Updating, Preferring, and Acting with Abductive Agents – theory and implementation

José Júlio Alferes†, Pierangelo Dell’Acqua‡, and Luís Moniz Pereira†

† Centro de Inteligência Artificial - CENTRIA
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{jja,lmp}@di.fct.unl.pt
‡ Department of Science and Technology - ITN
Linköping University, 601 74 Norrköping, Sweden
pier@itn.liu.se

Abstract. We present a logical framework for a kind of agent that can communicate with and update other agents, can react to the environment, is able to prefer, whether beliefs or reactions, when several alternatives are possible, and is able to abduce hypotheses to explain observations. The knowledge state of an agent is represented by an updatable prioritized abductive logic program, in which priorities among rules can be expressed to allow the agent to prefer. We present several examples to illustrate how our approach functions, including how to prefer abducibles to tackle the problem of multiple hypotheses and how to perform the interplay between planning and acting.

1 Introduction

We present a logical formalization of a framework for multi-agent systems and we define its semantics. In this framework, we embed a flexible and powerful kind of agent. In fact, these agents are rational, reactive, abductive, able to prefer and they can update the knowledge base of other agents (including their own).

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 present a declarative and procedural semantics of this kind of agent.

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. A project of the form $\alpha:C$ of an agent $\beta$ denotes the intention of $\beta$ of proposing to update the theory of an agent $\alpha$ with $C$. Correspondingly, an update of the form $\beta\triangleright C$ in the theory of $\alpha$ denotes the intention of $\beta$ to update the current theory of $\alpha$ with $C$. It is then up to $\alpha$ whether or not to accept that update. For example, if $\alpha$ trusts $\beta$ and therefore $\alpha$ is willing to accept it,
then α has to update its theory with C. The new information may contradict what α believes and, if so, the new believed information will override what is currently believed by α. β can also propose an update to itself by issuing an internal project β:C.

The semantics of the agents depends also on the goal that each agent has to prove at a certain moment in time. In fact, by proving a goal the agent may abduce hypotheses that explain that goal, and in turn these hypotheses may trigger reactive rules, and so on. Hypotheses abduced in proving a goal 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.

The engineering of our agent society is based exclusively on the notions of projects and updates to model the agent interactions since we aim at building autonomous and distributed agent systems. This form of interaction is powerful and flexible, and a number of communication protocols can be built based on it. However, the main contribution of the paper is to assemble all the ingredients of the agent architecture needed to engineer configurable, dynamic agent societies where the agents can self-organize themselves with respect to their goals, and self-evolve. In this way, the overall “emerging” structure will be flexible and dynamic: each agent will have its own representation of its organization which, furthermore, is updatable by preferring.

The remainder of the paper is structured as follows. Section 2 presents the logical framework. Section 3 introduces our conception of abductive agents. The declarative and procedural semantics of abductive agents and the agents’ cycle are presented in Section 4, 5, and 6, respectively. Sections 8 and 9 illustrate how the approach functions, including how to prefer abducibles to tackle the problem of multiple hypotheses, and how to perform the interplay between planning and acting. Our agent architecture and implementation are described in Section 10. Finally, Section 11 compares our work with relevant literature and expose future lines of research work.

Sections 2, 3, 4, 5 summarize and recapitulate, in a uniform framework, the work presented in [2, 22, 23], while the new contributions of this paper are gathered in Sections 8, 9 and 10.

2 Logic Programming Framework

Typically, an agent can hold positive and negative information, and it can update its own knowledge with respect to the new incoming information. Thus the language of an agent should be expressive enough to gather both positive and negative information. In order to represent negative information in logic programs, we need a language that allows default negation not A not only in premises of rules but also in their heads\(^1\), i.e., generalized logic programs. It is convenient to syntactically represent generalized logic programs as propositional

\(^1\) For further motivation and intuitive reading of logic programs with default negations in the heads see [4].
Horn theories. In particular, we may treat default negation \textit{not }\textit{A} as a standard propositional variable.

Propositional variables whose names do not begin with “\textit{not}” and do not contain the symbols “:\textquotedblleft,” “\textquotedblright” and “\textasciitilde” are called \textit{domain atoms}. For each domain atom \textit{A} we assume a complementary propositional variable of the form \textit{not }\textit{A}. Domain atoms and negated domain atoms are called \textit{domain literals}.

Communication is a form of interaction among agents. The aim of an agent \textit{\beta} when it communicates a message \textit{C} to an agent \textit{\alpha}, is to make \textit{\alpha} update its current theory with \textit{C} (i.e., to have \textit{\alpha} accept some desired mental state). In turn, when \textit{\alpha} receives the message \textit{C} from \textit{\beta}, it is up to \textit{\alpha} whether or not accept \textit{C}. This form of communication is formalized through the notion of projects and of updates (cf. Def. 8).

Propositional variables of the form \textit{\alpha:C} (where \textit{C} is defined below) are called \textit{projects}, \textit{\alpha:C} denoting the intention (of some agent \textit{\beta}) of proposing the updating of the theory of agent \textit{\alpha} with \textit{C}. Projects can be negated. A negated project of the form \textit{not }\textit{\alpha:C} denotes the intention of the agent of not proposing the updating of the theory of agent \textit{\alpha} with \textit{C}. Projects and negated projects are generically called \textit{project literals}.

Propositional variables of the form \textit{\beta:\neg C} are called \textit{updates}, \textit{\beta:\neg C} denoting an update, proposed by \textit{\beta}, of the current theory (of some agent \textit{\alpha}) with \textit{C}. Updates can be negated. A negated update of the form \textit{not }\textit{\beta:\neg C} in the theory of an agent \textit{\alpha} indicates that agent \textit{\beta} does not have the intention to update the theory of agent \textit{\alpha} with \textit{C}. Updates and negated updates are called \textit{update literals}.

Preference information is used along with incomplete knowledge. In such a setting, due to incomplete or partial knowledge, several models of a program may be possible. Preference reasoning proceeds by choosing among those possible models, by expressing priorities among the rules of the program. Preference information is formalized through the notion of priority atoms. Propositional variables of the form \textit{n}\textsubscript{r} < \textit{n}\textsubscript{u} are called \textit{priority atoms}. \textit{n}\textsubscript{r} < \textit{n}\textsubscript{u} means that rule \textit{r} (whose name is \textit{n}\textsubscript{r}) is preferred to rule \textit{u} (whose name is \textit{n}\textsubscript{u}). Priority atoms can be negated. \textit{not }\textit{n}\textsubscript{r} < \textit{n}\textsubscript{u} means that rule \textit{r} is not preferred to rule \textit{u}. Priority atoms and negated priority atoms are called \textit{priority literals}.

Domain atoms, projects, updates, and priority atoms are generically called \textit{atoms}. Domain literals, project literals, update literals, and priority literals are generically called \textit{literals}.

Let \(K\) be any set of atoms, where \(\alpha\) stands for an agent, such that:

- \textit{false} \in \(K\), and

- \((\alpha:\neg C \in K \text{ and distrust}(\alpha:\neg C) \in K) \text{ iff } \alpha:C \in K\).

Propositional variables of the form \textit{distrust}(\alpha:\neg C) and \textit{false} are domain atoms.

\textbf{Definition 1.} The propositional language \(L_K\) generated by \(K\) is the language which consists of the following set of literals:

\[L_K = \{A, \text{not } A \mid \text{for every } A \in K\}.
\]

Whenever a literal \(L\) is of the form \textit{not }\textit{A}, the literal \textit{not }\textit{L} indicates \textit{A}.
Definition 2 (Generalized rule). A generalized rule in the language $\mathcal{L}_K$ is a rule of the form:

$$L_0 \leftarrow L_1, \ldots, L_n \quad (n \geq 0)$$

where every $L_i (0 \leq i \leq n)$ is a literal in $\mathcal{L}_K$.

We use the following convention. Given a generalized rule $r$ of the form $L_0 \leftarrow L_1, \ldots, L_n$, we write $\text{head}(r)$ to indicate $L_0$, $\text{body}(r)$ to indicate the conjunction $L_1, \ldots, L_n$. We write $\text{body}^+(r)$ to indicate the conjunction of all positive literals in $\text{body}(r)$, and $\text{body}^-(r)$ to indicate the conjunction of all negated literals in $\text{body}(r)$.

Definition 3 (Domain rule). A domain rule $L_0 \leftarrow L_1, \ldots, L_n$ in the language $\mathcal{L}_K$ is a generalized rule in $\mathcal{L}_K$ 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.

In the sequel, we will make use of the following working example (adapted from [6]). In it, rules as well as preferences change over time, showing the need to be able to combine preferences and updates in agents, including the updating of preferences themselves.

**Happy Story.** (1) In the initial situation Maria is living and working everyday in the city. (2) Next, as Maria has received some money, Maria conjures up other, alternative but more costly, living scenarios, namely travelling, settling up on a mountain, or living by the beach. And, to go with these, also the attending preferences, but still in keeping with the work context, namely that the city is better for that purpose than any of the new scenarios, which are otherwise incomparable amongst themselves. (3) Consequently, Maria decides to quit working and go on vacation, supported by her increased wealth, and hence to define her vacation priorities. To wit, the mountain and the beach are each preferable to travel, which in turn gainsays the city. (4) Next, Maria realizes her preferences keep her all the while undecided between the mountain and the beach, and advised by a friend opts for the former.

To keep the notation short, some abbreviations are in order: below we let $c$ stand for “living in the city”, $mt$ for “settling on a mountain”, $b$ for “living by the beach”, $t$ for “travelling”, $wk$ for “work”, vac for “vacation”, and $mo$ for “possessing money”.

Example 1. Let $Q$ be the following set of domain rules underlying the theory of Maria, from the Happy Story in the Introduction.

$$Q = \begin{cases} c \leftarrow \text{not } mt, \text{not } b, \text{not } t & (1) \\ wk & (2) \\ vac \leftarrow \text{not } wk & (3) \\ mt \leftarrow \text{not } c, \text{not } b, \text{not } t, \text{not } mo & (4) \\ b \leftarrow \text{not } c, \text{not } mt, \text{not } t, \text{not } mo & (5) \\ t \leftarrow \text{not } c, \text{not } mt, \text{not } b, \text{not } mo & (6) \end{cases}$$
At present $Q$ has a single 2-valued stable model $\{c, wk\}$, so Maria decides to live in the city. Things change if we add to $Q$ the rule $mo$, indicating Maria to possess fresh money. This being true, $Q$ now exhibits four stable models: $\{c, mo, wk\}$, $\{mt, mo, wk\}$, $\{b, mo, wk\}$, and $\{t, mo, wk\}$. Thus, Maria is now unable to decide where to live.

**Definition 4 (Integrity constraint).** An integrity constraint in the language $\mathcal{L}_K$ is a generalized rule in $\mathcal{L}_K$ whose head is the literal $false$ or $not\ false$.

Integrity constraints are rules that can enforce some condition on states, and in this case they take the form of denials, without loss of generality, in a 2-valued semantics. Domain rules are distinct from integrity constraints and must not be expressed as denials. In domain rules, it is of crucial importance which atom occurs in their head whenever updating an agent’s theory. An integrity constraint can also take the form of the negation of a denial (i.e., its head is the literal $not\ false$). It will be used to reject other integrity constraints.

**Example 2.** The integrity constraint $false \leftarrow rain, m:goToBeach$ in the theory of Maria prevents her from going to the beach when it is raining.

The following definition introduces rules that are triggered bottom-up. To emphasize this aspect we employ a different notation for them.

**Definition 5 (Active rule).** An active rule in the language $\mathcal{L}_K$ is a generalized rule in $\mathcal{L}_K$ whose head $Z$ is a project literal and every literal $L_i \ (1 \leq i \leq n)$ 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, i.e., when their body is true in the semantics of the current state. 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 that is 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, or if some conditions are met, then it executes some goal. External projects instead are performed on other agents, e.g., when an agent wants to update the theory of another agent. A negated project that occurs in the head of an active rule denotes the intention (of some agent) not to perform that project at the current state. Note that projects can only occur in the heads of active rules and not in their bodies.

Given an active rule $r$ of the form $L_1, \ldots, L_n \Rightarrow Z$, we write $head(r)$ to indicate $Z$, and $body(r)$ to indicate $L_1, \ldots, L_n$.

**Example 3.** Suppose the underlying theory of Maria (represented by $m$) contains the active rules:

$$R = \left\{ \begin{array}{l} mo, wk \Rightarrow m: not \ wk \\
 b \Rightarrow m:goToBeach \\
 mt \Rightarrow p:goToMountains \end{array} \right\}$$

\(^2\) With the term stable model we intend the “classical” stable model as defined in [29].
The heads of the first two active rules are projects internal to Maria. The first rule states that if Maria has money and she is working, then she wants to update her own theory with \textit{not wk}. The head of the last active rule is an external project: if Maria wants to travel to the mountains, she proposes to go to the mountains to her agent Pedro (represented by \( p \)).

\textit{Example 4}. An active rule \textit{workingDay, not boss \vdash vac \Rightarrow m:goToOffice} in the theory of Maria instructs her to go to office if it is a working day and her boss has not told her that she is on vacation (i.e., he has not yet wanted to update her theory with \textit{vac}).

To express preference information in logic programs, we introduce the notion of priority rule. Let \( N = \{ n_r, \ldots, n_u \} \) be a name set containing a unique name for every domain rule and active rule in the language \( \mathcal{L}_K \). Given a rule \( r \), we write \( n_r \) to indicate its name. We assume that priority atoms take the form \( n_r < n_u \), where \( \{ n_r, n_u \} \subseteq N \).\(^3\) \( n_r < n_u \) means that rule \( r \) is preferred to rule \( u \). We assume that names in \( N \) do not include \( \langle \).

\textbf{Definition 6 (Priority rule).} A priority rule \( L_0 \leftarrow L_1, \ldots, L_n \) in the language \( \mathcal{L}_K \) is a generalized rule in \( \mathcal{L}_K \) 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.

