An implementation of a rational, reactive agent

Mattias Engberg

2003-06-05
An implementation of a rational, reactive agent

Examensarbete utfört i kommunikations och transportsystem
vid Linköpings Tekniska Högskola, Campus Norrköping

Mattias Engberg

Handledare: Prof. L.M. Pereira
Examinator: Ph.D. Pierangelo Dell’Acqua

Norrköping den 5 juni 2003
An implementation of a rational, reactive agent

Mattias Engberg

We are working on the development and design of an approach to agents that can reason, react to the environment and are able to update their own knowledge as a result of new incoming information. In the resulting framework, rational, reactive agents can dynamically change their own knowledge bases as well as their own goals. An agent can make observations, learn new facts and new rules from the environment, and then update its knowledge accordingly. The knowledge base of an agent and its updating mechanism has been implemented in Logic Programming. The agent’s framework is implemented in Java.

This aim of this thesis is to design and implement an architecture of a reactive, rational agent in both Java and Prolog and to test the interaction between the rational part and the reactive part of the agent. The agent architecture is called RR-agent and consists of six more or less components, four implemented in Java and the other two are implemented in XSB Prolog.

The result of this thesis is the ground for the paper “An architecture of a rational, reactive agent” by P. DellAcqua, M. Engberg, L.M. Pereira that has been submitted.

Nyckelord
Agent, logic, ProLog, reactive, rational, Java
Abstract

Logical based agents

We are working on the development and design of an approach to agents that can reason, react to the environment and are able to update their own knowledge as a result of new incoming information. In the resulting framework, rational, reactive agents can dynamically change their own knowledge bases as well as their own goals. An agent can make observations, learn new facts and new rules from the environment, and then update its knowledge accordingly. The knowledge base of an agent and its updating mechanism has been implemented in Logic Programming. The agent’s framework is implemented in Java.

This aim of this thesis is to design and implement an architecture of a reactive, rational agent in both Java and Prolog and to test the interaction between the rational part and the reactive part of the agent. The agent architecture is called RR-agent and consists of six more or less components, four implemented in Java and the other two are implemented in XSB Prolog.

The result of this thesis is the ground for the paper “An architecture of a rational, reactive agent” by P. DellAcqua, M. Engberg, L.M. Pereira that has been submitted.
Appendix A / RR-agent's Java classes................................................................. 35
1 Centralcontrol................................................................................................. 35
2 XSB_obj........................................................................................................... 36
3 Updatehandler ............................................................................................. 38
4 Actionhandler ............................................................................................... 38
5 ProLogCommand ........................................................................................... 39
6 Queue.............................................................................................................. 40

Appendix B / Prolog ......................................................................................... 41

Appendix C / Java............................................................................................... 42

Appendix D / InterProlog: a declarative Java-Prolog interface.......................... 44
1 Purpose

Our aim is to have a simple agent architecture and to test its behavior in different application domains. In particular, we are interested in testing the interaction between the rational/deliberative (D) and reactive (R) behavior of the agent, and its updating mechanism (U).

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

- **D-U**
  If an agent $\alpha$ has a goal to prove and contemporarily some updates to consider, which of the two tasks shall $\alpha$ perform first? The policy of giving priority to goals may decrease the agent performance when a quick reaction is required. In contrast, prioritizing updates can cause unwanted delays of the proof of a goal only to consider irrelevant updates. Another issue concerns the behavior of the agent when is proving a goal $G$ (i.e., it is deliberative) and it receives an update. Should $\alpha$ complete the execution of $G$, and then consider the update? or instead should it suspend the execution of $G$, consider the update and then relaunch $G$?

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

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

The definition of an agent varies depending on the person you ask. There are still more definitions of what an agent is than there are actual finished and working agents. If you ask a researcher in the Artificial Intelligence (AI) field you get one answer, and if you ask a robot manufacturer you get a completely different answer. In the beginning the AI community wanted to set a loosely definition of what an agent was, but today the term can be used in almost any situation and by nearly any program. One example is that if you put the word agent in the software name it will attract more buyers. What most can unite behind is one specific characterization that agents have and it is the ability to react to the environment and autonomously change it.

This chapter gives some historical references in agent programming and thereby tries to explain some different theories in this branch of AI.

2.1 Background

![Figure 1 Fields of Artificial Intelligence](image)

There are several ways to look at, or approach, the field of AI as shown above. It all depends on how you try to explain what AI is. Is AI a way to make computers able to mimic human behavior (i.e. the Turing test)? Is it a way for us to understand how the human mind works? Making the computer able to take decisions based on beliefs? Or is it a way to automate (often) monatomic work? Should it react instantly or reason forth a way to operate? Or is it all the above? Agent theory is closely knitted to the fundamentals of AI, as seen above.

Agent theory is a rather new area and it was not until the later part of the first half of the 80's scientists in the AI community were getting interested. For example, Brooks's paper "A Robust Layered Control System for a Mobile Robot"[1] is an early work on reactive agent
and dates back to 1985. Since then the science community have tried to accomplish the task of defining a common definition of what an agent is. They have not succeeded yet, but some keywords have permeated the discussion. Words like autonomy, flexibility and cooperation are some of the keyword that is central to agent theory.

It was not until mid 90's, especially during 1994, that this field of science started getting large attention and several agent-related publications appeared [2]. The first of now several special issues of the Communications of the ACM on agents appeared in 1994 and it included articles like Maes’ now-classic paper on “Agents that reduce work and information overload” [3] and Norman’s conjectures on “How might people interact with software agents” [4]. Then there was Reinhardt’s Byte article on “The network with smarts” [5]. Indeed, during late 1994 and throughout 1995 and 1996, an explosion of agent-related articles appeared in the popular computing press. It is no coincidence that this explosion coincided with that of the World Wide Web (WWW).

An agent can be almost anything. It can be a booking agent for trips, a virus in a computer, etc. Even a thermostat is a kind of agent. All these above are agents of different kind and usage area. T. Selker at IBM development center expressed already in 1995 that "An agent is a software thing that knows how to do things that you could probably do yourself if you had the time". Well, an agent does not always have to be a software agent, but what Selker really said symbolizes the essence of an agent. To have the ability to react to the environment and being able to perform autonomous tasks based on the environmental input. S. Franklin and A. Graesser [6] take it even further and try to go a step further. In their paper from 1996 they propose a formal definition of an autonomous agent, "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.", which is very similar to what Selker said the year before.

As mentioned above, there are some common features all agents share: to a large extent independently perform tasks on behalf of their contracting party or user for which specialized knowledge is needed and/or which consists of many time intensive tasks. This is probably the strongest argument of what an agent is. But that does not mean an agent have to be an agent all the time. The agent should be able to react to its environment and if it cannot do that, then some argue it is not an agent. Others just say it is dormant or temporary not reactive. Take for example an agent controlled by light, a light sensitive sensor. In a completely dark room the agent cannot act on the environment because it cannot see, so the agent has no idea of what happens around it. As soon as you move an agent and places it in an environment that the agent was not build for it ceases to be an agent or it is temporary not reactive.

The advent of software agents gave rise to much discussion of what such agent is. For now there is no standard definition of what an agent really is, it differs hugely between designer and area of application. B. Herman [7] speaks about weak and strong notions of the concept of agent.

2.1.1 The Weak notion of the concept "agent"

Perhaps the most general way in which the term agent is used, is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:
• **autonomy**: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;

• **social ability**: agents interact with other agents and (possibly) humans via some kind of agent communication language;

• **reactivity**: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it. This may entail that an agent spends most of its time in a kind of sleep state from which it will awake if certain changes in its environment (like the arrival of a new e-mail) give rise to it;

• **proactivity**: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative;

• **temporal continuity**: agents are continuously running processes (either running active in the foreground or sleeping/passive in the background) in contrast to “once-only” computations or scripts that map a single input to a single output and then terminate;

• **goal orientedness**: an agent is capable of handling complex, high-level tasks. The agent must be capable of splitting a task into smaller subtasks, and making the decision in which order and in which way these sub-tasks should be best performed.

Thus, a simple way of conceptualizing an agent is as a kind of UNIX-like software process that exhibits the properties listed above. A clear example of an agent that meets the weak notion of an agent is the so-called *softbot* ('software robot'). The softbot is an agent that is active in a software environment (for instance the previously mentioned UNIX operating system). One example is the daemon for printing in the UNIX-world.

### 2.1.2 The Strong(er) notion of the concept "agent"

For some researchers, particularly those working in the field of AI, the term agent has a stronger and more specific meaning than the ones presented in the previous section. These researchers generally understand an agent to be a computer system that, in addition to having the properties of the weak notion of agent, is either conceptualized or implemented using concepts that are more usually applied to humans. For example, it is quite common in AI to characterize an agent using mentalistic notions, such as knowledge, belief, intention, and obligation. Some AI researchers have gone further, and considered emotional agents.

