$$doctor : treatment(Q, insomnia, rest) < treatment(Q, insomnia, sPills)$$

The doctor updates his priority rules via the two active rules above in such a way as to suggest a different treatment to other patients with insomnia problems.

5 Related Work

The use of computational logic for modelling multi-agent systems has been widely investigated (e.g., see [1, 23] for a roadmap). One approach close to our own is the agent-based architecture proposed by Kowalski and Sadri [16], 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 [8]. 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 [6] 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.

The BDI approach [22] 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.

Another logic-based formalism proposed for representing agents is Agent0 [24]. 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 [23].

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 [13].

Sloman et al. [25, 26] 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. [18]. 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.

6 Conclusion

We believe to have shown that the application of proto-scientific reasoning in common-sense examples, modelled by collections of rational agents, is an avenue of research worth pursuing with a view to further the modelling of collaborative scientific theory development and refinement. Three worthwhile aspects we did not touch upon, but which are already well within reach of present formal machinery are rule induction, argumentation, and mutual debugging.

Acknowledgements

L. M. Pereira acknowledges the support of POCTI project 40958 "FLUX - FleXible Logical Updates".

References

1. 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/.
2. 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, editors, *Logics in Artificial Intelligence*, LNAI 2424, pages 62–74, Berlin, 2002. Springer-Verlag.
3. J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego, I. P. de Guzmn, G. Brewka, and L. M. Pereira, editors, *Logics in AI, Procs. JELIA'00*, LNAI 1919, pages 345–360, Berlin, 2000. Springer.
4. J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabled dual programs. In D. De Schreye, editor, *ICLP'99*. MIT Press, 1999.
5. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.
6. IMPACT system. Available at www.cs.umd.edu/projects/impact.
7. C. Baral and S. C. Tran. Relating theories of actions and reactive control. *Linköping Electronic Articles in Computer and Information Science*, 3(9), 1998. Available at: http://www.ep.liu.se/ea/cis/1998/009.
8. S. Costantini. Towards active logic programming. Electronic Proc. of COCL'99, Second Int. Workshop on Component-Based Software Development in Computational Logic, 1999.
9. P. Dell'Acqua, M. Engberg, and L. M. Pereira. An architecture for a rational, reactive agent. In F. M. Pires and S. Abreu, editors, *Progress in Artificial Intelligence*, LNCS 2902, pages 379–393, Berlin, 2003. Springer-Verlag.
10. 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.
11. P. Dell'Acqua and L. M. Pereira. Enabling agents to update their knowledge and to prefer. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence, 10th Portuguese Int. Conf. on Artificial Intelligence (EPIA'01)*, LNAI 2258, pages 183–190. Springer-Verlag, 2001.
12. P. Dell'Acqua and L. M. Pereira. Preferring and updating in abductive multi-agent systems. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Engineering Societies in the Agents' World (ESAW 2001)*, LNAI 2203, pages 57–73. Springer-Verlag, 2001.
13. K. Fischer and C. G. Jung. A layered agent calculus with concurrent, continuous processes. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV: Agent Theories, Architectures, and Languages (ATAL97)*, LNCS 1365, pages 245–258, Berlin, 1997. Springer-Verlag.
14. Susan Haack. *Defending Science within reason*. Prometheus Books, 2003.
15. R. M. Jensen and M. M. Veloso. Interleaving deliberative and reactive planning in dynamic multi-agent domains. In *Proc. of the AAAI Fall Symposium on Integrated Planning for Autonomous Agent Architectures*. AAAI Press, 1998.
16. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases, Int. Workshop LID'96*, LNCS 1154, pages 137–149. Springer, 1996.

17. D. Kumar and L. Meeden. A hybrid connectionist and BDI architecture for modeling enbedded rational agents. In *Proc. of the AAAI Fall Symposium on Cognitive Robotics*, pages 84–90. AAAI Press, 1998.
18. J. A. Leite, J. J. Alferes, and L. M. Pereira. MINERVA - A Dynamic Logic Programming Agent Architecture. In J. J. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, LNAI 2333, pages 141–157, Berlin, 2002. Springer-Verlag.
19. Y. Lesperance, K. Tam, and M. Jenkin. Reactivity in a logic-based robot programming framework. In *Proc. of the AAAI Fall Symposium on Cognitive Robotics*, pages 98–105. AAAI Press, 1998.
20. J. P. Müller. *The Design of Intelligent Agents: A Layered Approach*. LNCS 1177. Springer Verlag, 1997.
21. L. M. Pereira. Philosophical Incidence of Logic Programming. In Handbook of the Logic of Argument and Inference, D. Gabbay et al. (eds.), pp. 425-448, Studies in Logic and Practical Reasoning series, Vol. 1, Elsevier Science, 2002.
22. A. S. Rao and M. P. Georgeff. Modelling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-92)*, pages 473–484. Morgan Kaufmann, 1991.
23. 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.
24. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
25. A. Sloman. What sort of architecture is required for a human-like agent. In M. Wooldridge and A. Rao, editors, *Foundations of Rational Agency*, pages 35–52. Kluwer Academic Publishers, 1999.
26. A. Sloman and B. Logan. Architectures and tools for human-like agents. In *Proc. 2nd European Conf. on Cognitive Modelling (ECCM98)*, pages 58–65, 1998.
27. T. C. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, 2001.
28. L. Spalazzi and P. Traverso. A dynamic logic for acting, sensing, and planning. *J. Logic and Computation*, 10(6):787–821, 2000.
29. G. Wheeler and L. M. Pereira. Epistemology and Artificial Intelligence. Selected papers from 3rd Int. W. Computational Models of Scientific Reasoning and Applications. J. of Applied Logic, in this issue.