\textit{Example 5}. Let \( Q \) be the set of domain rules of Example 1 and \( P \) the following set of domain and priority rules:

\[
P = Q \cup \left\{ \begin{array}{ll}
4 < 6 \leftarrow \text{vac} & 1 < 4 \leftarrow \text{wk} \\
5 < 6 \leftarrow \text{vac} & 1 < 5 \leftarrow \text{wk} \\
6 < 1 \leftarrow \text{vac} & 1 < 6 \leftarrow \text{wk} \\
5 < 1 \leftarrow \text{vac}
\end{array} \right. \]

The first three priority rules say that rule 1 is preferable to rules 4, 5 and 6 if \( wk \) holds. If \( vac \) holds, then rules 4 and 5 are both preferable to rule 6, which in turn is preferable to rule 1. The intuition is that as \( wk \) holds, while \( vac \) does not, the first three priority rules are used to characterize the preferred model \( \{ c, wk \} \) of \( P \). Thus, in reducing the number of models of a program, priority rules allow us to select among several alternative models.

Since priority rules are generalized rules, preferences can also be updated. For example, we can update the priority rules of the program \( P \) in the previous example by updating it with rule \( 4 < 5 \leftarrow \text{vac} \). In this case \( P \) will have a unique model also in case \( vac \) holds.

\textbf{Definition 7 (Query).} A query in \( \mathcal{L}_K \) takes the form \( ?- \ L_1, \ldots, L_n \ (n \geq 0) \) where every \( L_i \) (\( 1 \leq i \leq n \)) is a domain literal, an update literal, or priority literal in \( \mathcal{L}_K \).

\(^3\) In order to establish the preferred abductive stable models (cf. Def. 16), we require the relation induced by \( < \) to be a well-founded, strict partial ordering on generalized rules.
Definition 8 (Projects, updates). Let $\alpha$ be an agent and $C$ either a domain rule, an integrity constraint, an active rule, a priority rule or a query. Then, $\alpha : C$ is a project and $\alpha \vdash C$ is an update.

Thus, a project (and similarly for updates) can only take one of the forms:

$$\alpha : (L_0 \leftarrow L_1, \ldots, L_n)$$
$$\alpha : (\text{false} \leftarrow L_1, \ldots, L_n, Z_1, \ldots, Z_m)$$
$$\alpha : (\lnot L_1, \ldots, L_n)$$

3 Abductive Agents

This section presents the conception of abductive agent. The knowledge of an agent can dynamically evolve when the agent receives new knowledge, albeit by self-updating rules (not just external ones), or when it temporarily abducts 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 domain atoms (abducibles).

Definition 9 (Updating program). 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 performable by any agent, they do not 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 $U = \{U^s \mid s \in S\}$ is a set of updating programs $U^s$ superscripted by the states $s \in S$.

Let $A \subseteq L_\mathcal{K}$ be a set of domain literals distinct from true and distrust(.). We call the domain atoms in $A$ 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 hypothetical solution or 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 in order to adopt an abducible as a new rule.

Definition 10 (Agent $\alpha$ at a state). Let $s \in S \cup \{0\}$ be a state. An agent $\alpha$ at state $s$, written as $\Psi_\alpha^s$, is a pair $(\mathcal{A}, U)$, where $\mathcal{A}$ is the set of allowed abducibles and $U$ is a sequence of updating programs $\{U^1, \ldots, U^s\}$. If $s = 0$, then $U = \{\}$.

An agent $\alpha$ 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, $\alpha$ is defined by $(\mathcal{A}, \{U^1\})$, where $U^1$ is the updating program containing all the updates that $\alpha$ has received at state 0, either from other agents or as self-updates. In general, an agent $\alpha$ at state $s$ is defined by $\Psi_\alpha^s = (\mathcal{A}, \{U^1, \ldots, U^s\})$, where each $U^i$ is the updating program containing the updates that $\alpha$ has received at state $i - 1$. Within logic
Example 6. Consider the "Happy Story" presented in Section 2. The dynamic evolution of this story can be captured by the different states in which Maria can be found. At state 0, the theory of Maria is $\Psi_0 = (A, \mathcal{U})$, where $\mathcal{U} = \{\}$ and $A = \{\}$ since she does not have any abducible available. Suppose that at state 1 the theory of Maria is $\Psi_1 = (A, \{U^1\})$, where $U^1 = \{t \vdash r \mid \text{for every rule } r \in Q \cup R\}$ where $Q$ is the set of domain rules in Example 1, $R$ is the set of active rules of Example 3 and $t$ is an agent that provides the initial theory to Maria. At the next state, Maria receives the information from the lottery (represented with $l$) that she has received some prize money. Thus, at state 2 the theory of Maria is $\Psi_2 = (A, \{U^1, U^2\})$, with $U^2 = \{l \vdash mo\}$. So, at this state, Maria decides not to work, issues a self-update to that effect, and moves into the next state. Hence, we have that $\Psi_3 = (A, \{U^1, U^2, U^3\})$, with $U^3 = \{m \vdash \text{not wk}\}$. At the last state, consider the situation where Maria receives an advice from her friend Pedro telling her to prefer mountains to beaches for a vacation. The theory of Maria is now $\Psi_4 = (A, \{U^1, U^2, U^3, U^4\})$ with $U^4 = \{p \vdash (4 < 5 \leftarrow \text{vac})\}$.

4 Declarative Semantics

This section introduces the declarative semantics for agents at a given state. In the remainder of the paper, by a 2-valued interpretation $M$ of $\mathcal{L}_K$ we mean any set of literals from $\mathcal{L}_K$ that satisfies the condition that for any atom $A$ in $K$, precisely one of the literal $A$ or $\text{not } A$ belongs to $M$. Given an interpretation $M$ we define:

$$M^+ = \{A \mid A \in M\}$$

$$M^- = \{\text{not } A \mid \text{not } A \in M\} = \{\text{not } A \mid A \notin M\}$$

We say that an interpretation $M$ satisfies a conjunction of literals $\text{WConj}$, and write $M \models \text{Conj}$, iff all literals in $\text{Conj}$ belong to $M$.

4.1 Abductive stable models

Definition 11 (Default assumptions). Let $P$ be a set of generalized rules over the language $\mathcal{L}_K$ and $M$ an interpretation of $\mathcal{L}_K$. The set of default assumptions$^4$ is:

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

$^4$ Note that positive abducibles are defined false by default whenever they are not abduced.
The knowledge of an agent $\alpha$ is characterized at the start (at state 0) by the set of all default assumptions not $A$ for every $A \in K$ (that is, by $\text{Default} (\emptyset, M)$). Its knowledge can dynamically evolve when $\alpha$ receives new knowledge, via a sequence of updating programs $U = \{U^1, \ldots, U^s\}$. 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. Similarly to the rationale of Dynamic Logic Programming [4], the rules proposed via updates can be added to the knowledge state of $\alpha$ provided that $\alpha$ does not explicitly distrust the update. For instance, distrust of an update may be elected by $\alpha$ when it goes against $\alpha$’s presently assumed hypotheses, or if it conflicts with $\alpha$’s previous knowledge and $\alpha$ does not desire it. For example, if an agent $\alpha$ receives an update of the form $\beta \vdash C$ and $\alpha$ does not distrust it, then the rule $C$ will be added to the knowledge state of $\alpha$. Similarly also to Dynamic Logic Programming, the role of updating is to ensure that the rules contained in these newly added updates are in force, and that previous rules are still valid (by inertia) as far as possible, i.e., they are in force to the extent that they do not conflict with newly added rules, and lose force if they so conflict and the new rules remain in force themselves. This rationale is at the basis of the notion of rejected rules, spelled out below.

Here we extend the approach proposed in [4] to handle integrity constraints and active rules. A rule $r$ proposed to an agent $\alpha$ via an update in $U^i$ by an agent $\beta$ is rejected at a state $s$ of $\alpha$ by an interpretation $M$ if there exists a rule $r'$ proposed to $\alpha$ via a subsequent update in $U^j$ by some agent $\alpha$ (possibly $\beta$ again), such that the head of $r'$ is the complement of the head of $r$, the body of $r'$ is true in $M$ and the update is not distrusted by $\alpha$.

**Definition 12 (Rejected rules).** Let $s \in S \cup \{0\}$ be a state. Let $U = \{U^i \mid i \in S\}$ be a sequence of updating programs over the language $L_K$ and $M$ an interpretation of $L_K$. The set of rejected rules at state $s$ is:

$$\text{Reject}(U, s, M) = \{r \mid \exists (\beta \vdash r) \in U^i \land \exists (\delta \vdash r') \in U^j \text{ such that } i < j \leq s, $$

$$\text{head}(r) = \text{not head}(r'), M \models \text{body}(r') \text{ and } M \not\models \text{distrust}(\delta \vdash r')\}$$

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 $\alpha$ can prevent any type of updates from an agent $\delta$ via the use of distrust in the theory of $\alpha$, e.g., $\text{distrust}(\delta \vdash C) \leftarrow \text{liar} (\delta)$.

**Example 7.** Let $U = \{U^1, U^2, U^3\}$ where $U^1 = \{\alpha \vdash a, \alpha \vdash \text{not } c \leftarrow d, \alpha \vdash \text{liar} (\delta), \alpha \vdash (\text{distrust} (\delta \vdash c) \leftarrow \text{liar} (\delta))\}$, $U^2 = \{\beta \vdash (\text{not } a \leftarrow e)\}$, $U^3 = \{\delta \vdash c, \alpha \vdash e\}$ and $M = \{\text{liar} (\delta), \text{distrust} (\delta \vdash c), e\}$. Then, the set of rejected rules $\text{Reject}(U, 3, M)$ is $\{a\}$.

Mark that:

- every generalized rule (i.e., domain rules, priority rules, integrity constraints, and active rules) can be rejected.
whenever an agent distrusts an update proposed by another, the proposed rule is not in force.
- at state 0 no rule can be rejected, i.e., \( \text{Reject}(\mathcal{U}, 0, M) = \{\} \).
- all non rejected rules persist by inertia.
- \text{distrust}(\cdot)\) can itself be updated.
- it is not required of \( r' \) itself not to be rejected to allow it to reject \( r \). Note that this does not cause problems since if \( r \) is itself rejected, then some subsequent rule has reinstated the conclusion of \( r \) and so the rejection of \( r \) is immaterial.

Domain rules defining abducibles can also be rejected. Abducibles may have defining rules in the theory of an agent \( \alpha \) because communicated by other agents via updates, or because of a self-update, like in the case where there exists an active rule of the form \( a \implies \alpha : a \) (with \( a \in \mathcal{A} \)). Thus, an agent may assume new hypotheses to motivate an observation, and later on discard those hypotheses because they turn out (on performing subsequent observations) to be incorrect, or because the world has evolved and those hypotheses have become obsolete.

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. Since integrity constraints are rules that enforce some condition on states, upon their rejection that enforcement is discharged.

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

Note that the abducibles are treated as a virtual update. That is, to compute the abductive stable models the abducibles abduced by \( \alpha \) at state \( s \) are treated as if they were temporary internal updates of \( \alpha \) at virtual state \( s + 1 \). The virtual update is only used to compute the abductive stable models, and there is no commitment to what \( \alpha \) will effectively receive as update at the next state \( s + 1 \).

**Definition 13 (Abductive stable model of agent \( \alpha \) at state \( s \) with hypotheses \( \Delta \)).** Let \( s \in S \cup \{0\} \) be a state. Let \( \psi^\alpha_s = (\mathcal{A}, \mathcal{U}) \) be an agent \( \alpha \) at state \( s \) and \( M \) an interpretation of \( \mathcal{L}_K \). Let \( \Delta \subseteq \mathcal{A} \) be a set of abducibles and \( \mathcal{U}' = \mathcal{U} + \Delta \) a sequence of updating programs. \( M \) is an abductive stable model of the agent \( \alpha \) at state \( s \) with hypotheses \( \Delta \) iff:

- \( \text{false} \not\in M \)
- \( M = \text{least}(\mathcal{X} \cup \text{Default}(T, M) \cup \bigcup_{1 \leq i \leq s} U^i) \), where:

\[ \text{least}(\mathcal{X} \cup \text{Default}(T, M) \cup \bigcup_{1 \leq i \leq s} U^i) \]

- Alternatively, virtual updates could be replaced by event updates as in LUPS, whose effect appears by means of an update and disappears (if not otherwise re-instated) by means of a second update.
\[ X = T - R \]
\[ T = \{ r \mid \exists (\delta \vdash r) \text{ in } \bigcup_{1 \leq i \leq s+1} U^i \text{ such that } M \not\models distrust(\delta \vdash r) \} \]
\[ R = Reject(U', s + 1, M) \]

The definition of abductive stable model semantics is based on the stable model semantics. The use of \( \text{least}(\ldots) \) for generalized logic programs derives from the following two equivalent definitions \([4]\).

A 2-valued consistent interpretation \( M \) is a stable model of a generalized logic program \( P \):

- if \( M \) is the least model of the Horn theory \( P \cup M^- \), i.e. \( M = \text{least}(P \cup M^-) \);
- if \( M = \{ L \mid L \text{ is an atom and } P \cup M^- \vdash L \} \).

**Example 8.** Consider the program:

\[
\begin{align*}
  a & \leftarrow \text{not} \ b \\
  c & \leftarrow \text{not} \\
  e & \leftarrow \text{not} \ b \\
  d & \leftarrow \text{not} \ c \\
\end{align*}
\]

and let \( K = \{ a, b, c, d, e \} \). This program has precisely one stable model \( M = \{ a, c, \text{not} b, \text{not} c, \text{not} d \} \). \( M \) is a stable models since:

\[ M = \text{least}(P \cup \{ \text{not} b, \text{not} c, \text{not} d \}) \]

Following established tradition, we omit negated atoms when describing interpretations and models.