Agents that fit the stronger notion of agent usually have one or more of the following characteristics:

• **mobility**: the ability of an agent to move around an electronic network;

• **benevolence**: is the assumption that agents do not have conflicting goals, and that every agent will therefore always try to do what has been asked;

• **rationality**: is the assumption that an agent will act in order to achieve its goals in a long-term way and plan its actions to achieve the goal;

• **adaptivity**: an agent should be able to adjust itself to the habits, working methods and preferences of its user;

• **collaboration**: an agent should not unthinking accept (and execute) instructions, but should take into account that the human user makes mistakes (e.g. give an order that contains conflicting goals), omits important information and/or provides ambiguous information. Thus, an agent should check things by asking questions to the user, or use a
built-up user model to solve conflicting situations. An agent should even be allowed to refuse to execute certain tasks, because (for instance) they would put an unacceptable high load on the network resources or because it would cause damage to other users.

Although no single agent possesses all these abilities, there are several prototype agents that possess quite a lot of them. At this moment no consensus has yet been reached about the relative importance (weight) of each of these characteristics in the agent as a whole. What most scientists have come to a consensus about is that it is these kinds of characteristics that distinguish agents from ordinary programs.

These more complex agents also often use *symbolic reasoning*. In symbolic reasoning all input to the agent is interpreted as symbols and the agent reasoning processes operate more like the human brain processes. Another way of giving agents human-like attributes is to represent them visually by using techniques such as a cartoon-like graphical icon or an animated face. Research into this matter has shown that, although agents are pieces of software code, people like to deal with them as if they were dealing with other people (regardless of the type of agent interface that is being used).

### 2.1.3 Agent taxonomy

To examine the agents more closely we need to categorize them, that is, to divide them in subgroups. You can divide them with respect to flexibility, depending on whether they work alone or in a multi-agent environment, the level of self-learning, etc. One common way is to divide them as the animal kingdom, using taxonomies. In general, autonomous agents can be divided into three major groups: biological/human agents, robots (i.e. hardware agents) and computational agents. This thesis treats computational agents so the other types of agents are only briefly mentioned.

![Agent taxonomy](image)

**Figure 2 Agent taxonomy**

**Biological agents**

We meet biological, i.e. human, agents' everyday. If, for example, you need a specific version of ink cartridge for your ink-jet printer, and the store is out of it. The agent, in this case the salesman in the store, looks up the manufacturer, makes the order and when he receives the item, he will hopefully send it to you. You expect the salesman to perform reliably, independently and quickly all tasks concerned with requiring the cartridge. The work of the agent saves you the time to acquiring the specialized knowledge required to find and purchase the cartridge. If this concept is transferred to the area of computers, it immediately produces the main task of the hardware and software agents. Using the analogy of biological agents, the hardware and the software agents should perform certain tasks for their users that they cannot undertake themselves because of insufficient time or lack of knowledge.
Robotic agents

The robotic, or hardware, agent is more common then you may think. The thermostat that controls the temperature in a sauna is one example of a hardware agent, a light sensitive resistor in a lamp is another kind of hardware agent. These agents are more or less complex. The lack of the ability to reflect over what it has been accomplished makes these kinds of agents not intelligent.

Computational agents

The computational agent, i.e. software agents, belongs to the largest group of agents. These agents can be divided into several different subgroups depending on the task of the agent. The major subclassification schemes are via the environments (e.g. database, file system, network, Internet etc.), via the language in which the agent is written, via the applications or via the mentalistic abilities the agent exhibits. The choice of the subclassification scheme depends upon the analyses that the user wants to perform. The rational, reactive agent in this thesis is a computational agent.
2.2 Reactive vs. rational agents

In this thesis computational agents are divided into three subgroups: reactive agents, rational agents and hybrid agents. Reactive agents react to external stimuli instantly while rational agents have the ability to plan and interact in a more complex way. A thermostat and a chess program are an example of a reactive and of an “almost purely” rational agent. A chess program is not purely rational because it has a time-bound to complete the next move (otherwise, it would be purely rational). Finally, a hybrid agent combines the characteristics of reactive and rational agents. The following sections illustrate the features of reactive, rational and hybrid agents.

2.2.1 Reactive agents

Work on reactive agents’ dates back to the work of Brooks in the middle '80s. The most common agent is the reactive since it is the easiest agent to make. The response time of the agent is short and reflective to the surrounding environment changes. It does not necessarily possess any symbolic representation of its environment, i.e. the world in which the agent is situated serves as its own model, and thereby does not perform any symbolic reasoning. It has no explicit goal.

Reactive agents are relatively simple, easy to understand and interact with other agents in basic ways. They are situated, i.e. they do not plan ahead or revise any world models, and their actions depend on what happens at the present moment. However, complex patterns of behavior emerge from these interactions when the reactive agents are viewed globally in a multi-agent system.

The behavior of a reactive agent is regulated by condition-action rules, also known as active rules. They are rules of the form:

IF Condition THEN Action

This rule is executed (triggered) if the condition holds in the environment, and as effect the rule generates an action to be performed. An example of conditions-action rule is:

IF it_is_raining THEN use_umbrella

Such a rule is triggered by an observation of raining and generates the action of using an umbrella as output. Note that the agent does not have any explicit goal or belief. This rule achieves a goal, i.e., not getting wet, which is implicit rather than explicit.

In "Intelligence Without Representation" [8] Brooks heavily criticized the symbolist tradition of AI and proposed three key theses:

1. Intelligent behavior can be generated without explicit representation.
2. Intelligent behavior can be generated without explicit reasoning of the kind that symbolic AI proposes.
3. Intelligence is an emergent property of certain complex systems.
Brooks argued that these theses obviate the need for symbolic representations or models because the world becomes its own best model. Furthermore, this model is always kept up-to-date since the system is connected to the world via sensors and/or actuators.

Brooks identified two key ideas that have characterized his research:

1. Real intelligence is situated in the world, not in systems like theorem provers or expert systems.
2. Intelligent behavior arises as a result of an agent's interaction with its environment.

In order to demonstrate his claims, he built a number of robots based on an abstract architecture, called the subsumption architecture [9]. This architecture consists of a set of modules, each of which is based on a finite state machine. Every finite state machine is triggered into action when its input signal exceeds some threshold. Finite state machine represents the only processing units in this architecture. In it there is no facility for global control and no means for accessing global data.

The modules are grouped and placed into layers that connect sensing to acting and run in parallel. Lower level layers allow the agent to react to important or dangerous events, while modules in a higher level can inhibit modules in lower layers. Each layer has a hardwired purpose or behavior, e.g., to avoid obstacles or to enable/control wandering. In this architecture it is possible to add new functionality by adding new, higher-level layers.

2.2.2 Rational/deliberative agents

Typically, the notion of a rational agent in AI focuses on the thinking process of the agent and ignores its interaction with the environment. A rational agent can be seen as an agent which contains an explicitly represented, symbolic model of the world, and in which decisions are made via logical reasoning based on patterns matching and symbolic manipulation. One example is the use of history- or future-based reasoning. The agent then uses information from different time states to combine into new information and knowledge.

The WLog agent system [10] is an example of a rational agent. The first prototype is used to make a web site more flexible to the user and to finely predict the users needs. It is a multi agent system with several specialized agents for different tasks making it easier to expand and implement new technology. Each agent is identified by its “location” which can be obtained by other agents via the specialized agent “Agent Locator”. The reasoning part of the agent is implemented in DyLOG and connected to a database with the user’s goals and actions.

The rational approach to agents has two major problems:

1. the problem of translating the world into an adequate symbolic description, and
2. the problem of how to represent information about the world in such a way that agents can reason with it in an acceptable fixed time bound.

Brooks, for example, argues that the lack of a time bound makes the approach unfeasible: a rational agent is not able to react appropriately and in real time to the changes in its environment. This criticism has lead to the development of a kind of agent, called hybrid agent, which is both rational and reactive.
2.2.3 Hybrid agents

A hybrid agent is an agent that incorporates the characteristics of two or more different types of agents with the aim to bring together the benefits of both. Frequently, hybrid agents are based on the combination of reactive agents, which are capable of reacting to events that occur in the environment, and rational agents, which are capable of developing plans and make decisions. Hybrid agents are typically complex since they combine the characteristics of several agents.

Kowalski & Sadri employ this kind of agent on their proposed unified architecture that combines rationality with reactivity. It uses definitions for "rational" reduction of goals to sub-goals and integrity constraints for reactive, conditions-action rule behavior. It also uses a proof procedure that combines forward and backward reasoning. Backward reasoning is used primarily for planning, problem solving and other deliberative activities. Forward reasoning is used primarily for reactivity to the environment, possibly including other agents. The proof procedure is executed within an observe-think-act cycle that allows the agent to be alert to the environment and to react to it as well as to think and to devise plans. Both the proof procedure and the agent architecture can deal with temporal information. Furthermore, they also allow the proof procedure to be interrupted (by making it resource-bounded) in order to assimilate observations from the environment and performing actions.