Note that abducing the negation of an abducible \( A \) is tantamount to a virtual update with \( \text{not} A \). Now abducibles may be abduced true or false explicitly.

**Example 9.** Consider the theory \( \Psi^3 = (A, U) \), where \( A = \{ \} \) and \( U = \{ U^1, U^2, U^3 \} \) is defined as in Example 7. The unique abductive stable model of \( \Psi^3 \) is \( M = U^1 \cup U^2 \cup U^3 \cup \{ \text{liar}(\gamma), \text{distrust}(\gamma \vdash c), e \} \) as we have that:

\[
\begin{align*}
  T &= \{ r \mid \exists (\alpha \vdash r) \in (U^1 \cup U^2 \cup U^3 - \{ \gamma \vdash c \}) \} \\
  U' &= \{ U^1, U^2, U^3, \{ \} \} \\
  R &= \text{Reject}(U', 4, M) = \{ a \} \\
  X &= T - R \\
  Default(T, M) &= \{ \text{not} a, \text{not} c, \text{not} d, \text{not} e \}
\end{align*}
\]

Thus, it holds that \( M = \text{least}(X \cup Default(T, M) \cup U^1 \cup U^2 \cup U^3) \).

The next example illustrates the use of abducibles in computing the abductive stable models.

**Example 10.** Consider a situation where an agent Elizabeth (represented by \( e \)) is told by an agent \( \alpha \) that if there is fire then there is smoke, and to sound off the alarm to the fire brigade (represented by \( f \)) in case of fire. Later on, Elizabeth is told by another agent \( \beta \) that if John is smoking then there is smoke, and in
such a case she should scream at him. Moreover, for Elisabeth, both fire and the
fact of John being smoking are abducible hypotheses, i.e. are hypotheses that
Elisabeth is willing to assume in case they serve for explaining her observations.

The theory of Elisabeth can be formalized as \( \Psi_2^E = \{ \mathcal{A}, \mathcal{U} \} \) where \( \mathcal{A} = \{ \text{fire, smoking(john)} \} \) and \( \mathcal{U} = \{ U^1, U^2 \} \):

\[
U^1 = \begin{cases}
\alpha \vdash (\text{smoke} \leftarrow \text{fire}) \\
\alpha \vdash (\text{fire} \Rightarrow f : \text{alarm})
\end{cases}
\]

\[
U^2 = \begin{cases}
\beta \vdash (\text{smoke} \leftarrow \text{smoking(john)}) \\
\beta \vdash (\text{smoking(john)} \Rightarrow j : \text{scream})
\end{cases}
\]

Consequently, at state 2 the model \( M_1 = U^1 \cup U^2 \cup \{ \text{smoke, fire, f:alarm} \} \) is an
abductive stable model of Elisabeth with hypotheses \( \Delta_1 = \{ \text{fire} \} \). Indeed, we have that:

\[
T = \{ r \mid \exists (\alpha \vdash r) \in (U^1 \cup U^2 \cup \Delta_1) \} \quad U' = \{ U^1, U^2, \Delta_1 \}
\]

\[
R = \text{Reject}(U', 3, M_1) = \{ \} \quad \mathcal{X} = T
\]

\[
\text{Default}(T, M_1) = \{ \text{not smoking(john), not j:scream} \}
\]

Thus, it holds that \( M_1 = \text{least}(\mathcal{X} \cup \text{Default}(T, M_1) \cup U^1 \cup U^2 \cup \Delta_1) \). Note that
\( M_2 = U^1 \cup U^2 \cup \{ \text{smoke, smoking(john), j:scream} \} \) is the other abductive stable
model of Elisabeth with hypotheses \( \Delta_2 = \{ \text{smoking(john)} \} \).

**Example 11.** Let \( \Psi_m^1 = (\mathcal{A}, \{ U^1 \}) \) be the theory of Maria at state 1, where \( \mathcal{A} \) and
\( U^1 \) are defined as in Example 6. Suppose that at state 1 Maria has gotten the
information from the lottery (represented with \( l \)) that she has won some prize
money (i.e., she has received the update \( l: \text{mo} \)). Then, at state 2 the theory of
Maria is \( \Psi_m^2 = (\mathcal{A}, \{ U^1, U^2 \}) \), where \( U^2 = \{ l: \text{mo} \} \). Maria has now four
abductive stable models:

\[
M_1 = U^1 \cup U^2 \cup \{ c, wk, mo, 1<4, 1<5, 1<6, m : \text{not wk} \}
\]

\[
M_2 = U^1 \cup U^2 \cup \{ mt, wk, mo, 1<4, 1<5, 1<6, m : \text{goToMountains} \}
\]

\[
M_3 = U^1 \cup U^2 \cup \{ b, wk, mo, 1<4, 1<5, 1<6, m : \text{goToBeach} \}
\]

\[
M_4 = U^1 \cup U^2 \cup \{ t, wk, mo, 1<4, 1<5, 1<6 \}
\]

For example, \( M_1 \) is an abductive stable model since we have that:

\[
T = \{ r \mid \exists (\alpha \vdash r) \in (U^1 \cup U^2) \} \quad U = \{ U^1, U^2, \} \}
\]

\[
R = \text{Reject}(U', 3, M_1) = \{ \} \quad \mathcal{X} = T - R
\]

\[
\text{Default}(T, M_1) = \{ \text{not mt, not b, not t, not vac} \}
\]

\[
\cup \{ \text{not 4<6, not 5<6, not 6<1} \}
\]

\[
\cup \{ \text{not m : goToBeach, not p : goToMountains} \}
\]

Thus, it holds that \( M_1 = \text{least}(\mathcal{X} \cup \text{Default}(T, M_1) \cup U^1 \cup U^2) \).
4.2 Preferred abductive stable models

While updates allow us to deal with a dynamically evolving world, where rules change over time, preferences allow us to choose among various possible models of a state of the world and among possible incompatible reactions. In [6], the notion of preferences of [15] has been combined with updates by establishing two criteria 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 but whose body is defeated by the model.

Definition 14 (Unsupported generalized rule). Let $P$ be a set of generalized rules in the language $L_K$ and $M$ an interpretation of $L_K$. The set of unsupported generalized rules of $P$ and $M$ is:

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

Definition 15 (Unpreferred generalized rule). Let $P$ be a set of generalized rules in the language $L_K$ and $M$ an interpretation of $L_K$. Unpref$(P, M)$ is a set of unpreferred generalized rules of $P$ and $M$:

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

where $\mathcal{X} = \{ r \in P \mid \exists u \in (P - \text{Unpref}(P, M)) \text{ such that: } M \models n_u < n_r, M \models \text{body}^+(u) \text{ and } \neg\text{head}(u) \in \text{body}^-(r) \text{ or } \neg\text{head}(r) \in \text{body}^-(u), M \models \text{body}(r) \}$.

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 preferred abductive stable model of an agent $\alpha$ at a state $s$ with set of hypotheses $\Delta$. Given a sequence of updating programs $U$ and the hypotheses $\Delta$ assumed at state $s$ by $\alpha$, a preferred abductive stable model of $\alpha$ at state $s$ with hypotheses $\Delta$ is a stable model of the program $X$ that extends $P$ to contain all the updates in $U$, all the hypotheses in $\Delta$, 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 16 (Preferred abductive stable model of agent $\alpha$ at state $s$ with hypotheses $\Delta$). Let $s \in S \cup \{0\}$ be a state. Let $\Psi^s_\alpha = (A, U)$ be an agent $\alpha$ at state $s$ and $M$ an abductive stable model of $\alpha$ at state $s$ with hypotheses $\Delta$. Let $U' = U + \Delta$ be a sequence of updating programs. $M$ is a preferred abductive stable model of agent $\alpha$ at state $s$ with hypotheses $\Delta$ iff:

- $\forall r_1, r_2 : (n_{r_1} < n_{r_2}) \in M, \text{ then } (n_{r_2} < n_{r_1}) \notin M$
- $\forall r_1, r_2, r_3 : (n_{r_1} < n_{r_2}) \in M \text{ and } (n_{r_2} < n_{r_3}) \in M, \text{ then } (n_{r_1} < n_{r_3}) \in M$
- $\text{false} \notin M$
\[ M = \text{least}(\mathcal{X} \cup \text{Default}(T, M) \cup \bigcup_{1 \leq i \leq s} U^i), \]

where:

\[
\begin{align*}
\mathcal{X} &= (T - R) - \text{Unpref}(T - R, M) \\
T &= \{ r \mid \exists (\delta \vdash r) \text{ in } \bigcup_{1 \leq i \leq s+1} U^i \text{ such that } M \not\models \text{distrust}(\delta \vdash r) \} \\
R &= \text{Reject}(U', s + 1, M)
\end{align*}
\]

\( T \) is the set containing all the rules in updates that are trusted from \( \alpha \) 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.

In general, at a certain state an agent \( \alpha \) may have several preferred abductive stable models. Note that if \( s = 0 \), then the first condition of the definition above reduces to:

\[
\begin{align*}
T &= \Delta \quad R = \{ \} \quad \mathcal{X} = \Delta \\
M &= \text{least}(\Delta \cup \text{Default}(\Delta, M)).
\end{align*}
\]

When the language contains only domain rules and priority rules (that is, there are neither active rules, integrity constraints nor abducibles), the semantics reduces to the Preferential semantics of Brewka and Eiter [15]. If integrity constraints and active rules are then introduced, the semantics then generalizes to the Updates and Preferences semantics of Alferes and Pereira [6], which extends Dynamic Logic Programming with preferences. Our semantics takes the latter and complements it with abducibles, and with mutual and self-updates by means of active rules and projects.

**Definition 17 (Abductive explanation of agent \( \alpha \) at state \( s \) for query \( Q \)).** Let \( s \in S \cup \{0\} \) be a state and \( \Psi^s_\alpha = (\mathcal{A}, \mathcal{U}) \) an agent \( \alpha \) at state \( s \). Let \( Q \) be a query. An abductive explanation of agent \( \alpha \) at state \( s \) for \( Q \) is any set \( \Theta \subseteq \mathcal{A} \) of hypotheses such that there exists a preferred abductive stable model \( M \) of \( \alpha \) at \( s \) with hypotheses \( \Theta \), and \( M \models Q \).

Mark that at state \( s \) an agent \( \alpha \) may have several abductive explanations for a query \( Q \).

**Example 12.** Let \( \Psi^1_m = (\mathcal{A}, \{U^1\}) \) be the theory of Maria at state 1 as defined in Example 6. Then, \( M_1 = U^1 \cup \{ c, wk, 1<4, 1<5, 1<6 \}^6 \) is the unique preferred stable model of \( \Psi^1_m \) with hypotheses \( \Delta = \{ \} \). In fact, we have that:

\[
\begin{align*}
T &= \{ r \mid \exists (\alpha \vdash r) \in U^1 \} \quad U' = \{ \} \quad R = \text{Reject}(U', 2, M_1) = \{ \}
\end{align*}
\]

\[
\text{Default}(T, M_1) = \{ \text{not mt}, \text{not b}, \text{not t}, \text{not vac}, \text{not mo} \}
\]

\[
\cup \{ \text{not } 4<6, \text{not } 5<6, \text{not } 6<1 \}
\]

\[
\cup \{ \text{not } m:(\text{not } wk), \text{not } m : \text{goToBeach}, \text{not } p : \text{goToMountains} \}
\]

\[
\text{Unpref}(T - R, M_1) = \{ \} \quad \mathcal{X} = T
\]

Thus it holds that \( M_1 = \text{least}(\mathcal{X} \cup \text{Default}(T, M_1) \cup U^1) \).

[^6]: Note that we do not write default atoms in models. Thus a model \( \{a, \text{not } b\} \) is written as \( \{a\} \).
Example 13. Suppose now that at state 1 Maria has gotten the information from the lottery (represented with $l$) that she has won some prize money (i.e., she has received the update $l = mo$). At state 2 the theory of Maria is $\Psi_m^2 = (A, \{U^1, U^2\})$, where $A$ and $U^1$ are defined as in Example 12 and $U^2 = \{l = mo\}$. Then, $M_2 = U^1 \cup U^2 \cup \{e, wk, mo, 1<4, 1<5, 1<6, m:not \ wk\}$ is the unique preferred stable model of Maria at state 2. In fact, we have that:

$$T = \{ r | \exists (\alpha \vdash r) \in (U^1 \cup U^2) \}$$

$$U' = \{U^1, U^2, \{\}\}$$

$$R = \text{Reject}(U', 3, M_2) = \{\}$$

$$\text{Default}(T, M_2) = \{ \text{not } mt, \text{not } b, \text{not } t, \text{not } vac \}$$

$$\cup \{ \text{not } 4<6, \text{not } 5<6, \text{not } 6<1 \}$$

$$\cup \{ \text{not } m : \text{goToBeach}, \text{not } p : \text{goToMountains} \}$$

$$\text{Unpref}(T - R, M_2) = \{4, 5, 6\}$$

Thus, it holds that $M_2 = \text{least}(X \cup \text{Default}(T, M_2) \cup U^1 \cup U^2)$.

Not always do the priority rules allow us to select exactly one model. This may happen when the priority rules do not specify a complete linear order among the alternative choices as shown by the example below.