Kowalski & Sadri [11] outline an abstract procedure that defines the observation-think-action cycle for hybrid agents. They express the cycle at the top-most level as follows.

To cycle at any time $t$,

(i) observe any input at time $t$,
(ii) record any such input,
(iii) check the inputs for satisfaction of integrity constraints by reasoning forwards from the input,
(iv) solve goals by constructing a plan, using step (iii) and (iv) a total of $r$ units of time,
(v) select a plan from among the alternatives, and select from the plan an atomic action which can be executed at time $t+r+2$,
(vi) execute the selected action at time $t+r+2$ and record the result,
(vii) cycle at time $t+r+3$.

The cycle starts at time $t$ by observing and recording any input from the environment (step (i) and (ii)). Time $t$ is the clock of the agent. Step (i) and (ii) are assumed to take one time unit each. Step (iii) and (iv) together are assumed to take $r$ units of time. The amount that an agent can spend on "thinking" in step (iii) and (iv) is bounded by $r$. Note that only after having generated a complete plan (step (iv)), the agent begins to execute it (step (vi)). Step (v) and (vi) together are assumed to take one unit of time.

The RR-agent described in this thesis is a hybrid agent.
3 RR-agent Language

This section introduces the language of RR-agents. Typically, an agent can hold positive and negative information. An agent, for example, looking at the sky, can observe that it is snowing. Then, after a while, when weather conditions change, the agent observes that it is not snowing anymore. Therefore, it has to update its own knowledge with respect to the new incoming information. This implies that the language of an agent should be expressive enough to represent both positive and negative information. Knowledge updating refers not only to facts, as above, but also rules which can also be overridden by newer rules which negate previous conclusions.

In order to represent negative information in logic programs, we need more general logic programs that allow default negation not \( A \) not only in premises of their clauses but also in their heads. We call such programs generalized logic programs. It is convenient to syntactically represent generalized logic programs as propositional Horn theories. In particular, we represent default negation not \( A \) (which is the opposite of \( A \)) as a standard propositional variable.

**Definition 1 (Atom).** Propositional variables whose names do not begin with ”not” and do not contain the symbols ”:” and ”÷” are called **objective atoms**. Propositional variables of the form not \( A \) are called **default atoms**. Objective atoms and default atoms are generically called **atoms**.

**Definition 2 (Project).** Propositional variables of the form \( α:C \) (where \( C \) is defined in def. 10) are called **projects**.

A project can be a query or an update.

\( α:C \) denotes the intention (of some agent \( β \)) of proposing the updating of the theory of agent \( α \) with \( C \). Projects can be negated. A negated project of the form not \( α:C \) denotes the intention of the agent of not proposing the updating of the theory of agent \( α \) with \( C \).

**Definition 3 (Update).** Propositional variables of the form \( β÷C \) are called **updates**.

An update is used to expand the knowledgebase of the agent. It can be either a fact or a new rule. \( β÷C \) denotes an update that has been proposed by \( β \) of the current theory (of some agent \( α \)) with \( C \). Updates can be negated. A negated update of the form not \( β÷C \) in the theory of an agent \( α \) indicates that agent \( β \) does not have intention to update the theory of agent \( α \) with \( C \). Atoms, updates and negated updates are generically called **literals**.

**Definition 4.** Let \( K \) be a set of propositional variables consisting of objective atoms and projects such that false \( \not\in K \). The propositional language \( L_K \) generated by \( K \) is the language, which consists of the following set of propositional variables:

\[
L_K = K \cup \{ \text{false} \} \cup \{ \text{not } A \mid \text{for every objective atom } A \in K \} \\
\cup \{ \text{not } α:C, α÷C, \text{not } α÷C \mid \text{for every project } α:C \in K \}.
\]
**Definition 5 (Generalized rule).** A *generalized rule* in the language \( L_K \) is a rule of the form 
\[
L_0 \leftarrow L_1 \land \ldots \land L_n \quad (n \geq 0),
\]
where \( L_0 \) (\( L_0 \neq \text{false} \)) is an atom and every \( L_i \) \((1 \leq i \leq n)\) is a literal from \( L_K \).

Note that, according to the above definitions, only objective atoms and default atoms can occur in the head of generalized rules.

**Definition 6 (Integrity constraints).** An *integrity constraint* in the language \( L_K \) is a rule of the form
\[
\text{false} \leftarrow L_1 \land \ldots \land L_n \land Z_1 \land \ldots \land Z_m \quad (n \geq 0, \ m \geq 0),
\]
where every \( L_i \) \((1 \leq i \leq n)\) is a literal, and every \( Z_j \) \((1 \leq j \leq m)\) is a project or a negated project from \( L_K \).

Integrity constraints are rules that enforce some condition over the state, and therefore always take the form of denials. Note that generalized rules are distinct from integrity constraints and should not be reduced to them. In fact, in generalized rules it is of crucial importance which atom occurs in the head.

**Definition 7 (Generalized logic program).** A *generalized logic program* \( P \) over the language \( L_K \) is a set of generalized rules and integrity constraints in the language \( L_K \).

**Example 1.** Let \( Q \) be the following generalized logic program underlying the theory of Maria.

\[
\begin{align*}
\text{city} & \leftarrow \text{not mountain} \land \text{not beach} \land \text{not travel} \\
\text{work} & \\
\text{vacation} & \leftarrow \text{not work} \\
\text{mountain} & \leftarrow \text{not city} \land \text{not beach} \land \text{not travel} \land \text{money} \\
\text{beach} & \leftarrow \text{not city} \land \text{not mountain} \land \text{not travel} \land \text{money} \\
\text{travel} & \leftarrow \text{not city} \land \text{not mountain} \land \text{not beach} \land \text{money}
\end{align*}
\]

As \( Q \) has a unique intended model\(^1\) \( \{\text{city, work}\} \), Maria decides to live in the city. Things change if we add the rule \( \text{money} \) (indicating that Maria possesses money) to \( Q \). In that case, \( Q \) has four different intended models: \( \{\text{city, money, work}\} \), \( \{\text{mountain, money, work}\} \), \( \{\text{beach, money, work}\} \), and \( \{\text{travel, money, work}\} \). Thus, Maria is now unable to decide where to live.

**Definition 8 (Query).** A *query* \( Q \) in the language \( L \) takes the form \( \text{?-} \ L_1 \land \ldots \land L_n \quad (n \geq 0) \), where every \( L_i \) \((1 \leq i \leq n)\) is a literal from \( L_K \).

**Definition 9 (Active rule).** An *active rule* in the language \( L_K \) is a rule of the form
\[
L_1 \land \ldots \land L_n \Rightarrow Z \quad (n \geq 0),
\]
where every \( L_i \) \((1 \leq i \leq n)\) is a literal, and \( Z \) is a project or a negated project from \( L_K \).

Active rules are rules that can modify the current state, to produce a new state, when triggered. If the body \( L_1 \land \ldots \land L_n \) of the active rule is satisfied, then the project \( Z \) can be selected and executed. The head of an active rule must be a project that is either internal or

---

\(^1\) A model of a logic program \( P \) is a set of all the atoms logically implied by \( P \).
An internal project operates on the state of the agent itself, 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 operate on the state of 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.

Example 2. Suppose that the underlying theory of an agent called Maria contains the following active rules:

\[
R = \{\text{money} \Rightarrow \text{Maria : not work} \}
\]

The head of the first two active rules is an internal project to Maria. The first rule states that if Maria has money, she wants to update her own theory with not work. The head of the last rule is an external project: if Maria wants to travel to the mountains, she proposes to go to the mountains to Pedro.

Definition 10. We assume that for every project \( \alpha : C \) in \( K \), \( C \) is either a generalized rule, an active rule an integrity constraints or a query. Thus, a project can only take one of the following forms:

\[
\begin{align*}
\alpha : & (L_0 \leftarrow L_1 \wedge \ldots \wedge L_n) \\
\alpha : & (L_1 \wedge \ldots \wedge L_n \Rightarrow Z) \\
\alpha : & (false \leftarrow L_1 \wedge \ldots \wedge L_n \wedge Z_1 \wedge \ldots \wedge Z_m) \\
\alpha : & (?-L_1 \wedge \ldots \wedge L_n)
\end{align*}
\]

Note that projects and negated projects can only occur in the head of active rules and in the body of integrity constraints. For example, the integrity constraint \( false \leftarrow A \wedge \beta : B \) in the theory of an agent \( \alpha \) prevents it to perform a project \( \beta : B \) when \( A \) holds. The active rule \( A \wedge \neg \beta : B \Rightarrow \beta : B \) in the theory of an agent instruct it to perform the project \( \beta : B \) if \( A \) holds and agent \( \beta \) has not wanted to update the theory of \( \alpha \) with \( B \).