Example 14. Since the unique model $M_2$ of Maria at state 2 contains the internal project $m:not \ wk$, at the next state the theory of Maria is $\Psi_m^3 = (A, \{U^1, U^2, U^3\})$, where $A$, $U^1$, and $U^2$ are the same as in Example 13 and $U^3 = \{m \vdash \text{not } wk\}$. At state 3 Maria has two preferred stable models:

$$M_3 = U^1 \cup U^2 \cup U^3 \cup \{ \text{vac, mt, mo, 4<6, 5<6, 6<1, 5<1, p : \text{goToMountains}} \}$$

$$M_4 = U^1 \cup U^2 \cup U^3 \cup \{ \text{vac, b, mo, 4<6, 5<6, 6<1, 5<1, m : \text{goToBeach}} \}$$

$M_3$, for example, is a preferred stable model since we have that:

$$T = \{ r | \exists (\alpha \vdash r) \in (U^1 \cup U^2 \cup U^3) \}$$

$$U' = \{U^1, U^2, U^3, \{\}\}$$

$$R = \text{Reject}(U', 4, M_3) = \{2\}$$

$$\text{Default}(T, M_3) = \{ \text{not } c, \text{not } b, \text{not } t \}$$

$$\cup \{ \text{not } 1<4, \text{not } 1<5, \text{not } 1<6 \}$$

$$\cup \{ \text{not } m : \text{goToBeach}, \text{not } m : \text{not } wk \}$$

$$\text{Unpref}(T - R, M_3) = \{1, 6\}$$

Thus it holds that $M_3 = \text{least}(X \cup \text{Default}(T, M_3) \cup U^1 \cup U^2 \cup U^3)$.

If the priority rules do specify a complete linear order among the alternative choices then there will exist a unique preferred abduction stable model, if one exists at all. Consider the situation where Pedro advises Maria to prefer mountains to beaches for vacation. Thus, at the next state Maria receives the update $p \vdash (4<5 \leftarrow \text{vac})$ from her friend Pedro.
Example 15. Let $\Psi^4_{\text{m}} = (\mathcal{A}, \{U^1, U^2, U^3, U^4\})$ be the theory of Maria at state 4, where $\mathcal{A}, U^1, U^2,$ and $U^3$ are the same as in Example 14 and $U^4 = \{p \vdash (4 < 5 \leftarrow \text{vac})\}$. As the set of unpreferred rules at state 4 is $\{1, 5, 6\}$, Maria has a unique preferred abductive stable model:

$$M5 = U^1 \cup U^2 \cup U^3 \cup U^4 \cup \{\text{vac, mt, mo, } 4 < 5, 4 < 6, 5 < 6, 6 < 1, p: \text{goToMountains}\}$$

5 Proof theory

In this section we present a syntactical transformation which forms the basis of a proof procedure for abductive agents. We begin in Section 5.1 by showing how to compile a sequence of updating programs into generalized logic programs, and then in Section 5.2 how to compile preferences.

5.1 Compiling Updating Programs

A transformational semantics for dynamic logic programs has been presented in [4]. According to it, a sequence of generalized logic programs is translated into a single generalized program whose stable models are in one-to-one correspondence with the stable models of the dynamic logic program. In this section we adapt that transformation to cope with priority rules, integrity constraints, and active rules; and we add also new generalized rules to make explicit which rules in the program are rejected. These modifications make the transformation suited to be further extended to incorporate preference information as well (in Section 5.2).

By $\hat{K}$ we denote the following superset of the set $K$ of atoms:

$$\hat{K} = K \cup \{A^-, A_i^-, A_P, A_P^r | A \in K, i \in S \cup \{0\} \cup \{\text{ reject}(n_r) | n_r \in N\} \cup \{u\}.$$

where $N$ is a name set containing a unique name for every domain rule and active rule in the language $L_K$, and $n_r$ is the name of a rule $r$.

This definition assumes that the original set $K$ does not contain any of the newly added atoms $A^-, A_i^-, A_P, A_P^r, \text{ reject}(n_r)$ and $u$, so that they are all disjoint sets.

Let $[.]$ be a function from literals to domain atoms, where $[A] = A$ for every atom $A$ and $[\text{ not } A] = A^-$ for every negated atom $\text{ not } A$. We abbreviate $[L_1], \ldots, [L_n]$ by $[L_1, \ldots, L_n]$.

Definition 1. Let $s \in S \cup \{0\}$ be a state. Let $\mathcal{U} = \{U^i | i \in S\}$ be a sequence of updating programs. Then $\text{DLP}(s, \mathcal{U})$ is the generalized logic program over the language $L_{\hat{K}}$ consisting of the following rules:

(RUP) Rewritten updating programs:

$$H_{P_i} \leftarrow [\text{Body}], \text{ distrust}(\alpha \leftarrow r)^- \quad (1)$$

or

$$H_{P_i}^r \leftarrow [\text{Body}], \text{ distrust}(\alpha \leftarrow r)^- \quad (2)$$
for any update $\alpha \vdash r$, where $r$ is a rule of the form:

$$H \leftarrow \text{Body} \quad \text{or} \quad \text{Body} \Rightarrow H$$

respectively, of the form:

$$\neg H \leftarrow \text{Body} \quad \text{or} \quad \text{Body} \Rightarrow \neg H$$

in the updating program $U^i$ for all $i \in S$. The rewritten rules are obtained from the original ones by replacing atoms $H$ (resp. $\neg H$) occurring in their heads by the atoms $H_{P_i}$ (resp. $\neg H_{P_i}$) and by replacing negated literals $\neg A$ in their bodies by $A^-$. Note, crucially, that the atoms in bodies of generalized rules are not subscripted with any state and are therefore always evaluated at the current state.

(RU) Rewritten updates:

$$(\alpha \vdash r)_{P_i}$$

for any update $\alpha \vdash r$ in the updating program $U^i$ for all $i \in S$. The rewritten updates are obtained from the original ones by replacing any $\alpha \vdash r$ by the corresponding atom $(\alpha \vdash r)_{P_i}$.

(UR) Update rules:

$$A_i \leftarrow P_i$$

$$A_i^- \leftarrow A_{P_i}^-$$

for all atoms $A \in K$ and all $i \in S$. The update rules state that $A$ must be true (resp. false) in the state $i \in S$ if it is true (resp. false) in the updating program $P_i$.

(IR) Inheritance rules:

$$A_i \leftarrow A_{i-1}, \neg A_{P_i}^-$$

$$A_i^- \leftarrow A_{i-1}^-, \neg A_{P_i}$$

for all atoms $A \in K$ and all $i \in S$. The inheritance rules say that $A$ is true (resp. false) in a state $i \in S$ if it is true (resp. false) in the previous state $i-1$ and it is not forced to be false (resp. true) by the updating program $P_i$.

If we need the ability to later on impose restrictions on the inheritance rules, we may add to the rules a reserved predicate refuse which can be extended with additional cases:

$$A_i \leftarrow A_{i-1}, \neg \text{refuse}(A_{i-1})$$

$$A_i^- \leftarrow A_{i-1}^-, \neg \text{refuse}(A_{i-1}^-)$$

$$\text{refuse}(A_{i-1}) \leftarrow A_{P_i}^-$$

$$\text{refuse}(A_{i-1}^-) \leftarrow A_{P_i}$$
(DR) Default Rules:

$$A_0^-$$  \hspace{1cm} (8)

for all atoms $A \in \mathcal{K}$. Default rules describe the initial state 0 in making all atoms initially false.

(RE) Rejection rules:

$$\text{reject}(n_r) \leftarrow H^-_{P_r}$$  \hspace{1cm} (9)

or

$$\text{reject}(n_r) \leftarrow H^-_{P_r}$$  \hspace{1cm} (10)

for any update $\alpha \vdash r$, where $r$ is a rule of the form:

$$H \leftarrow \text{Body} \quad \text{or} \quad \text{Body} \Rightarrow H$$

respectively, of the form:

$$\text{not } H \leftarrow \text{Body} \quad \text{or} \quad \text{Body} \Rightarrow \text{not } H$$

in the updating program $U^i$, and for all $t \in S$ such that $i < t \leq s$. Rejection rules state that a rule $r$ in an update $\alpha \vdash r$ in the updating program $U^i$ will be rejected if it is forced to be false (resp. true) by a subsequent updating program $U^t$.

(CSC) Current State Constraints:

$$\text{false} \leftarrow A, A^-$$  \hspace{1cm} (11)

$$u \leftarrow \text{false}, \text{not } u$$  \hspace{1cm} (12)

for all atoms $A \in \mathcal{K}$. Current state constraints prevent an atom $A$ and its default negation to hold. The second rule allows us to exclude all models containing false.

(CS$_s$) Current State Rules:

$$A \leftarrow A_s$$  \hspace{1cm} (13)

$$A^- \leftarrow A_s^-$$  \hspace{1cm} (14)

$$\text{not } X \leftarrow A_s^-$$  \hspace{1cm} (15)

for all atoms $A \in \mathcal{K}$. Current state rules specify the current state $s$ in which the dynamic prioritized program is being evaluated and determine the values of the atoms $A, A^-$ and not $A$.

The next theorem proves the correctness of the $DLP(s,U)$ transformation. The proof of this and the next proposition (presented in [2]) can be found in [3].

Theorem 1 (Soundness and Completeness). Let $s \in S \cup \{0\}$ be a state and $\Psi_{\alpha} = (A,U)$ an agent $\alpha$ at state $s$. Let $\triangle \subseteq A$ be a set of abducibles. The (regular) stable models of $DLP(s + 1,U + \triangle)$, restricted to $L_{\mathcal{K}}$, coincide with the abductive stable models of $\alpha$ at state $s$ with hypotheses $\triangle$.
The following proposition shows that an atom $\text{reject}(n_r)$ belongs to a stable model $M$ of $\text{DLP}(s, \mathcal{U})$ transformation iff the rule $r$ is effectively rejected by the model.

**Proposition 1.** Let $s \in S \cup \{0\}$ be a state and $\Psi^s_\alpha = (\mathcal{A}, \mathcal{U})$ an agent $\alpha$ at state $s$. Let $\triangle \subseteq \mathcal{A}$ be a set of abducibles. Let $M$ be a (regular) stable model of $\text{DLP}(s+1, \mathcal{U} + \triangle)$ and $J = M \cap \mathcal{L}_K$. Then

$$\text{reject}(n_r) \in M \iff r \in \text{Reject}(\mathcal{U} + \triangle, s+1, J).$$

### 5.2 Compiling Preferences

In this section we show how the Updates Plus Preferences approach [6] for dynamic prioritized programs can be encoded within stable model semantics. This encoding was first presented in [2] and the proofs of its properties in [3].

The encoding consists essentially of two parts: the encoding $\text{DLP}(s, \mathcal{U})$ for updating generalized and priority rules, and the new encoding $\tau(r)$ for taking into consideration preference information. The encoding $\tau(r)$ is based on the encoding developed in [18] for preferences alone (cf. [19] for a general framework for expressing preference information in logic programming expanding [18]). This encoding relies on several predicates that enable us to detect when a rule $r$ has been applied $\text{ap}(n_r)$, blocked $\text{bl}(n_r)$, unsupported $\text{ko}(n_r)$, and define a predicate $\text{ok}(n_r)$ that enables application of rules. If $r$ is OK, then $r$ can be applied or otherwise found to be inapplicable, that is, $r$ is blocked. $\text{ok}(n_r)$ relies on an auxiliary predicate $\text{ry}(n_r, n_u)$. The idea of the translation is to control the order of rule application with respect to the priority information. This is achieved in the following way: If a rule $r$ is preferable to a rule $u$ (i.e., $n_r < n_u$), then $\text{ok}(n_u)$ holds if $r$ is OK and either $r$ has been applied (i.e., $\text{ap}(n_r)$) or found to be inapplicable (i.e., $\text{bl}(n_r)$). Basically, the translation delays the consideration of less preferred rules until the applicability question has been settled for higher ranked rules. If there exists a rule in the program that cannot be proved OK, then the model does not respect the preference information and therefore it is not preferred.

In order to combine the $\tau(r)$ encoding with the $\text{DLP}(s, \mathcal{U})$ encoding, we have adapted the encoding in [18] to take into account the language $\mathcal{L}_K$ of $\text{DLP}(s, \mathcal{U})$. Thus, we have substituted each default atom $\neg A$ in a rule $r$ with $A^-$. Since in the updating phase rules can be rejected, and therefore cannot be used within the preferring phase, we are required to consider them not in $\tau(r)$. This is achieved via the predicate $\text{reject}(n_r)$. For the head of a rule $r$ to be true, we further require that $r$ be not rejected. Moreover, a rule is only enabled applicable if each more preferred rule has been applied, has been blocked, or has been rejected. The integrity constraint in $\tau(r)$ rules out unpreferred models. A model is unpreferred if there exists a rule that is not rejected in the updating phase and cannot be proved OK.

By $\overline{K}$ we denote the following superset of the set $K$ of atoms:

$$\overline{K} = \hat{K} \cup \{\hat{A}, \hat{A}^- \mid A \in K\} \cup \{\text{ko}(n_r), \text{ap}(n_r), \text{bl}(n_r), \text{ry}(n_r, n_u), \{n_r, n_u\} \subseteq N\}.$$
Given a rule \( r \), we write \( \tilde{r} \) to indicate the rule obtained from \( r \) by replacing each domain atom \( A \) occurring in \( r \) with \( \tilde{A} \), and each default atom \( \text{not} \ A \) with \( \text{not} \ \tilde{A} \).

For instance, if \( r \) is the rule \( \text{not} \ A \leftarrow A_1, \ldots, A_n, \text{not} \ A_{n+1}, \ldots, \text{not} \ A_m \), then \( \tilde{r} \) is \( \text{not} \ \tilde{A} \leftarrow \tilde{A}_1, \ldots, \tilde{A}_n, \text{not} \ \tilde{A}_{n+1}, \ldots, \text{not} \ \tilde{A}_m \).

**Definition 2.** Let \( s \in S \cup \{0\} \) be a state and \( \mathcal{U} = \{U^1, \ldots, U^s\} \) a sequence of updating programs over the language \( L^\alpha \). Let \( \mathcal{P} \) be the set of all the domain rules and active rules \( r \) such that \( \exists (\alpha : r) \in U^i \), for some \( 1 \leq i \leq s \). Suppose \( \mathcal{P} = \{r_1, \ldots, r_k\} \). Then \( \Gamma(s, \mathcal{U}) \) is the following generalized logic program over the language \( L^\alpha \):

\[
\Gamma(s, \mathcal{U}) = DLP(s, \mathcal{U}) \cup \bigcup_{r \in \mathcal{P}} \tau(r) \cup DA \cup SPO.
\]

\( \tau(r) \) Rules: consists of the following collection of rules, for \( A \in \text{body}^+(r) \), \( \text{not} \ B \in \text{body}^-(r) \) and any rule \( u \in \mathcal{P} \):

\[
\begin{align*}
\text{[head}(\tilde{r})] & \leftarrow \text{ap}(n_r), \text{not reject}(n_r) \\
\text{ap}(n_r) & \leftarrow \text{ok}(n_r), \text{[body}(\tilde{r})], \text{[body}^-(\tilde{r})] \\
\text{bl}(n_r) & \leftarrow \text{ok}(n_r), \tilde{A}^-, \tilde{A}^- \\
\text{bl}(n_r) & \leftarrow \text{ok}(n_r), B, \tilde{B} \\
\text{ok}(n_r) & \leftarrow \text{ry}(n_r, n_{r_1}), \ldots, \text{ry}(n_r, n_{r_k}) \\
\text{ry}(n_r, n_u) & \leftarrow \text{not} \ (n_u < n_r) \\
\text{ry}(n_r, n_u) & \leftarrow (n_u < n_r), \text{ap}(n_u) \\
\text{ry}(n_r, n_u) & \leftarrow (n_u < n_r), \text{bl}(n_u) \\
\text{ry}(n_r, n_u) & \leftarrow \text{ko}(n_u) \\
\text{ry}(n_r, n_u) & \leftarrow \text{reject}(n_u) \\
\text{false} & \leftarrow \text{not} \text{ok}(n_r), \text{not reject}(n_r) \\
\text{ko}(n_r) & \leftarrow \text{[head}(\tilde{r}), B
\end{align*}
\]

\( \text{(DA) Default Atom Rules:} \)

\[
\tilde{A}^- \leftarrow \text{not} \ \tilde{A} \quad (16)
\]

\( \text{for all atoms} \ A \in K. \)

\( \text{(SPO) Strict Partial Order Constraints:} \)

\[
\begin{align*}
\text{false} & \leftarrow n_{r_1} < n_{r_1} \quad (17) \\
\text{false} & \leftarrow n_{r_2} < n_{r_1}, n_{r_2} < n_{r_2}, (n_{r_1} < n_{r_2})^- \quad (18)
\end{align*}
\]

\( \text{for every generalized rule} \ \{r_1, r_2, r_3\} \subseteq \mathcal{P}. \)

The next result proves the correctness of the \( \Gamma(s, \mathcal{U}) \) transformation. Its proof can be found in [3].

**Theorem 2 (Soundness and Completeness).** Let \( s \in S \cup \{0\} \) be a state and \( \Psi^\alpha_s = (A, \mathcal{U}) \) an agent \( \alpha \) at state \( s \). Let \( \Delta \subseteq A \) be a set of abducibles. The (regular) stable models of \( \Gamma(s + 1, \mathcal{U} + \Delta) \), restricted to \( L^\alpha \), coincide with the preferred abductive stable models of \( \alpha \) at state \( s \) with hypotheses \( \Delta \).
Next we summarize the technical properties of the $\tau(r)$ translation.

**Proposition 2.** Let $s \in S \cup \{0\}$ be a state and $\Psi_\alpha^s = (A, \mathcal{U})$ an agent $\alpha$ at state $s$. Let $\Delta \subseteq A$ be a set of abducibles and $M$ a (regular) stable model of $\Gamma(s + 1, \mathcal{U} + \Delta)$. Let $\mathcal{P}$ be the set of all the domain rules and active rules $r$ such that $\exists (\alpha \vdash r) \in U^i$, for some $1 \leq i \leq s$. Then the following properties hold:

1. $\forall r \in \mathcal{P}$ if reject($n_r$) $\notin M$, then ok($n_r$) $\in M$.
2. $\forall r \in \mathcal{P}$ if reject($n_r$) $\notin M$, then $[\text{ap}(n_r) \in M \iff \text{bl}(n_r) \notin M]$.
3. $\forall r \in \mathcal{P}$ $\text{ko}(n_r) \in M$ iff $r \in \text{Unsup}(\mathcal{P}, M)$.
4. $\forall r \in \mathcal{P}$ if reject($n_r$) $\notin M$, then $[\text{ko}(n_r) \implies \text{bl}(n_r)]$.

5.3 title

More comments on the updates and preferences implementation availability and site, independently of the agent implementation.

6 Agent Cycle

In this section we describe the behaviour of an agent by defining the meaning and execution of queries and projects. We omit the details to keep the description general and abstract.

Every agent $\alpha$ can be thought of as a pair $\Psi_\alpha = (A, \mathcal{U})$, where $A$ is a set of abducibles and $\mathcal{U}$ is a sequence of updating programs, equipped with a set of inputs represented as updates. The abducibles are used as explanations for proving the goals of the agent, and updates can be used to solve the goals as well as to trigger new goals. The basic “engine” schema of an agent $\alpha$ requires an abductive logic programming proof procedure, executed via the cycle represented in Fig. 1.

\[
\text{Cycle}(\alpha, s, \Psi_\alpha^{s-1}, G), \text{ where } \Psi_\alpha^{s-1} = (A, \mathcal{U}) \text{ and } \mathcal{U} = \{U^1, \ldots, U^{s-1}\}.
\]

1. Observe and record any input in the updating program $U^s$.
2. Select a goal $g$ in $G \cup \text{Goals}(U^s)$ and execute $g$ with respect to the program $P = \Gamma(s, \mathcal{U} \cup \{U^s\})$. Let $\Delta \subseteq A$ be an abductive explanation of $g$, if $g$ is provable; otherwise, let $\Delta = \emptyset$. Let $G' = G \cup \text{Goals}(\alpha, U^s) - \{g\}$.
3. Execute all the projects in $\text{ExecProj}(\Delta, P)$.
4. Cycle with $(\alpha, s + 1, \Psi_\alpha^s, G')$, where $\Psi_\alpha^s = (A, \mathcal{U} \cup \{U^s\})$.

Fig. 1. The agent cycle

**Step 1:** The cycle of an agent $\alpha$ 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:** A goal $g$ is selected from $G \cup \text{Goals}(\alpha, U^s)$, where

\[
\text{Goals}(\alpha, U^s) = \{\neg g \mid \alpha \vdash \neg g \in U^s\}.
\]
Note that only the goals issued by the agent $\alpha$ itself are executed. The goals issued by other agents are treated as normal updates. Then $g$ is executed with respect to the generalized logic program $P = \Gamma(s, \mathcal{U} \cup \{U^s\})$, obtained by updating the current one (i.e., $\Gamma(s-1, \mathcal{U})$) with the updating program $U^s$. Here, we can use any abductive proof procedure, such as ABDUAL [7], or preprocess $P$ into sent SModels or DLV, as we have done (the use of a recent version of XSB facilitates the link from the preprocessor into SModels; cf. URL of implementations). In Section 10 we provide more details on the implementation.

**Step 3:** The executable projects are executed. The set $\text{ExecProj}$ of executable projects of an agent depends on the kind of agent we want to model. In case of a cautious agent, the executable projects of the agent are all the projects that belong to every preferred abductive stable model with hypotheses $\Delta$ of $P$,

$$\text{ExecProj}(\Delta, P) = \{ \beta : C \mid \text{for every preferred abductive stable model } M \text{ with hypotheses } \Delta \text{ of } P \text{ it holds that } \beta : C \in M \}. $$

In case of a brave agent, the executable projects are all the projects that belong to at least one preferred abductive stable model $M$ with hypotheses $\Delta$ of $P$. If an executable project takes the form $\beta : C$ (meaning that agent $\alpha$ intends to update the theory of agent $\beta$ with $C$ at state $s$), then (once executed) the update $\alpha \vdash C$ will be available as input to the cycle of agent $\beta$. Executable projects can be thought of as outputs into the environment, and observations as inputs from the environment. From every agent’s viewpoint, the environment contains all other agents.

**Step 4:** Finally, the agent cycles by increasing its state index, by incorporating the updating program $U^s$ into $\mathcal{U}$, and with the new list $G'$ of goals.

Initially, the cycle of $\alpha$ is $\text{Cycle}(\alpha, 1, \Psi^{\alpha}_0, \{\})$ with $\Psi^{\alpha}_0 = (A, \{\})$.

Asynchronous communication among agents is tractable with the techniques spelled out in [21].

### 7 Temporary Updates

Until now we have considered permanent updates: whenever an agent $\alpha$ 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 holds 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. To do so, we employ projects of the form

---

8 In this way, $\alpha$ retains control upon deciding on which goals (requested by other agents) to execute. For example, the theory of $\alpha$ may contain the active rule: $\beta : (!g), \text{Cond} \Rightarrow \alpha : (!g)$ which states that if $\alpha$ has been requested to prove a goal $g$ by $\beta$ and some condition Cond holds, then $\alpha$ will issue the internal project to prove the goal $g$. 

\(\alpha : \text{once}(\ldots)\). The active rule:

\[
\text{cond } \Rightarrow \alpha : \text{once}(C)
\]

in the theory of an agent \(\alpha\), states that if the conditions \text{cond} holds at the current state (in the theory of \(\alpha\)), then \(\alpha\) 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 first formalize a counter that allows an agent to count its states.

\[
\text{counter}(s(0))
\]

\[
\text{counter}(X) \Rightarrow \alpha : \text{not} \text{ counter}(X)
\]

\[
\text{counter}(X) \Rightarrow \alpha : \text{counter}(s(X))
\]

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

By means of the counter, active rules of the form:

\[
\text{cond}_1 \Rightarrow \alpha : \text{once}(a)
\]

\[
\text{cond}_2 \Rightarrow \alpha : \text{once}(a \leftarrow a_1, \ldots, a_n)
\]

\[
\text{cond}_3 \Rightarrow \alpha : \text{once}(a_1, \ldots, a_n \Rightarrow z)
\]

can be coded in our framework as:

\[
\text{cond}_1, \text{counter}(X) \Rightarrow \alpha : (a \leftarrow \text{counter}(s(X)))
\]

\[
\text{cond}_2, \text{counter}(X) \Rightarrow \alpha : (a \leftarrow a_1, \ldots, a_n, \text{counter}(s(X)))
\]

\[
\text{cond}_3, \text{counter}(X) \Rightarrow \alpha : (\text{counter}(s(X)), a_1, \ldots, a_n \Rightarrow z)
\]

8 Preferring Abducibles

In our framework we defined the priority relation over generalized rules. A possible question is: Can we also express preferences over abducibles? Being able to do so will allow us to compare the competing explanations for an observed behaviour. The evaluation of alternative explanations is one of the central problems of abduction. Indeed, an abductive problem of a reasonable size (for example in diagnosis) may have a combinatorial explosion of possible explanations to handle. Thus, it is important to generate only the explanations that are relevant for the problem at hand. Several proposals about how to evaluate competing explanations have been proposed. Some of them involve a “global” criterion against which each explanation as a whole can be evaluated. A general drawback of those approaches is that global criteria are generally domain independent and computationally expensive. An alternative to global criteria for competing alternative assumptions is to allow the theory to contain rules encoding domain specific information about the likelihood that a particular assumption be true. In our approach we can express preferences among abducibles to discard the unwanted assumptions. Preferences over alternative abducibles can be coded into
cycles over negation, and preferring a rule will break the cycle in favour of one abducible or another. Let $\triangleright$ be a binary predicate symbol whose set of constants includes all the abducibles in $\mathcal{A}$. $a \triangleright b$ means that abducible $a$ is preferred to abducible $b$.

**Example 16.** Consider a situation where an agent $\alpha$ drinks either tea or coffee (but not both). Suppose that $\alpha$ prefers coffee to tea when sleepy. This situation can be represented by a set $Q$ of generalized rules in the language $L_{K_Q}$ generated by the set of propositional variables:

$$K_Q = \{ drink, tea, coffee, sleepy \} \cup \{ a \triangleright b \mid \text{for every } a, b \in \mathcal{A}_Q \}$$

with the set of abducibles $\mathcal{A}_Q = \{ tea, coffee \}$.

$$Q = \begin{cases} 
\text{drink} \leftarrow \text{tea} \\
\text{drink} \leftarrow \text{coffee} \\
\text{coffee} \triangleright \text{tea} \leftarrow \text{sleepy}
\end{cases}$$

Notice that the preference relation $\triangleright$ expresses preference among abducibles that are alternative. Therefore, $Q$ need not contain the two rules:

- $\text{tea} \leftarrow \text{not coffee}$
- $\text{coffee} \leftarrow \text{not tea}$

expressing that the abducibles $\text{tea}$ and $\text{coffee}$ exclude one another. In our framework, $Q$ can be coded into the following set $P$ of generalized rules in the language $L_{K_P}$ generated by the set of propositional variables:

$$K_P = \{ drink, tea, coffee, sleepy \} \cup \{ \text{confirm}(x), \text{expect}(x), \text{expect_not}(x) \mid \text{for every } x \in \mathcal{A}_Q \}$$

with the set of abducibles $\mathcal{A}_P = \{ \text{abduce} \}$. The role of the abducible $\text{abduce}$ is to enact the assumption of one of the alternative assumptions $\text{tea}$ or $\text{coffee}$ needed to prove $\text{drink}$.

$$P = \begin{cases} 
\text{drink} \leftarrow \text{tea} \\
\text{drink} \leftarrow \text{coffee} \\
\text{coffee} \leftarrow \text{abduce, not tea, confirm(coffee)} \\
\text{tea} \leftarrow \text{abduce, not coffee, confirm(tea)} \\
\text{confirm(tea)} \leftarrow \text{expect(tea), not expect_not(tea)} \\
\text{confirm(coffee)} \leftarrow \text{expect(coffee), not expect_not(coffee)} \\
\text{expect(tea)} \\
\text{expect(coffee)} \\
1 < 2 \leftarrow \text{sleepy, confirm(coffee)}
\end{cases}$$

The rules (1) and (2) code the alternative assumptions $\text{tea}$ and $\text{coffee}$ into cycles over negation. The rule (1) says that $\text{coffee}$ can be assumed if $\text{abduce}$ has been
abduced, tea is not assumed, and coffee is confirmed. Confirming an assumption $x$ means that one has an expectation for $x$ while he does not have an expectation for not $x$. Having the notion of expectation allows one to express the preconditions for an expectation for $x$. If the preconditions do not hold, then $x$ cannot be confirmed, and therefore $x$ will not be assumed. For example, the rules:

\[
\begin{align*}
\text{expect(tea)} &\leftarrow \text{have\_tea} \\
\text{expect(coffee)} &\leftarrow \text{have\_coffee}
\end{align*}
\]

express that one has an expectation for tea and coffee if he has them. By means of $\text{expect\_not}$ one can express situations where he does not expect something. For example, the rule

\[
\text{expect\_not(coffee)} \leftarrow \text{blood\_pressure\_high}
\]

states not to expect coffee if one has high blood pressure. In this case, coffee will not be confirmed, and therefore tea will be assumed. The last rule in $P$ is a priority rule stating that rule (1) is preferable to rule (2) if sleepy holds and coffee can be confirmed.