Example 3. Suppose that the underlying theory of Maria consists of a generalized logic program \( Q \cup \{\text{money}\} \) of Example 1 and the set of active rules \( R \) of Example 2. Then, the theory of Maria has 4 intended models:

\[
M_1 = \{\text{city, money, work, Maria : not work}\}
\]
\[
M_2 = \{\text{mountain, money, work, Maria : not work}\}
\]
\[
M_3 = \{\text{beach, money, work, Maria : not work}\}
\]
\[
M_4 = \{\text{travel, money, work, Maria : not work}\}
\]
Since “Maria: not work” holds in all models, Maria executes the intended project “Maria: not work”.
4 An architecture for RR agents

This chapter describes an architecture for RR-agents. This architecture can be divided into two major parts, one part is implemented in XSB Prolog system [12] and the other implemented in Java [13]. Here we mainly focus on the latter.

We start by presenting an overview of the agent architecture and then we explain in more details its components.

Appendix A specifies the variables and the methods used in the implementation for each component.

4.1 Overview

The architecture of RR-agents consists of six different components. Each component has its own specific task and is implemented in Java to enhance flexibility. You can also say that a RR-agent consists of several subagents, where each subagent is a component, with a specific task to perform. The components are connected to each other as illustrated in Figure 3. Each component is implemented via a Java thread. Each Java-thread runs its own process in the Java-engine and allows the agent to execute code in several places at the same time. This means that the behavior of the agent is not sequential like in the Kowalski and Sadri’s agent cycle mentioned in Section 2.2.3. Rather, the agent has the ability to execute several tasks in our implementation at the same time. Thereby it can exhibit both rational and reactive behavior concurrently. Each component of the agent is controlled by its own Java-thread, explained in Appendix C.

The architecture consists of the following components:

- The Centralcontrol (abbreviated CC) controls the behavior of the entire architecture via a number of control parameters. For example, it determines when a query or update should be sent to the reactive and/or the rational processes.
- The Reactive and the Rational processes (abbreviated ReP and RaP) are the brain of the agent. The Reactive process directive simulates the reactive behavior of the agent for quick answers and the Rational process simulates the rational behavior. Each of the two components consists of one XSB process. The knowledge base is identical in both processes.
- The Updatehandler (abbreviated UH) sorts the information sent from the reactive XSB process and from the surrounding environment.
- The Actionhandler (abbreviated AH) manages the different task (actions) the agent wants to execute. It has the ability to physically effect the environment.
- The External interface (abbreviated EI) main task is to handle communication between the agent and the environment (including other agents).

Together all these components combine the agents base structure. All components except the Reactive and Rational processes are implemented in Java. They consist of a main class and some help classes.
The solid arrows between the components in Figure 3 illustrate all the possible paths of the projects. The dashed line indicates that the Actionhandler can change some parameters in Centralcontrol. The three lines from the Actionhandler indicate that it can perform several things to effect the environment, depending on the situation and task to execute. The knowledgebase of the two XSB processes is preprocessed and loaded from the files whose name can be specified in the preference file of the agent, called prefs.txt.

![Figure 3 Agent base structure](image)

4.1.1 Queue

Between every two components connected by a solid arrow there exists a queue from the class Queue. This is necessary for handling synchronization between the threads of the connected components when exchanging information. The queues are not shown in Figure 3. If a component gets a large workload, the queue collects the information to be exchanged and waits until the receiving component has the time to handle the new incoming information.

The queues themselves do not belong to any specific component, since they work as gates between the different components. That means the same queue has two different names, one name from the component that is inserting new data in the queue and one from the component that is retrieving data from the queue. When referring to a queue in the thesis the component’s name is always present before the queue’s name. For example **UH.toCC** implies the queue **toCC** in the component **UH** (Updatehandler).

4.1.2 ProLogCommand

When the queries and updates for the Reactive and the Rational processes are in the Java part of the agent they are encapsulated into a Java object called **ProLogCommand**. This facilitates the handling of the data in the Java part of the agent. The ProLogCommand-class contains
either a query or one/several updates, and a reference to the previous ProLogCommand-object. The ProLogCommand-object contains either one query or one to several updates. The reference to the previous ProLogCommand is used to know what the XSB process previously has answered to.

Some projects coming from the reactive XSB process have a higher priority than the others. These are called interrupts. We allow them to use a special queue to get executed directly in the XSB processes. Interrupts will be discussed more in detail in Section 4.2.

4.1.3 Project flows in RR-agents

This section illustrates the path of projects in RR-agents. This is done from the perspective of an agent $\alpha$, illustrated in Figure 4.

Suppose that an agent $\beta$ asks $\alpha$ to prove a query $G$. To do so, $\beta$ executes the external project “$\alpha : {?-G}$” to $\alpha$.

![Figure 4 Project flow through an agent $\alpha$](image)

At step 1 $\alpha$ receives the update “$\beta# {?-G}$” in the External interface which in turn sends it to the Updatehandler (step 2). Since the update contains a query requested by another agent to be proved, the Updatehandler sends “$\beta# {-?-G}$” to Centralcontrol (step 3). Centralcontrol sends the update to the Reactive (step 4a) and the Rational (step 4b) XSB processes as regular updates. This means that their knowledge bases are updated with “$\beta# {-?-G}$” indicating that $\beta$ has requested to prove $G$. If there exists an active rule of the form

$$(\beta# {-?-G}) \land Conditions \Rightarrow \alpha : {?-G}$$

whose body is true, then the project “$\alpha : {?-G}$” internal to $\alpha$ is executed. The effect of this project is to update the knowledge bases of the two XSB processes with “$\alpha# {-?-G}$” and since
the request to prove $G$ comes from $\alpha$, Centralcontrol launches the goal $G$ to RaP to be proved at the next cycle.

The project cannot be sent directly to the Rational process since there is no solid arrow from the Reactive process to the Rational process. This means the project has to pass through the Updatehandler (step 6) and the Centralcontrol (step 7) before it can reach the Rational process. Before the Rational process can prove the query both processes are updated with the request from $\alpha$ of proving $G$, “$\alpha\# ?-G$” (step 8a and 8b). CC launches the goal $G$ to the Rational process (step 9). When CC receives back the answer ANS of $G$ from Rap, it updates both the knowledge bases with “$\alpha\# ans(G,ANS)$” (step 10a and 10b). Now the Reactive process again checks if any active rule is true (step 11). If for instance there exists an active rule like

$$(\beta\# ?-G) \land \alpha\# ans(G,ANS) \Rightarrow \beta : ans(G,ANS)$$

the answer $G$ is sent back to agent $\beta$ through the Updatehandler (step 12) and then through the External interface (step 13).

### 4.2 Centralcontrol

The Centralcontrol component consists of the main class Centralcontrol and the class XSB_obj. The second class is necessary to handle communication between CC and the reactive and the rational processes. A third class, Controlframe, is also connected to CC. It contains a graphical interface that allows the user to change some of the variables in CC (this is useful during the testing phase of the agent implementation).

It is from here the agent starts up. CC controls all other components by means if its control parameters. Every component must check whether it is allowed to perform an action depending on the values of the control parameters in CC. One example is the parameter (called waitRe) controlling the time that the Rational process must wait before taking any action when an interrupt is issued.

![Figure 5 Overview of Centralcontrol](image-url)
There are two priority levels of data divided into two types of data being sent to the XSB processes. CC collects the data from the Updatehandler via two queues, one with normal projects (called \texttt{UH.toCC} queue) and one queue with the interrupt projects (called \texttt{UH.toCCinterrupt} queue). The interrupt queue has a higher priority than the normal queue, making the interrupt projects the first to be collected and executed. When all interrupts have been executed CC continues to the queue with the normal projects. In this queue CC only launches only one goal before it checks if any new interrupts have come to the high priority queue.

The interrupts have the ability to break the execution of the current query in the XSB processes and to execute the ProLog command associated with the interrupt. Then the XSB process may relaunch the suspended query. This is a good feature, especially if the rational XSB process has entered a loop with no end, or in some other way takes too long time (the agent decides) to execute the current query.

1. Check the interrupts queue from the Updatehandler component, i.e. the \texttt{UH.toCCinterrupts} queue, and launch all the interrupts in the queue, one by one.
2. Check the queue with normal updates from the Updatehandler component, i.e. the \texttt{UH.toCC} queue, and execute the first ProLogCommand-object before continuing.
3. The thread is paused, i.e. sleeps, for a predefined number of seconds specified by the variable \texttt{delay} in CC.
4. Change the values of the variables in CC if needed.
5. Go to step 1.

CC works in a cyclical way. The time for each cycle depends on the variable \texttt{delay} and on how many interrupts the queue from Updatehandler contains. See above.

The class \texttt{XSB\_obj} implements the interface between CC and a XSB process. Therefore two instances of this class are needed to connect CC to the Rational and the Reactive processes. \texttt{Centralcontrol} then calls methods in \texttt{XSB\_obj} to be able to send ProLog commands to the two processes.