Note that in our framework it is also possible to assume negative abducibles. This can be achieved by a simple generalization of the rules in $P$.

Suppose that at state 0 $\alpha$ is told by another agent $\beta$ of the rules in $P$. Thus, at state 1 the theory of $\alpha$ is $\Psi^1_\alpha = (\mathcal{A}, \{U^1\})$ with $U^1 = \beta \div P$.\footnote{Given an agent $\beta$ and a set $P$ of generalized rules, we write $\beta \div P$ to indicate the set $\{\beta \div x \mid \text{for every } x \in P\}$.} At state 1, $\alpha$ has two preferred abductive stable models with hypotheses $\Delta = \{\text{abduce}\}$:

\[
\begin{align*}
M_1 &= \{\text{abduce}, \text{confirm(tea)}, \text{confirm(coffee)}, \text{expect(tea)}, \text{expect(coffee)}, \text{coffee, drink}\} \\
M_2 &= \{\text{abduce}, \text{confirm(tea)}, \text{confirm(coffee)}, \text{expect(tea)}, \text{expect(coffee)}, \text{tea, drink}\}
\end{align*}
\]

The number of abductive stable models reduces to one if at a later state $\alpha$ is told that sleepy holds. In that case, the unique abductive stable model of $\alpha$ would be:

\[
M_3 = \{\text{abduce, confirm(tea), confirm(coffee), expect(tea), expect(coffee), coffee, drink, 1<2}\}.
\]

By removing the abducible in $\Delta$, the priority atoms, and atoms of the form confirm(…), expect(…), and expect\_not(…) from the abductive stable models, we obtain the intended models of $\alpha$. This technique can be generalized to prefer among sets of abducibles.

Another application of expressing preferences over abducibles is exploratory data analysis which aims at suggesting a pattern for further inquiry, and contributes to the conceptual and qualitative understanding of a phenomenon. Assume that a unexpected phenomenon, $x$, is observed by an agent $\alpha$, and that $\alpha$ has three possible hypotheses (abducibles), $a$, $b$ and $c$, that are capable of explaining it. In exploratory data analysis, after observing some new facts, we abduce explanations and explore them to check predicted values against observations. Though
there may be more than one convincing explanation, we abduce only the more plausible ones. Suppose explanations $a$ and $b$ are both more plausible than (and therefore preferable to) explanation $c$. This can be expressed as:

$$Q = \left\{ \begin{array}{l} x \leftarrow a \\ x \leftarrow b \\ x \leftarrow c \end{array} \right\} \cup \left\{ \begin{array}{l} a \triangleleft c \\ b \triangleleft c \end{array} \right\}. $$

with abducibles $\mathcal{A} = \{a, b, c\}$. $Q$ can be represented by the set of generalized rules with abducibles $\Delta = \{\text{abduce}\}$:

$$P = \left\{ \begin{array}{l} x \leftarrow a \\ x \leftarrow b \\ x \leftarrow c \\ a \leftarrow \text{abduce}, \text{not } b, \text{not } c, \text{confirm}(a) \quad (1) \\ b \leftarrow \text{abduce}, \text{not } a, \text{not } c, \text{confirm}(b) \quad (2) \\ c \leftarrow \text{abduce}, \text{not } a, \text{not } b, \text{confirm}(c) \quad (3) \\ \text{confirm}(a) \leftarrow \text{expect}(a), \text{not } \text{expect}_\text{not}(a) \\ \text{confirm}(b) \leftarrow \text{expect}(b), \text{not } \text{expect}_\text{not}(b) \\ \text{confirm}(c) \leftarrow \text{expect}(c), \text{not } \text{expect}_\text{not}(c) \\ \text{expect}(a) \\ \text{expect}(b) \\ \text{expect}(c) \\ 1 < 3 \leftarrow \text{confirm}(a) \\ 2 < 3 \leftarrow \text{confirm}(b) \end{array} \right\}. $$

The three rules (1), (2) and (3) code the alternative abducibles $a$, $b$ and $c$ into cycles over negation, and the last two priority rules state that the rules (1) and (2) are preferable to the rule (3).

Let us consider a simple example that illustrates explanatory data analysis.

**Example 17.** Consider the following set of generalized rules:

$$Q = \left\{ \begin{array}{l} x \leftarrow a \\ x \leftarrow b \\ x \leftarrow c \end{array} \right\} \cup \left\{ \begin{array}{l} a \triangleleft c \leftarrow \text{not } e \\ b \triangleleft c \leftarrow \text{not } e \\ b \triangleleft a \leftarrow d \end{array} \right\}$$

where the propositional variables have the meaning:

- $x$ - the car does not start,
- $a$ - the battery has problems,
- $b$ - the ignition is damaged,
- $c$ - there is no gasoline in the car,
- $d$ - the car’s radio works,
- $e$ - the wife has used the car, and
- $\text{exp}$ - test if the car’s radio works.
Suppose that at state 1 the theory of $\alpha$ is $\Psi^1_\alpha = (A, \{U^1\})$ with $U^1 = \beta \div P$, where $P$ is a set of generalized rules obtained from $Q$ by applying the same transformation as described above. At state 1, $\alpha$ has two preferred abductive stable models with hypotheses $\Delta = \{\text{abduce}\}$ capable of explaining its observation $x$:

$M_1 = \{\text{abduce}, a, x, 1<3, 2<3, \text{confirm}(c), \text{confirm}(b), \text{confirm}(a), \text{expect}(c), \text{expect}(b), \text{expect}(a)\}$

$M_2 = \{\text{abduce}, b, x, 1<3, 2<3, \text{confirm}(c), \text{confirm}(b), \text{confirm}(a), \text{expect}(c), \text{expect}(b), \text{expect}(a)\}$

By removing the abducibles, the priority atoms, and atoms of the form $\text{confirm}(\ldots)$, $\text{expect}(\ldots)$, and $\text{expect}\_\text{not}(\ldots)$ from $M_1$ and $M_2$, we obtain the intended models of the initial program $Q$, where abducibles $a$ and $b$ are both preferred over $c$.

In this example, we have only a partial priority theory over abducibles. Thus, we cannot select exactly one abducible (i.e., one model), as it were the case had we a complete priority relation over all abducibles in $A$. To prefer between $a$ and $b$, $\alpha$ can perform some experiment $\exp$ to obtain confirmation (by observing the environment) about the most plausible hypothesis. For example, by using the following rules (where $\text{env}$ plays the role of the environment):

\[
\begin{align*}
\{ \text{choose } &\leftarrow a \\
\text{choose } &\leftarrow b \}
\end{align*}
\]

together with

\[
\begin{align*}
a &\Rightarrow \alpha : \text{chosen} \& b &\Rightarrow \alpha : \text{chosen} \\
\text{choose} &\Rightarrow \alpha : \text{not chosen }\Rightarrow \text{env }\exp
\end{align*}
\]

At state 1 $\alpha$ has two hypotheses, $a$ and $b$, that are capable to explain the observed phenomena $x$. Thus, $\alpha$ must discover the correct one. $\alpha$ chooses an hypothesis if $a$ or $b$ (possibly both) holds:

\[
\begin{align*}
\text{choose } &\leftarrow a \\
\text{choose } &\leftarrow b
\end{align*}
\]

With this knowledge, agent $\alpha$ has still two preferred abductive stable models: $M_3 = M_1 \cup \{\text{choose}\}$ and $M_4 = M_2 \cup \{\text{choose}\}$. As $\text{choose}$ holds in both models, the last active rule is triggerable. When triggered, it will add (at the next state) the active rule $\text{not chosen }\Rightarrow \text{env }\exp$ to the theory of $\alpha$, and if $\text{not chosen}$ holds, $\alpha$ will perform the experiment $\exp$. The first two active rules are needed to prevent $a$ by performing $\exp$ when $\alpha$ has chosen one of the abducibles.

9 Reactive Planning

Next we apply our agent theory to the problem of 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, a lot of research in planning (see e.g. [13, 32, 35, 38, 52]) 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 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.

A typical use of abduction is in generating plans in a planning problem (cf. [5, 24, 33]). Suppose the situation where some information of the planning problem is missing or changes during the planning process. To tackle this situation we must be able to make a (possibly partial) plan, to start executing it, and in case something goes wrong, to detect the anomaly and to replan. This entails the ability to detect such cases and to replan in order to achieve the original goal.

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.

9.1 Expressing plans

To express reactive planning we introduce the language $\mathcal{L}_{K,B}$. Given a set $B$ of action names, the language $\mathcal{L}_{K,B}$ extends the language $\mathcal{L}_K$ to contain p-rules and p-active rules. We begin by introducing the notion of plan as a sequence of actions. We write $a_1 \ldots a_k$ to indicate the sequence $a_1, \ldots, a_k$ of action names.

**Definition 3 (Plan).** Let $B$ be a set of action names. Then, a plan is any sequence $a_1 \ldots a_k$ ($k \geq 0$) of action names in $B$.

Intuitively, a plan $a_1 \circ a_2 \circ a_3$ states to execute first $a_1$, then $a_2$, and finally $a_3$. This capability allows an agent to execute actions in sequence. Next, we define rules that allow plans to occur in their body.

**Definition 4 (p-rule).** Let $B$ be a set of action names. A p-rule in the language $\mathcal{L}_{K,B}$ is a rule of the form:

$$A \leftarrow a_1 \circ \ldots \circ a_k \quad (k \geq 0)$$

where $A$ is a domain atom, and $a_1 \circ \ldots \circ a_k$ is a plan in $\mathcal{L}_{K,B}$. $A$ is called p-atom.

Note that plans in the body of p-rules cannot be negated. Given a p-rule $q \leftarrow a_1 \circ a_2$, the query $?-q$ has the effect of launching the execution of the plan $a_1 \circ a_2$. Actions are formalized via the notion of p-active rules.

28
**Definition 5 (p-active rule).** Let $a \in B$ be an action name and $\alpha$ an agent. A p-active rule in the language $L_{K,B}$ is a rule of the form:

$$a, L_1, \ldots, L_n \Rightarrow \alpha : Z \quad (n \geq 0)$$

where every $L_i$ ($1 \leq i \leq n$) is a domain literal or an update literal, and $\alpha : Z$ is a project atom over $L_{K,B}$.

A p-active rule of the form $a, L_1, \ldots, L_n \Rightarrow \alpha : Z$ expresses the preconditions $L_1, \ldots, L_n$ and the effect $\alpha : Z$ of an action $a$. Typically, for every action name $a \in B$ there will be one or more p-active rules defining it. We assume that the body of p-active rules defining the same action are mutually exclusive. Thus, for every action at most one p-active rule will be executed. In the following, we write a p-active rule for an action as $a; precA \Rightarrow \alpha : effectA$.

**Example 18.** Consider an agent $\alpha$ that has a plan consisting of two actions, $a$ followed by $b$. Suppose that the plan is executable if some condition $\text{cond}$ holds. Such a planning problem can be expressed as:

$$\begin{align*}
\text{p} & \leftarrow \text{cond}, q \\
q & \leftarrow a \circ b \\
a, \text{precA} & \Rightarrow \alpha : \text{effectA} \\
b, \text{precB} & \Rightarrow \alpha : \text{effectB}
\end{align*}$$

The execution of the plan $a \circ b$ is launched by means of the query $?-\text{p}$, provided that the preconditions $\text{cond}$ of the plan are fulfilled. When the execution of the plan $a \circ b$ starts, if $\text{precA}$ holds the action $a$ is executed. The effects of executing $a$ are expressed via the project $\alpha : \text{effectA}$. Finally, the action $b$ is executed provided that $\text{precB}$ holds. If the preconditions of any action do not hold, then the plan cannot be accomplished and it is therefore abandoned.

### 9.2 Coding plans

We introduce a syntactic transformation mapping a set of generalized rules, p-rules and p-active rules over the language $L_{K,B}$ into a set of generalized rules over the language $L_{K}$ generated by:

$$K = K \cup \{ \text{exec}(a), \text{once}(\text{exec}(a)), \text{exec}(ta), \text{once}(\text{exec}(ta)) \mid \text{for every } a \in B \}$$

$$\cup \{ \text{exec}(tp), \text{once}(\text{exec}(tp)) \mid \text{for every p-atom } p \in L_{K,B} \}$$

**Definition 6.** Let $B$ be a set of action names and $Q$ a set of generalized rules, p-rules and p-active rules over the language $L_{K,B}$ with set of abducibles $A$. Assume that $Q$ is the underlying theory of an agent $\alpha$. The following transformation $\Psi$ maps $Q$ into a set $\Psi(Q)$ of generalized rules over the language $L_{K}$ with abducibles $A \cup \{ \text{start} \}$.