The method \texttt{sendInterrupt} in \texttt{XSB\_obj}-class interrupts the execution of the current goal in the corresponding XSB processes and launch a new goal or execute a new update in the knowledge database of the process. After the interrupt, CC can then relaunch the interrupted goal from the beginning if needed. An example with an interrupt is available in chapter 5.4.

\texttt{Centralcontrol} contains several important parameters needed to control the other components. (They are explained in Appendix A).
4.3 Reactive process

This is one of two parts where the agent's knowledge lies. It is based on the XSB Prolog system and uses the notion of projects to specify which actions the agent must do to react to the changes on the environment.

![Figure 6 Overview of XSB processes](image)

There are two XSB processes necessary for the agent's reasoning skills. These two processes actively work for the agent's capacity to reason. The two XSB processes, called Reactive XSB process (ReP) and Rational XSB process (RaP), implement the reactive and rational behavior of the agent. Each XSB process has its own database. When CC execute an update, CC updates the databases of ReP and RaP at the same time. The databases of the two XSB processes formalize the knowledge of the agent that is needed to characterize the reactive and rational behavior. The database is written in ProLog. There is a window in Java connected to each of the two XSB processes, making the user able to debug and to send in extra commands to the process.

To be able to send information between Java and XSB processes one has to use an interface of some kind. We use the InterProlog [14] interface from the Portuguese company Declarativa. There are two variants of this interface, one that uses TCP/IP sockets to communicate and one that uses the Java Native Engine, JNE. We use JNE since it's faster and more stable then the other one. For more information about InterProlog see Appendix D.

The task of ReP is to quickly react to the environment's changes by using active rules. The reactive process does not have to formalize a working plan but merely respond quickly to external and internal events. It also has the control over what the Rational process should reason about when requested from another agent.

There are two priority types for a project. The projects can contain any information that is supposed to be sent as an internal or external project. The projects with a higher priority include only internal information and are called interrupts. They are only sent from ReP. The interrupt has the ability to break an execution of a query, execute a new query, and when finished continue the interrupted query if needed. An interrupt sent to the Rational counterpart
can contain a new update or a query to be proved or it can just be a way to stop the execution of a never-ending query in the Rational process.

The ReP is connected to the Updatehandler through the queue **CC.toUH**.

### 4.4 Rational process

When the agent $\alpha$ receives a request to prove a goal $G$ from another agent $\beta$ (via the update “$\beta \# \text{?-} G$” in the theory of $\alpha$), $G$ is not proven directly. Instead, the agent $\alpha$ has the ability to decide whether or not to prove $G$. In fact, $\alpha$ proves a goal $G$ in the Rational process only if $\alpha$ itself requests that via an internal project “$\alpha: \text{?-} G$”.

RaP is connected to ReP through CC as mentioned earlier, through the queue **CC.fromRatoRe**. The Reactive process on the other hand is indirectly connected to the next part of the agent, the Updatehandler, through the queue **CC.toUH** in CC.

### 4.5 Updatehandler

The Updatehandler collects all the output from ReP and the output from EI. The components task is to send the queries and updates to the right destination. Since a ProLogCommand-object can contain several updates this means there can be several destinations for every ProLogCommand-object entering the UH. The queries and updates (incapsulated in a ProLogCommand-object) are distributed to the right output queue in a new ProLogCommand-object depending of type of the data and destination. For now only updates that have the same destination are all collected in the same ProLogCommand-object and sent to its destination. Queries are sent one by one in new ProLogCommand-objects to their destinations.

1. Check with **Centralcontrol** if it is ok to fetch new data from the queue **fromXSBandEXT** that is connected to the reactive XSB-process. This is possible by checking the value of the control parameter **CC.Uhokreceive**. If it is not ok then it waits until it is ok.
2. A ProLogCommand-object is fetched from the queue. If the queue is empty, then it sleeps (i.e. wait) until a new ProLogCommand-object arrives in the queue.
3. The data (from the ProLogCommand-object) is distributed to the right output queue in a new ProLogCommand-object depending on the type of the data. If several updates have the same destination they are all collected in the same ProLogCommand-object.

A project can be (i) an external project that has to be sent to another agent, (ii) an external project sent to the agent by another agent to be processed, (iii) an internal update or query that has to be sent to the XSB processes again or (iv) an action to be performed. The internal cycle of this component is smaller then the one in CC and it does not need any other kind of Java-classes to help execute its task. UH consists mainly of the sorting algorithm between the queues that collects ProLogCommand-objects from ReP and the output queues and redistribute them to the right destination. There are four different output queues. One queue
sends normal updates and queries to the CC component, another sends interrupt updates to the CC component, one sends the external projects to the External interface and the last queue is connected to the Actionhandler.

Projects marked with "stateModify" and "doAct" are sent to the queue that is connected with Actionhandler. If a project should be sent to another agent it ends up in the queue to the External interface. Updates and queries for internal use are usually sent to UH. If it contains a special predicate it is sent to either the UH or the Actionhandler. Interrupt projects, marked with “sendInterrupt”, always ends up in the high priority queue. The syntax of “sendInterrupt” has two arguments, on for the new goal to be launched (or the list of urgent updates) and one flag for allowing the previous goal to be relaunched when the interrupt is finished. For example, “sendInterrupt(α: (?-X), true)” breaks the proving of the current goal, launches the new goal (X). When the proving of the goal X is finished, it relaunch the proving of the previously goal.

4.6 Actionhandler

The agent can manipulate its environment in two ways: either by communicating with other agents using the External interface or using the Actionhandler's ability to directly change its environment. The Actionhandler (abbreviated as AH) is implemented by the class Actionhandler and receives its projects from UH.

Into AH two types of commands are sent. It is the “doAct” command and the “stateModify” command.

The “doAct” command tells the agent to physically change its environment. When AH task is to change its environment, it mostly uses a specialized subagent to complete the task. The command “doAct” uses only one argument, “doAct(ARGUMENT)”, where the argument is an action for the agent to perform.

AH has also the capability of changing the internal variables of the agent via the use of the dedicated command “stateModify”. For example, “stateModify(delay,5000)” assigns the value 5000 to the control parameter delay in CC.
4.7 External interface

The last component of the agent is the external interface between the agent and its environment. The External interface (abbreviated as EI) uses its own Java-thread to compute its tasks making it rather independent from the rest of the agent. External projects can come from other agents, the surrounding environment or from a monitor window where a user can provide inputs. The queue $EI.fromUH$ (or $UH.toEXT$ as it is called in $UH$) connects UH with EI allowing it to send the projects to external receivers.

All data received from external source, wherever its origin is, is collected in $EI.toCCfromWindow$, which is the input queue to UH from EI. This is the same queue that is called $fromXSBandEXT$ in UH.

This component is not thoroughly described here since it is treated in another thesis that describes the communication between several agents. At the moment this component only consists of a window for input/output.
5 RR-agent examples

The next chapter illustrates the use of generalized rules to change the agent’s current state.

5.1 Example 1

This example taken from [15] illustrates John watching television. While the television is on John stays watching the television, but as soon as the television is turned off he goes to sleep.

State 1

\[
\begin{align*}
\text{sleep} & \leftarrow \text{not watch\_tv} \\
\text{watch\_tv} & \leftarrow \text{tv\_on} \\
\text{tv\_on} & \leftarrow \text{not watch\_tv} \\
\end{align*}
\]

John is controlled by ordinary rules and from the beginning the television is on. This implies that John is watching television. When the update power\_failure is issued the television is not on and John goes to sleep.

State 2

\[
\begin{align*}
\text{not tv\_on} & \leftarrow \text{power\_failure} \\
\text{power\_failure} & \leftarrow \text{not tv\_on} \\
\end{align*}
\]

If not power\_failure holds, then the television is on and John does not sleep since he is watching television.

5.2 Example 2

This example extends Example 1 to add a reactive behavior to John. Suppose the case where John goes to bed if it is late and the television is off.

State 1

\[
\begin{align*}
\text{sleep} & \leftarrow \text{not watch\_tv} \\
\text{watch\_tv} & \leftarrow \text{tv\_on} \\
\text{tv\_on} & \leftarrow \text{not watch\_tv} \\
\text{late} & \\
\end{align*}
\]

sleep, late \Rightarrow \text{John: doAct( go\_to\_bed)}
John changes his state when the power_failure is issued. This time an action is triggered by the active rule “sleep, late ⇒ John: doAct(go_to_bed)” which says to John to go to bed if he wants to sleep and it is late.

5.3 Example 3

This example illustrates the use of the agent’s time dependant active rules. Consider a situation where a basic decoder controls which channels John can watch on a television. The decoder contains two channels, tv8 and tv1000. When John wants to watch television he turns the television on and can watch the channels he has access to. TV1000 is a special case. Normally everybody can watch it, but between 0 o’clock and 2 o’clock the channel sends adult entertainment. Only persons with a special code can watch this.

% one can watch channel X on the television if the channel is accessible
% and the tv is on

watch(X) ← access(X), tv_on

% tv8 is always accessible

access(tv8)

% rules for when it is adult tv or not on channels that have this between 0 and 2 o’clock
% the built-in predicate sys(X) executes X in the XSB environment and not in the agent
% environment, like the query “?-X”

sys(time(H, M, S)), sys(H < 3) ⇒ decoder: adult_tv
sys(time(H, M, S)), sys(H > 2) ⇒ decoder: not adult_tv

% rules for watching tv1000

not adult_tv ⇒ decoder: access(tv1000)

% this rule is for turning the access on during the adult entertainment
% B is a variable that stands for any person

B# open(tv1000, CODE), code(tv1000, CODE), adult_tv ⇒ decoder: access(tv1000)

sys(time(H, M, S)), eq(H, 0), eq(M, 0), eq(S, 0) ⇒ decoder: not access(tv1000)
sys(time(H, M, S)), eq(H, 2), eq(M, 0), eq(S, 0) ⇒ decoder: access(tv1000)
John watches television. He is watching tv1000 and the clock passes midnight. At the same time the decoder codes the tv1000 channel. John switches over to tv8 and sees that the channel works. He changes back to tv1000 and it is still no access to this channel. He inserts the code for tv1000 using the command open(tv1000, 123654). In the decoder the update looks like “John# open(tv1000, 123654)” since it came from John. Now the access to tv1000 is granted and he can continue to watch the channel. If now John wants to stop watch all he have to do is to turn off the television and the decoder removes the access to tv1000 until 2 o’clock when the normal programs starts again. If he does not want to shut off the television he can instead send the close command. That looks like “John# close(tv1000)” to remove the access to channel tv1000.

5.4 Example 4

This example shows how the current activity of an agent can be interrupted. Suppose a situation where an agent $\alpha$ is performing some rational activity, here for simplicity we model an agent that count for one minute.
6 Comparison

Comparing the RR-agent with DALI agent architecture shows the difference between the two agents and thereby also showing their different way of implementing reactive and rational behavior.

6.1 DALI

The DALI agent system is a similar project in Italy by Stefania Constatini[16],[17]. It consists of a Java-Prolog agent-based system. The DALI agent has the ability to reactively respond to events and does also have rational features implemented with its reasoning of present events. The last means that if the agent is aware of events but has not reacted to them, it allows the agent to reason about the event. As soon as the reactive rule is applied the event is not longer treated as a present event.

6.1.1 The DALI system

The DALI system consists of three parts: one server, several clients and each client can manage several DALI modules (i.e. programs/agents the user wants to run). The DALI modules are connected to a client, which itself is connected to the server. The DALI server is a simple server that returns the address of the server in such a way the agents can connect to it.

The DALI language does not commit to any specific agent architecture and does not commit to any planning formalism. It is a logic programming language that includes some predefined features.

The agent is controlled by reactive rules. The reactive rules act on states the agent can achieve when different events occur. The reactive rule implies most of the time that a action should be performed. This is done by an action atom.

There are also four types of events atoms in the DALI language and then there is the reactive rule that makes the agent do an action when different states are achieved. The predefined events of the agent language are:

- **external events** are either messages from other agents or changes in the surrounding environment;
- **internal events** are internal actions that are executed repeatedly at a predefined time interval. It is implemented to make the agent proactive and execute actions by itself;
- **past events** records previously actions, events and states the agent has executed;
- **present events** are events the agent is aware of, but it has not yet reacted to. As soon as a reaction to an event occurs, the event is not perceived as present event any more;

The DALI system uses, as the RR-agent system does, preprocessing to load events and information to knowledge base when the agent starts. New events, conditions and actions for the agent are supposed to be learned while the agent is running. These kinds of updates can only come from the user.
6.1.2 DALI language

When comparing what the DALI and the RR-agent language system can accomplish, you notice that the two systems are similar. In this section the syntax of DALI is explained in more detail and translated into the RR-agent syntax. The DALI language is more complex then the language for RR-agents. For example, an RR-agent does not have different types of events. They are all treated as updates. We assume that the name of the RR-agent is $\alpha$ in all comparisons.

**Reactive rules**

Reactive rules are the base for a DALI agent to respond and change the environment. It consists of conditions and actions to perform under certain conditions.

**Syntax:**
- $a$(Action):-Conditions

The reactive rule can be divided into two parts; the conditions (i.e. the head and otherwise called *action rules*) and the actions (i.e. the body). The agents based on the DALI language use of the *reactive rules* are the same way as the RR-agents use *active rules*.

*Action rules*, or just *Conditions*, specify the conditions for performing an action. The body of the reactive rule can consist of any mixture of actions and other subgoals, since a reaction to an event can involve rational activity.

*Actions* are intended to somehow affect the environment. Current DALI agents are *infobots*, i.e., software agents that act in a purely computational environment. Therefore, the only actions possible are sending messages to other agents and printing a text in standard output. In case DALI should be used for robotic applications, a coupling with physical actuators and sensors would be required. *Actions* with no action rules are executed unconditionally and, as a subgoal, always succeed. For now there are two types of *Actions*, one for sending messages to other agent and one for printing.

**Syntax:**
- $a$(Action): prints *Action* on standard output.
- $a$(message(To,Content)): sends the message *Content* to the agent *To*.
- $a$(message(all,Content)): sends the message *Content* to all the agent that are currently active.

**External Events**

The reactive capabilities of an agent are determined by its way of copying with external events. An external event can be:
- a message coming from another agent.
- some change in the state of the external world, of which the agent becomes aware.

**Syntax:**
• **eve(External_Event):-Reaction**: where *eve* is a distinguished predicate whose argument is an external event. *Reaction* is a conjunction composed of standard Sicstus Prolog subgoals, and/or DALI actions.

### Internal Events

Internal events implement the proactive capabilities of DALI agents. Each *Internal_Event* is a goal (occurring in the head of at least one rule), which is specially designated as a conclusion to which the agent may "react" in some way, i.e. a conclusion that triggers further inference.

**Syntax:**

- **Internal_Event:-Conditions**
- **evi(Internal_Event):-Reaction**: where *Internal_Event* is a conclusion, reached by the agent whenever the *Conditions* are true.

The reactive rule is similar to that for the external events (see above), except that it is indicated by the distinguished predicate *evi*. The interpreter automatically attempts each *Internal_Event*, from time to time, to check whether the *Conditions* are true, and thus a reaction should occur. As default, each internal event is tried as long as the agent is alive, every three seconds. You can customize the treatment of internal events, by telling the interpreter when and how often each internal event should be tried. To this aim, you can add directives to initialization file `.plf` (same name as the program file, extension `.plf`). This can be done according to the following syntax: `evento_interno(Event,Frequency,Option)`. where *Option* can be:

- **forever**: the event will be attempted as long as the agent will stay alive.
- **until_cond(Condition)**: as soon as *Condition* is verified, the event is not attempted any more, i.e., in a sense it "expires".
- **until_date(from(YY,MM,DD,HH,mm,ss),to(YY,MM,DD,HH,mm,ss))**: the event is attempted at the specified frequency starting at time *from* and ending at time *to*.

### Past Events

Past events constitute the memory of the agent, which is capable of recording (some of) the events and actions:

- External and internal events to which the agents has reacted
- Actions previously performed by the agent.

**Syntax:**

- **evp(Goal)**

As a default, each past event is kept for 20 seconds, and then removed from the memory of the agent. You can customize this aspect by adding directives to the `.plf` file: `evento_passato(Event,Option)`. where *Option* can be:

- **forever**: the event is never removed from the agent's memory.
- **until(Condition)**: as soon as *Condition* is verified, the event is removed.
• **How_long**: the event is kept for **How_long** seconds

**Present Events**

Present events are events of which the agent is aware, but to which it has not reacted yet (i.e., the reactive rule has not been applied). They occur in the body of rules, and allow the agent to reason about events, thus going beyond pure reactivity. As soon as a reaction to an external event occurs, this external event is not perceived as a present event any more.

**Syntax:**

- In the following situation, an internal event depends upon a present event:

 Internal Event 1:-en(External Event 2)
 evi(Internal Event 1):-Reaction1
 eve(External Event 2):-Reaction2

If the internal event **Internal Event 1** is attempted before reacting to the external event **External Event 2**, then the corresponding present event is available. Thus the agent performs both **Reaction1** and **Reaction2**. If instead **External Event 2** is reacted before attempting **Internal Event 1**, then the present event is no more available, **Internal Event 1** cannot be concluded, and **Reaction1** does not take place.