1. If $r \in Q$ is a p-active rule of the form:

$$a, \text{precA} \Rightarrow \alpha : \text{effectA}$$

then

$$\Psi(r) = \{ \text{exec}(a), \text{precA} \Rightarrow \alpha : \text{effectA} \}$$
2. If \( r \in Q \) is a p-rule of the form:

\[
p \leftarrow a_1 \circ \ldots \circ a_k
\]

then

\[
\Psi(r) = \begin{cases}
  p \leftarrow \text{start} & (1) \\
  \text{start} \Rightarrow \alpha : \text{once}(\text{exec}(a_1)) & (2) \\
  \text{exec}(a_1), \text{precA}1 \Rightarrow \alpha : \text{once}(\text{exec}(t\alpha_1)) & (3) \\
  \text{exec}(t\alpha_1) \Rightarrow \alpha : \text{once}(\text{exec}(a_2)) & (4) \\
  \ldots \\
  \text{exec}(a_k), \text{precA}k \Rightarrow \alpha : \text{once}(\text{exec}(t\alpha_k)) & (2k + 1) \\
  \text{exec}(t\alpha_k) \Rightarrow \alpha : \text{once}(\text{exec}(t\nu)) & (2k + 2)
\end{cases}
\]

A p-active rule \( r \) is transformed (step 1) by \( \Psi \) into a rule where the action name \( a \) in \( r \) is replaced with the domain atom \( \text{exec}(a) \), indicating that the action \( a \) must be executed. Being every literal in \( \text{precA} \) be a domain or an update literal and the effect of the rule \( \alpha : \text{effectA} \) a project atom (by definition of p-active rule), the rule \( \text{exec}(a), \text{precA} \Rightarrow \alpha : \text{effectA} \) is an active rule in the language \( L_K \).

A p-rule is transformed (step 2) by \( \Psi \) into a domain rule together with a number of active rules. The query \( ?-p \), by means of the rule (1), has the effect of abducing \( \text{start} \) which in turn will trigger the active rule (2). When the agent \( \alpha \) executes the project \( \alpha : \text{once}(\text{exec}(a_1)) \), at the next agent cycle it will update its theory with \( \text{once}(\text{exec}(a_1)) \) indicating that it must execute the action \( a_1 \) of the plan. This update will hold only at the next state of \( \alpha \) (temporary update). (For the coding of \( \text{once}(\ldots) \) see Section 7.)

The active rules (3)–(2k+2) model the sequencing of the actions \( a_1, \ldots, a_k \). If the action \( a_1 \) must be executed and its preconditions \( \text{precA}1 \) hold, then the active rule (3) is triggered. Doing so has the effect of making \( \alpha \) (at the next cycle) to temporarily update its theory with \( \text{once}(\text{exec}(a_1)) \) indicating that the execution of \( a_1 \) is terminated. In turn, \( \text{exec}(t\alpha_1) \) will trigger the active rule (4) which temporarily updates the theory of \( \alpha \) with \( \text{once}(\text{exec}(a_2)) \). This means that at the next cycle \( \alpha \) must execute the action \( a_2 \). This proceeds until \( \alpha \) has terminated the execution of \( a_k \) (i.e., \( \text{once}(\text{exec}(t\alpha_k)) \)), the execution of the entire plan is then terminated (rule (2k + 2)).

### 9.3 Executing plans

Plans that can be executed in parallel and plans containing actions that can start a subplanning process can also be expressed in \( L_{K,B} \). The parallel execution of two plans \( p_1 \) and \( p_2 \) can be expressed as:

\[
p \leftarrow p_1, p_2 \\
p_1 \leftarrow a_1 \circ \ldots \circ a_n \\
p_2 \leftarrow b_1 \circ \ldots \circ b_m
\]

30
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 $a_2$ whose effect is to start the execution of a subplan $q$.

\[
p \leftarrow a_1 \odot a_2 \odot a_3 \\
q \leftarrow b_1 \odot b_2 \\
a_2 \Rightarrow \alpha : ?\neg q
\]

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

\[
p \leftarrow a_1 \odot a_2 \odot a_3 \\
a_3 \Rightarrow \alpha : ?\neg \text{planner}(\ldots)
\]

Executing the last action of the plan $a_1 \odot a_2 \odot a_3$ 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:

\[
p \leftarrow a_1 \odot a_2 \odot a_3 \\
a_2 \Rightarrow \alpha : ?\neg b
\]

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

### 9.4 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 [51, 52]. 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 $\mathcal{B}$ is either a sensing action or a non-sensing action. Both sensing and non-sensing actions are formalized via $p$-active rules.

**Definition 7 (Conditional plan).** Let $\mathcal{B}$ be a set of action names.

1. A sequence $a_1 \odot \ldots \odot a_k$ ($k \geq 0$) of non-sensing actions names in $\mathcal{B}$ is a conditional plan.
2. If $a_1 \odot \ldots \odot a_k$ ($k \geq 0$) is a sequence of non-sensing actions names in $\mathcal{B}$, $a_{k+1}$ is a sensing action that determines $f$, and $b$ and $c$ are conditional plans, then $a_1 \odot \ldots \odot a_k \odot a_{k+1} \odot \text{if}(f, b, c)$ is a conditional plan.
3. Nothing else is a conditional plan.
To execute a conditional plan $a_1 \circ \ldots \circ a_k \circ a_{k+1} \circ \text{if}(f, b, c)$, an agent $\alpha$ first executes $a_1 \circ \ldots \circ a_k \circ a_{k+1}$. Then it evaluates $f$: if $f$ is true, it executes $b$ else $c$.

Consider the following planning problem over the language $\mathcal{L}_{K,B}$:

$$p \leftarrow a_1 \circ a_2 \circ \text{if}(f, b_1 \circ b_2 \circ b_3, c_1)$$

where $a_2$ is a sensing action determining $f$, and $b_1 \circ b_2 \circ b_3$ and $c_1$ are conditional plans. The conditional plan above can be first translated into the (ordinary) plan:

$$p \leftarrow a_1 \circ a_2 \circ \text{if}(f, p_1, p_2)$$
$$p_1 \leftarrow b_1 \circ b_2 \circ b_3$$
$$p_2 \leftarrow c_1$$
$$\text{if}(f, p_1, p_2), f \Rightarrow \alpha : ?-p_1$$
$$\text{if}(f, p_1, p_2), \neg f \Rightarrow \alpha : ?-p_2$$

where $\text{if}(f, p_1, p_2)$ is understood as an ordinary action name. This plan can then be coded into a set of generalized rules over the language $\mathcal{L}_K$ via the transformation $\Psi$.

### 9.5 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 over the language $\mathcal{L}_{K,B}$:

$$p \leftarrow a \circ b$$
$$a, \text{pre}A \Rightarrow \alpha : \text{effect}A$$
$$b, \text{pre}B \Rightarrow \alpha : \text{effect}B$$
$$\text{cond}1 \Rightarrow \alpha : ?-\text{stop}(p)$$
$$\text{cond}2 \Rightarrow \alpha : ?-\text{suspend}(p)$$
$$\text{cond}3 \Rightarrow \alpha : ?-\text{resume}(p)$$

where the last three active rules state the conditions to stop, suspend, and resume the plan. This planning problem can be coded into a set of generalized rules over
the language \( \mathcal{L}_K \) as:

\[
\begin{align*}
  p & \leftarrow \text{not block}, \text{start} \quad (1) \\
  \text{block} & \leftarrow \text{cond1} \\
  \text{block} & \leftarrow \text{cond2} \\
  \text{not block}, \text{exec}(a), \text{precA} & \Rightarrow \alpha : \text{effectA} \quad (4) \\
  \text{not block}, \text{exec}(b), \text{precB} & \Rightarrow \alpha : \text{effectB} \quad (5) \\
  \text{start} & \Rightarrow \alpha : \text{once}(\text{exec}(a)) \quad (6) \\
  \text{not block}, \text{exec}(a), \text{precA} & \Rightarrow \alpha : \text{once}(\text{exec}(ta)) \quad (7) \\
  \text{not block}, \text{exec}(ta) & \Rightarrow \alpha : \text{once}(\text{exec}(b)) \quad (8) \\
  \text{not block}, \text{exec}(b), \text{precB} & \Rightarrow \alpha : \text{once}(\text{exec}(tb)) \quad (9) \\
  \text{cond2}, \text{exec}(X) & \Rightarrow \alpha : \text{suspended}(X) \quad (10) \\
  \text{cond3}, \text{suspended}(X) & \Rightarrow \alpha : \text{once}(\text{exec}(X)) \quad (11) \\
  \text{cond3}, \text{suspended}(X) & \Rightarrow \alpha : \text{not suspended}(X) \quad (12)
\end{align*}
\]

where \( \mathcal{A} = \{ \text{start} \} \). The query \(?- p\) has the effect to start the execution of the plan provided that the plan is not blocked, i.e., both \text{cond1} and \text{cond2} do not hold. This is achieved by abducing \text{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 \( \text{not block} \) one of their preconditions, as soon as one of the conditions \text{cond1} or \text{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 \text{cond2} is true, then \text{block} holds and the agent \( \alpha \) updates its theory with the action \text{suspended} (i.e., with \text{suspended}(X)). This is needed to allow \( \alpha \) to resume its plan. If the condition \text{cond3} becomes true, then the plan is resumed by triggering the last two active rules (11) and (12). \( \alpha \) updates its theory with \text{once}(\text{exec}(X)) (rule (11)) indicating that the suspended action \( X \) must be executed and with \text{not suspended}(X) (rule (12)) \( X \) being no longer suspended.

### 9.6 Preferring actions and plans

By combining the ability of preferring and planning one can formalize plans whose actions are the most preferred with respect to some declaratively specified criteria. In our approach we can express preferences among “alternatives” plans to discard the unwanted ones. Preferences over alternative plans can be coded into cycles over negation, and preferring a rule will break the cycle in favor of one plan or another. Let \( \prec \) be a binary predicate symbol whose set of constants includes all the \( p \)-atoms in \( \mathcal{L}_{K,B} \). \( p_1 \prec p_2 \) means that plan \( p_1 \) is preferred to plan \( p_2 \).
Consider the following planning problem over the language $\mathcal{L}_{K,B}$ extended to contain preference literals of the form $p_1 \prec p_2$:

$$
\begin{align*}
p &\leftarrow q_1 \\
p &\leftarrow q_2 \\
q_1 &\leftarrow a_1 \circ \ldots \circ a_n \\
q_2 &\leftarrow b_1 \circ \ldots \circ b_m \\
q_1 \prec q_2 &\leftarrow \text{cond1} \\
q_2 \prec q_1 &\leftarrow \text{cond2}
\end{align*}
$$

together with the p-active rules defining all the actions of the planning problem. $p$ has two alternative plans defined by the p-rules $q_1$ and $q_2$. The last but one rule states that $q_1$ is preferable to $q_2$ if some condition $\text{cond1}$ holds. That means that the plan $a_1 \circ \ldots \circ a_n$ is preferable to the plan $b_1 \circ \ldots \circ b_m$. On the contrary, the last rule states to prefer $q_2$ to $q_1$ in the case that $\text{cond2}$ holds. This planning problem can be first coded into a planning problem over the language $\mathcal{L}_{K,B}$ as:

$$
\begin{align*}
p &\leftarrow \text{pref}(q_1), q_1 \\
p &\leftarrow \text{pref}(q_2), q_2 \\
q_1 &\leftarrow a_1 \circ \ldots \circ a_n \\
q_2 &\leftarrow b_1 \circ \ldots \circ b_m \\
\text{pref}Q_1 &\leftarrow \text{not} \text{pref}Q_2 \quad (r_1) \\
\text{pref}Q_2 &\leftarrow \text{not} \text{pref}Q_1 \quad (r_2) \\
r_1 &< r_2 \leftarrow \text{cond1} \\
r_2 &< r_1 \leftarrow \text{cond2}
\end{align*}
$$

which in turn can be coded into a set of generalized rules over the language $\mathcal{L}_{K}$ via the transformation $\Psi$. In a way similar to the one for plans, actions can also be preferred.

Example 19. Consider a situation where an agent $\alpha$ has the goal to take a plane. $\alpha$ has a three action plan: to prepare its suitcase, to get to the airport, and to do check-in at the airport. To get to the airport, $\alpha$ has two alternatives: to go either by taxi or by train. $\alpha$ prefers the first alternative when the train is delayed, and the second alternative if the train is cheap and not delayed. This situation can be formalized in the language $\mathcal{L}_{K,B}$ extended to contain preference literals of the form $q_1 \prec q_2$, where $q_1$ and $q_2$ are p-atoms. Let $B = \{a, b, c, \text{taxi}, \text{train}\}$.

$$
\begin{align*}
p &\leftarrow a \circ b \circ c & a &\Rightarrow \alpha : d \\
q &\leftarrow q_1 & b &\Rightarrow \alpha : ?-q \\
q &\leftarrow q_2 & c &\Rightarrow \alpha : e \\
q_1 &\leftarrow \text{taxi} & \text{taxi} &\Rightarrow \alpha : f \\
q_2 &\leftarrow \text{train} & \text{train} &\Rightarrow \alpha : f \\
q_1 \prec q_2 &\leftarrow \text{delayed} \\
q_2 \prec q_1 &\leftarrow \text{cheap, not delayed} \\
\text{cheap}
\end{align*}
$$
where the propositional variables have the following meaning:

- $a$ - prepare suitcase
- $b$ - get to airport
- $c$ - do check-in
- $d$ - suitcase prepared
- $e$ - check-in done
- $f$ - at the airport
- $q$ - plan to get to airport

Plan $p$ is executed in stages: after having executed the action $a$ to get the suitcase ready, the action $b$ is attempted. The effect of $b$ is to launch the goal $?-q$, a subplan process whose aim is to get to the airport. The alternative chosen for $q$ (i.e., whether to go by train or taxi) is decided at execution time based on the preference criteria of $a$ and the state of the world (i.e., whether or not the train is delayed).