6.1.3 **DALI language vs. RR-agent language**

The language of DALI and the language of the RR-agent are very similar. Since both are ProLog languages they both uses the Horn clause as the basic structure. Another example is what the DALI language refers to as reactive rules are equivalent to the RR-agent language’s active rules. One big difference is that the DALI language copes with several kinds of events that are represented as special atoms called event atoms. There are also action atoms that imply the agent should perform a task. In the RR-agent language this distinction is not made. The RR-agent atoms are instead called objective or default atoms, see section 3. An event is just a state and an action is just an update. The RR-agent distinguishes instead between internal projects and external projects.

The execution model of DALI for external events and reactive rules is simulated in RR-agents via updates and active rules. In fact, when an RR-agent \( \alpha \) receives an external event \( ext_E \) from agent \( \beta \) (possibly, the environment), the agent \( \alpha \) updates its own theory with the update \( \beta \div (ext_E) \). External events in DALI language are treated as external projects in the RR-agent’s language. This new update may trigger a reaction if the theory of \( \alpha \) includes an active rule whose body is true.

A DALI reactive rule of the form:

\[
eve(ext_E): \neg G, \ a(\ldots)
\]

where \( G \) is a subgoal to be proven and \( a(\ldots) \) a DALI action, can be simulated via the following active rules in a RR-agent:
\[ \beta \div (\text{ext}_E) \Rightarrow \alpha : (\neg G) \]
\[ \beta \div (\text{ext}_E) \Rightarrow \alpha : a(\ldots) \]

When a reactive rule is true an action is derived from it. Comparing the DALI language Action with the RR-agent language you see that an Action can be formalized as a project. The project could be addressed to another agent (agent \( \alpha \) sends a project like "\( \beta \): new\_condition" to agent \( \beta \)) or it could be sent to itself to perform an action (like "\( \alpha \): doAct(\ldots)" or "\( \alpha \): stateModify(\ldots)") or a new update (i.e. "\( \alpha \): new\_condition").

In the RR-agent a DALI Internal Event can be simulated using an active rule. "\( \text{int}_E \) : \neg \text{Conditions}" equals in the RR-agent as an update like "\( \text{Conditions} \Rightarrow \alpha : \text{int}_E \)". When the RR-agent changes its state and update with \( \text{int}_E \), then the reaction can occur, i.e. "\( \alpha \div (\text{int}_E) \Rightarrow \alpha : \text{Reaction} \)".

A DALI Present Event could be simulated using an active rule in the reactive process that states that goal \( G \) has to be proven in the rational process of the RR-agent.
7 Conclusions and future work

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

Our aim is to have a simple architecture and to test its behavior in different application domains. Currently, we are working on an experimental implementation to test our agent as logic-based controller. The idea is to have an artificial world whose physical laws are modeled in MatLab and to connect it with our agent implementation to control some object of that world.

An interesting line of research is to investigate how to formally describe the behavior of our agent architecture. This will allow us (i) to formally compare our architecture with other hybrid architectures proposed in literature, (ii) to verify its logical properties, and (iii) to analyze and expand its architectural design to accommodate other components. In the future, we plan to add more modules to the basic structure, such as a planner, a learning module, etc., modeling other rational abilities and their functionalities, and to test their interactions with the basic architecture.

The design of our architecture allows an agent to tune its behavior to the environmental needs. Thus, the agent has capabilities of self-monitoring and self-control. Regarding self-evaluation, we are investigating the benefits of integrating a meta-management component into the architecture along the way proposed by Sloman [18]. Our aim is to design an approach to agents where self-evaluation capabilities allow an agent to dynamically reconfigure its architectural components.

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


[14] InterProlog. Available at http://www.declarativa.com


9 Acknowledgements

I would like to thank prof. L.M. Pereira for his fruitful discussions on the agent architecture design. Dr. M. Calejo from Declarativa for making available InterProlog and for his help.

I am grateful to prof. D. Warren for having adapted the XSB system to allow the use of interrupts.

Finally I would like to thank M. Blixt and C. Wilczoch, Mattias for the interesting discussions on my work and the multi-agent interface and protocols, and Christian for his work on reading, discussions and comments on my thesis.
Appendix A / RR-agent’s Java classes

Below the Java-classes are briefly explained with the most important variables and methods. Some variables are shared between different classes with different names.

1 Centralcontrol

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>waitRe</td>
<td>the waiting time before an interrupt should brake the Reactive XSB process, default value is 3000 milliseconds</td>
</tr>
<tr>
<td>waitRa</td>
<td>the waiting time before an interrupt should brake the Rational XSB process, default value is 3000 milliseconds</td>
</tr>
<tr>
<td>ReactiveUpdateOn</td>
<td>a boolean value for turning the time bound check of the rules on and off, default is true</td>
</tr>
<tr>
<td>ReactiveUpdateTime</td>
<td>time between the time bound check of the rules in the XSB process, default value is 10000 milliseconds</td>
</tr>
<tr>
<td>delayForPrint</td>
<td>the waiting time before the XSB process can send information to Java, default value is 250 milliseconds</td>
</tr>
<tr>
<td>prologStartCommand</td>
<td>describes where in the file system the XSB engine is located, default is &quot;D:\exjobb\xsb\Xsb_2_5\config\x86-pc-windows\bin\xsb.exe&quot;</td>
</tr>
<tr>
<td>UHokreceive</td>
<td>checks if UH is allowed to receive ProLogCommand objects from XSB-processes</td>
</tr>
<tr>
<td>ReactiveOutputWin</td>
<td>variable that activates a debugger window for the Reactive XSB process, default is true</td>
</tr>
<tr>
<td>RationalOutputWin</td>
<td>variable that activates a debugger window for the Rational XSB process, default is true</td>
</tr>
<tr>
<td>agentname</td>
<td>name of agent, default is &quot;alpha&quot;</td>
</tr>
<tr>
<td>prefsfile</td>
<td>name of preference file, default is &quot;prefs.txt&quot;</td>
</tr>
<tr>
<td>CF</td>
<td>reference to Controlframe object</td>
</tr>
<tr>
<td>klockan</td>
<td>the clock for managing the timebound checks of the XSB process</td>
</tr>
<tr>
<td>UH</td>
<td>reference to the Updatehandler</td>
</tr>
<tr>
<td>fromTtoR</td>
<td>the queue that collects answers from the Rational process to update both the Rational process and the Reactive process</td>
</tr>
</tbody>
</table>
Reactive reference to the Reactive process
Rational reference to the Rational process
new_Command contains the current ProLogCommand-object sent to the Reactive process and the Rational process

Methods Description

actionPerformed(ActionEvent e) called by the variable klockan, used for proving timebound rules
getAgentName() returns the value in variable agentname as a string
Centralcontroll() class constructor
read_prefs() reads the preferencefile defined by the variable prefsfile and updates the corresponding variables
writePrefs() writes the variables values to the preferencefile defined by the variable prefsfile
fetchinginterupts() fetches the interrupts from the UH.toCInterrupts queue and sends them to the XSB processes
fetchingProvedQueries() updates the XSB processes with the answers from proven queries from the Rational XSB process
fetching() send one ProLogCommand-object to the XSB processes
changevariable() changes the variables in Centralcontrol
UHok() returns true or false depending of the value in the private variable UHokreceive

2 XSB_obj

Variables Description

PLengine object containing the XSB SubprocessEngine
PLcom string containing the output of the XSB process, which is later inserted in the New_PLcommand
interrupted_Command ProLogCommand-object containing the interrupted XSB update if an interrupt is issued, else null
Current_PLcommand ProLogCommand-object containing the current XSB update. If the XSB process is idle, this is null
New_PLcommand: ProLogCommand-object where the output from the ProLogEngine is collected.
goalfinished: boolean that indicates if the XSB process is working or not.
goalinterrupted: boolean that indicates if the XSB process is interrupted or not.
goalinterrupted2: boolean that indicates if the XSB process is interrupted or not.
outputQueue: object linked to the queue connected to the next part in the agent.
processname: String containing the name of the process, Rational or Reactive.
outPutten: boolean value that indicates if the output from the XSB process should be sent to outputQueue.
filename: String containing the filename in the preprocessing edition.
agentname: String containing the agents name.
collectoutput: boolean value that indicates if the output from the XSB process should be sent to outputQueue.
prologStartCommand: String that keeps track of where the engine is located in the os environment.