10 Agent Architecture and Implementation

One instance of the agent framework is implemented as follows: its logical parts (e.g., logical reasoning, preferring, updating, etc.) are implemented in XSB Prolog [12], while its non-logical parts (e.g., agent communication, user interface, etc.) are implemented in Java. We then employ InterProlog [11] to interface Java and the XSB Prolog. The next subsections present an overview of the architecture, its components and the flow of projects (for more details consult [27, 20]).

10.1 Overview

The architecture consists of six components, as illustrated in Figure 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 [34]. Rather, the system has the ability to execute at the same time, asynchronously, several tasks differently located in the agent cycle (defined in Section 6) . 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\textsuperscript{10}.

\textsuperscript{10} 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.
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.

10.2 Project flow

This section illustrates the flow of projects in the agent architecture. This is narrated from the perspective of an agent $\alpha$, depicted in Fig. 2 (b). Suppose that an agent $\beta$ asks agent $\alpha$ to prove a query $\neg g$ via the external project $\alpha : \neg g$. At the next state, the external interface of $\alpha$ receives (step 1) the update $\beta \div \neg g$, indicating that $\beta$ has requested $\alpha$ to prove $g$. In turn, this update is sent to the update handler (step 2) whose task it is to sort the different types of updates.

---

An asynchronous transition rule system that characterizes the interactions among agents is presented in [21].
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 $\beta \vdash ?-g$ 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 $\beta \vdash ?-g$. Note that when $\alpha$ receives a request to prove a query $g$ from another agent $\beta$ (via the update $\beta \vdash ?-g$ in the theory of $\alpha$), $g$ is not proven directly. Instead, $\alpha$ has the ability to decide whether or not to prove $g$. Indeed, $\alpha$ proves $g$ only if $\alpha$ itself posts this request via the internal project $\alpha \vdash ?-g$. Suppose that the knowledge base of the reactive process contains an active rule of the form:

$$\beta \vdash ?-g \Rightarrow \alpha \vdash ?-g$$

saying that, if $\alpha$ receives a request to prove a query $g$ from an agent $\beta$, then $\alpha$ assumes the project to prove $g$ (i.e., $\alpha \vdash ?-g$). At the next state of $\alpha$, the internal project $\alpha \vdash ?-g$ will be executed (step 5). This implies that the update handler will receive (step 6) the update $\alpha \vdash ?-g$ from the reactive process, which in turn sends it to central control (step 7). Central control updates again the knowledge bases of the rational and reactive process with $\alpha \vdash ?-g$ (steps 8a and 8b). Since the query is issued by the agent itself, central control will launch the query $?-g$ (step 9). If $g$ is provable with substitution $\theta$ and list of abduced hypotheses $\Delta$, central control will update the knowledge bases of the rational and reactive process with the answer $\alpha \vdash \text{ans}(g, g\theta, \Delta)$ of the query $g$ (steps 10a and 10b). This allows $\alpha$ to eventually send the answer back to $\beta$ in case its knowledge base contains an active rule of the form:

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

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

### 10.3 Rational process

The rational process models the rational ability of agents. It is implemented by an XSB Prolog process. If an agent $\alpha$ is at state $s$ and $\Psi^\alpha_s = \{A, U\}$, then the knowledge base of the rational process is the generalized logic program $P = \Gamma(s, \Psi^\alpha_s)$ obtained via a syntactic transformation $\Gamma$ presented in [2]. The ABDUAL procedure [7, 8] is employed with respect to $P$ 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 $\theta$ and list of abduced hypotheses $\Delta$, it returns the answer as $\text{ans}(g, g\theta, \Delta)$ to central control which in turn updates the knowledge bases of the rational and reactive processes with $\alpha \vdash \text{ans}(g, g\theta, \Delta)$. 

37
10.4 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 $P = \Gamma(s, U)$\textsuperscript{12}. The task of the reactive process is to trigger any 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 $\neg \text{exec}(La, L)$ to the reactive process, where $La$ is the list of hypotheses abduced by the rational process at state $s$. Executing $\neg \text{exec}(La, L)$ returns a list $L$ of executable projects to central control. Basically, the call to $\text{exec}(\ldots)$ checks the active rules whose body holds in $\Gamma(s, U)$ and returns their heads in the list $L$. $\text{exec}(\ldots)$ is defined by a generalized logic program implementing the behaviour of $\text{ExecProj}(\ldots)$.

The definition of $\text{exec}(\ldots)$ 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 $\triangle$ assumed to prove a query $q$ 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 $a \Rightarrow \alpha : \alpha$ where $\alpha \in A$, states to execute the internal update $\alpha : \alpha$ when the hypothesis $\alpha$ is abduced.

Distinguished projects are those defining the predefined predicates $\text{sendInterrupt}$, $\text{doAct}$, and $\text{stateModify}$. For example, the active rule in the theory of an agent $\alpha$:

$$\beta \div \text{urgentRequest}(G) \Rightarrow \alpha : \text{sendInterrupt}(G, \text{true})$$

instructs $\alpha$ 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 $\text{sendInterrupt}$ is true), just in case an agent $\beta$ has urgently requested it. The active rule:

$$\text{meeting} \Rightarrow \alpha : \text{doAct}(\text{displayMessage})$$

instructs $\alpha$ 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 $\text{stateModify}$. The active rule

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

once triggered allows $\alpha$ to assign the value 100 to the control parameter $\text{reactiveDelayTime}$, in case a dangerous situation supervenes. $\text{reactiveDelayTime}$ is a parameter in central control that defines the time interval between two distinct tests of which active rules of $\alpha$ that can be triggered. Basically, its value

\textsuperscript{12} Recall that the rational and reactive process have the same knowledge base.
defines the level of reactivity of \( \alpha \). The list of parameters that control the agent behaviour can be found in [27].

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.

### 10.5 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 \( \alpha \) 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.

### 10.6 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 `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 has 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 \( \alpha \div \neg g \)
(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 \( \text{ans}(g, g\theta, \Delta) \) and it updates the knowledge bases of the rational and reactive process with it. Then, it launches the query \( \text{exec}(\Delta, L) \) 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 \( \text{stateModify} \) command, and cycles again.

11 Related Work

In this section we relate our approach to other related work. In particular we relate our work with earlier research on preferring abducibles, reactive planning and agent theories and architectures.

11.1 Preferring Abducibles

To evaluate the assumption alternatives of an abductive problem, several approaches have been proposed. In what follows we mention some of the approaches that use local criteria during the evaluation process.

In [16] Console et al consider the case where knowledge about abducibles is available during the abductive process. They distinguish two types of knowledge: (i) taxonomic relationship between abducible atoms, and (ii) constraints between abducibles represented in the form of denials. Constraints are used to represent mutual exclusion conditions, while the taxonomic relation is used to express preference among abducibles according to the principle that more general explanations (i.e., the ones which do not make unnecessary assumptions) should be preferred.

In the context of learning abductive theories for the problem of attribute-based classifications, Dimopoulos and Kakas [25] introduce an abductive framework based on a subsumption relation between the values of the attributes. The subsumption relation defines a value hierarchy that is used to prefer explanations. They prefer explanations that can explain a given tuple with values that are lower in the value hierarchy.

In a different strand of research, Appelt et al and Poole propose a different approach to the problem of preferring competing alternative explanations. Appelt et al [9] propose an approach, called weighted abduction, whose idea is to associate an assumption cost to every conjunct in the body of rules. This cost can then be propagated from the body to the head of rules. The authors present an algorithm for computing a weighted-abductive explanation. Thus, the weight of abductive explanations can be used to prefer the best explanation (i.e., the one with lowest weight). Poole [42] develops a framework for logic-based abduction that incorporates probabilities with assumptions. The idea is to use probabilities to discard unlikely assumptions to reduce the combinatorial explosion in abductive processes (e.g., in diagnostic problems).
Mayer and Pirri [40] present a logical framework (based on a modal logic approach) where the abductive relation (i.e., the relation between a sentence explaining a given observation and the observation itself) can be expressed. In the proposed logic it is possible to formalize preference criteria to select the best explanations via suitable preference relations on formulae.

Sakama and Inoue [46, 47] introduce a framework for representing priorities in logic programming. Prioritized logic program represents preference knowledge that is used to reduce non-determinism in logic programming. The authors show that their approach can realize various forms of commonsense reasoning such as abduction. In [31] Sakama and Inoue tackle the problem of learning abducibles and priorities to derive desired conclusions in non-monotonic reasoning. They propose an abductive framework that enables to infer priorities to explain a given observation.

For complexity results of approaches on logic-based abduction see [26].

11.2 Reactive Planning

Several approaches have been pursued for the integration of planning and execution. In the logic programming framework, Kowalski and Sadri [34] proposed a framework unifying rational and reactive behaviour. They proposed an observe-think-act cycle that allows the interleaving of sensing, planning and execution. In their approach reactivity is wired into the agent cycle and the amount of resources spent on planning is defined via a parameter. However, it is not clear in their approach how the planning process can be interrupted, and therefore exogenous events that call for immediate action are ignored by the planner. To overcome this limitation, Prendinger and Ishizuka [43] introduced a planning system APS with a loose conceptual tie between the planning and the reactive component. APS is a planner that can generate approximate plans and can be interrupted at any time. APS is also able to detect inconsistencies of new observations with an already constructed plan and can launch a replanning mechanism.

In [35] Kumar and Meeden outline a hybrid architecture that takes advantage of the benefits of both the deliberative and the reactive behaviour. They couple their architectural model and the BDI architecture. They accommodate both synchronous and asynchronous sensing facilities in their agent architecture. In our framework we only employ an asynchronous reactive module that can interrupt and launch new goals to the deliberative module. Their synchronous module can be simulated in our approach through the use of abducibles that tie the deliberation and reactive process. Jensen and Veloso [32] noticed that in most agent’s approaches that integrate planning and acting, the agents are not able to respond instantly to changes in the environment during plan execution. This ability is crucial in application domains with a high frequency of dynamic changes, e.g., in a flight control scenario.

Jensen and Veloso [32] proposed an agent architecture that is based on an adaptive interleaving of planning and reactive phases. At each agent cycle, the
response demand\textsuperscript{13} of the agent is measured. If the response demand is low, then the agent enters in a deliberative phase and therefore it can perform planning and execute the actions of the computed plans. If the response demand is high (i.e., the agent must react quickly), the agent enters into a reactive phase where active rules encoding pre-compiled plans can be used for a quick reaction. Thus, in their approach, the agent can adapt dynamically to the changes of the environment and override the planned actions. While this type of adaptive behaviour is hardwired at the architectural level, in our approach it can be declaratively defined. In fact, we can specify active rules whose head defines the value of the response demand of the agent. Then, we can achieve this type of adaptive behaviour through interrupts and via predefined atoms that can change the values of the control variables at the agent’s architectural level (see Section 10 for more details).

In the 90s the Cognitive Robotics group at the University of Toronto developed several logic-based execution languages, e.g., Golog \cite{Golog} and ConGolog \cite{ConGolog}. Golog allows to specify the execution program of a robot via constructs such as conditional statements, non-deterministic choice of action, recursive procedures, etc. As summarized in \cite{Golog}, Golog does not consider exogenous actions and does not allow sensing actions. This makes the language not suitable for a robot situated in an environment where changes beyond the control of the robot may happen. ConGolog is an extension of Golog to allow for interrupts that can suspend or abandon the execution of the current plan. However, ConGolog does not support execution monitoring and therefore it is not possible to specify sensing actions. The ability of execution monitoring has been added to Golog in \cite{GologExecutionMonitoring}. This ability allows a robot to observe the world and to make plans to recover from states of the world reached due to exogenous actions. In \cite{LespéranceConGolog} Lespérance et al showed that ConGolog is a language suitable for the design of high-level reactive control modules for robotics applications. In Section 9 we have shown how to specify sequential executions of actions, parallel plans, etc. It would be of interest to simulate all the planning constructs of ConGolog in our framework. In addition to ConGolog, we can use preferences to prefer actions and plans. This is a desirable feature especially in conjunction with the sensing action ability.

Boella and Damiano \cite{Replanning} presented an algorithm for replanning in a reactive agent architecture. Basically, the behaviour of the agent is controlled by an execution-sensing cycle. After executing each action in the plan, the sensing module monitors the effects of the execution of that action, and updates the agent’s representation of the world. Then, the deliberation module of the agent evaluates the current representation of the world, and if it does not meet the agent’s expectations, a replanning component is invoked on the current plan with the task to find a better plan. In contrast to this approach, in our framework the executing-sensing cycle is not wired into the agent architecture. Rather, it can be declaratively specified at the agent’s theory level. Though the focus of their work is on the replanning algorithm and not on the agent’s architecture.

\textsuperscript{13} The response demand measures how quick the agent’s response should be.
Baral and Tran [13] established a formal connection between action theories and reactive control. They also considered sensing actions and conditional plans. They provided a method for constructing correct control modules.

A formal theory that captures the basic aspects of reactive planning has been proposed by Spalazzi and Traverso [52]. They introduced a language that extends the language of dynamic logic to include the basic operations for failure handling. This allows one to reason about failure detection and recovery in planning, acting and sensing.

11.3 Agent Theories and Architectures

The use of computational logic for modelling multi-agent systems has been widely investigated (e.g., see [1, 45] for a roadmap). One approach close to our own is the agent-based architecture proposed by Kowalski and Sadri [34], 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 [17]. 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 [10] 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 [44] 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 [48]. 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 [45].

An example of a BDI architecture is Interrap [41]. 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 [28].

Sloman et al. [49, 50] 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. [36]. This architecture consists of several components sharing a common knowledge base and performing various tasks, like deliberation, reactivity, planning, etc. All the architectural components share a common representation mechanism to capture knowledge and state transitions.

**Acknowledgements**

This work was partially supported by POCTI project 40958 “FLUX - FleXible Logical Updates”.

**References**

12. XSB-Prolog. Available at xsb.sourceforge.net.