Methods

Description

XSB_obj(Queue output, String typen, boolean outputwin, String agentname2) throws InterruptedException
constructor

XSB_obj(Queue output, String typen, boolean outputwin) throws InterruptedException
constructor

sendAnswer (): returns the output from the latest update or query in a ProLogCommand object.

print(String s): method for collecting the output from the XSB process and saving it in the variable New_PLcommand.

goalFinished(): returns true if the XSB process is idle, else false.
isGoalinterrupted(): returns true if an interruption is executed, else false.

sendExec(int delay): sends the command exec to the XSB process.

sendToPrologQuery(ProLogCommand new_goal, int delay): sends a query to the XSB process.
setgoalfinished(boolean temp)
    changes the variable that is checked to see if the process is idle

sendToPrologUpdate(ProLogCommand new_goal, int delay)
    sends an update to the XSB process

sendInterrupt(ProLogCommand new_goal, int delay, int waitforinter)
    sends an interrupt to the XSB process

3 Updatehandler

Variables
    Description
activity
    thread, makes this component independent of executions in other
    components
fromXSBandEXT
    the input queue to the component
toCC
    the output queue to Centralcontrol for normal updates
toCCInterupts
    the output queue to Centralcontrol for interrupts
toEXT
    the output queue to External interface
toAH
    the output queue to Actionhandler
CC
    a reference to Centralcontrol
EW
    a reference to the External interface
AH
    a reference to the Actionhandler
Agentnamn
    the agents name

Methods
    Description
Updatehandler(Centralcontrol temp)
    constructor
run()
    collects ProLogCommand-objects from the input queue, reformats
    them and distributes them to the right output queue
splitInQueriesAndUpdates(ProLogCommand in)
    divides the updates in a ProLogCommand by destination and creates a
    new ProLogCommands to each destination
sendInterrupt(ProLogCommand current, String the_command)
    puts a ProLogCommand object in the toCCInterupts queue

4 Actionhandler
Variables Description

the_command String, contains the command sent to Actionhandler
CC a reference to Centralcontroll-object
ToAH the queue containing ProLogCommand-objects from Updatehandler
ToAgent the queue containing ProLogCommand-objects to Updatehandler

Methods Description

run() checks if there exists a ProLogCommand-object in the queue from Updatehandler and executes it, else it waits 500 milliseconds before trying again
executeCommand(ProLogCommand command) divides the different types of commands the Actionhandler can manage to respective method
stateModify(ProLogCommand command) this method changes the value in the corresponding variable in Centralcontroll

5 ProLogCommand

Variables Description

question ProLogCommand-object containing the update that led to the output in this ProLogCommans-object
type i1 the object is an internal update, 2 the object is an internal query, 3 the object is to external destination, 4 the object is from external source
PLC2 string containing the updates and queries

Methods Description

newCommand(String s) inserts the value of s into the PLC2 string
before(ProLogCommand before) inserts a the ProLogCommand-object in the variable before in the variable question
getBefore() returns the ProLogCommand-object in the variable question
getCommand() returns the value of variable PLC2 and then removes the content of PLC2
readCommand() returns the value of variable PLC2

6 Queue

Methods Description
putLast (Object obj) inserts an object last in the queue
getFirst() returns the first object in the queue
Appendix B / Prolog

The beginning of logic programming can be attributed to Kowalski and Colmerauer. Kowalski formulated the procedural interpretation of Horn clause logic and showed that the axiom $A$ if $B$ can be read as a procedure of a recursive programming language, where $A$ is the procedure head and $B$ is its body. At the same time, early 1970's, Colmerauer and his group at the University of Marseille-Aix developed a specialized theorem prover, which they used to implement natural processing systems. The theorem prover, called PROLOG (for Programation et Logique or Programming in Logic), embodied Kowalski's procedural interpretation.

The first Prolog interpreters was not as fast as Lisp systems but this changed in mid 1970's when David H.D. Warren and his colleagues developed an efficient implementation of Prolog. The compiler, which was almost entirely written in Prolog, translated Prolog clauses into instructions of abstract machine that is now known as Warren Abstract Machine (WAM). However, the computer science and artificial intelligence community was still ignorant or, at bests, indifferent to logic programming.

This was changed a bit with announcement of the Japanese Fifth Generation Project, which clearly stated the important role of logic programming in the next generation of computer systems. Not only researchers but also general public started to spell the world PROLOG. From that time Prolog is developing to new heights. Currently, one of the most promising extensions is Constraint Logic Programming (CLP).

Logic programming and one of its representatives, PROLOG, make a declarative approach to writing computer programs. Logic programs can be understood and studied using two abstract concepts: truth and logical deduction. One can ask whether an axiom in a program is true, under some interpretation of the program symbols, or whether a logical statement is a consequence of the program. These questions can be answered independently of any concrete execution mechanism.

On the contrary, Prolog is a programming language with precise operational meaning that borrows its basic concepts from logic programming. Prolog programs are instructions for execution on a computer. These instructions can almost always be read as logical statements and, most important, the result of a computation of a Prolog program is a logical consequence of the axioms in it.

Note that effective Prolog programming requires an understanding of the theory of logic programming.
Appendix C / Java

Java is an object-oriented language and originally an extension to C++ (called C++ ++ --).

Back in the beginning the name of the language was Oak, because of a large oak standing outside the window of the Sun Microsystems programmers. It later had to change name because there existed already a language that had that name. After a lot of brainstorming and a simple visit to the local coffee shop, the new name Java was born. It had to be completely platform independent, and function seamlessly regardless of the type of CPU in the device. For this reason, Java was designed to be an interpreted language, since it would be practically impossible for a compiled version to run on all available platforms. To facilitate the job of the interpreter, Java was to be converted to an intermediate "byte-code" format, which is then passed around across the network and executed/interpreted dynamically.

Java is also a multithreaded language. It provides support for multiple threads of execution (sometimes called lightweight processes) that can handle different tasks at the same time. This means that several instances of a class can execute at the same time and do different tasks allowing multitasking.

One equal important design objective for Java would then have to be high reliability by allowing the least amount of programmer-introduced errors. This was the motivation for several important modifications to C++. The concepts of multiple-inheritance and operator overloading were identified as sources of potential errors, and eliminated in Java. Furthermore, in contrast to C++, Java included implicit garbage collection thereby providing efficient memory utilization and higher reliability. Finally, Java attempted to eliminate all unsafe constructs used in C and C++ by only providing data structures within objects.

Another essential design criterion was security. By design, Java-based devices were to function in a network and often exchange code and information. Inherently, security is of great concern in a networked environment, especially in an environment as network dependent as the conceived Java-based systems. For this reason, pointers were excluded in the design of Java. This would theoretically eliminate the possibility of malicious programs accessing arbitrary addresses in memory.

If Java were to be widely accepted and used within the consumer electronics industry, it would have to be simple and compact, so that the language could be mastered relatively easily, and development would not be excessively complex.

On its voyage to become a multi-platform language, as it is today, it has been used for several areas. It was originally designed in the beginning of the 90's to be used in a programmable remote control to manage several, if not all, home electronics machines. While impressive, the market was not conducive to this type of technology, as later demonstrated by Apple's Newton PDA. Sun then changed the market area for the technology. Now the focus was upon interactive television. After another failure in 1992 with a canceled product launching most of the technology was cancelled and the developers were reassigned. In mid 1994 one of the developers had an excellent idea. The World Wide Web, by nature, had requirements such as reliability, security, and architecture independence, which were fully compatible with Java's
design parameters. A perfect match had been found. By September of 1994 a Java-based web browser which was later renamed "HotJava" saw the daylight. By October 1994, HotJava was stable and demonstrated to Sun executives. This time, Java's potential, in the context of the World Wide Web, was recognized and the project was supported. Although designed with a different objective in mind, Java found a perfect match in the World Wide Web. Many of Java's original design criteria such as platform independence, security, and reliability were directly applicable to the World Wide Web as well. The introduction of Java marked a new era in the history of the web. Information providers were now given the capability to not only deliver raw data, but also the applications that would operate on the data. Sun formally announced Java and HotJava at SunWorld '95.
Appendix D / InterProlog: a declarative Java-Prolog interface

Miguel Calejo
Declarativa
R. Cerca 88, 4150-200 Porto, Portugal
mc@declarativa.com

Abstract
Declarativa's InterProlog is a library allowing the development of combined Java+Prolog applications. Java and Prolog developers can symmetrically and dynamically invoke predicates and procedures with bi-directional object/term conversion based on the underlying combination of Java's built-in data-driven Serialization API with a Prolog Definite Clause Grammar. The result has been used in both academic and commercial scenarios, and in server and desktop environments. The current GPL-licensed version is based on TCP/IP sockets; there is also a proprietary Java Native Interface implementation, offering the same core functionality but with significant performance gains.

Keywords: Java, Prolog, declarative, GUI, integration, grammar, remote procedure calls

Introduction
Logic programmers need a second language; the world simply isn't totally declarative. Unless you're willing to ignore that, or to stick to heavy weight Prolog environments providing all sorts of operational OS/GUI/DB primitives, typically funded out of exotic economic or academic niches, it's obvious a second language is needed to make projects useful and complete. This is due to both technical and economical reasons. Logic programming foundations still shake occasionally for the good, and we should leave the door open for innovation (e.g., enhanced proof procedures, higher-level languages etc); and maintenance and integration costs are a factor advising careful choice of tools. In other words, Prolog environment implementation work is (perhaps) better focused on intrinsic declarative engine issues, delegating other problems elsewhere.

Looking at the other side, (perhaps) the software world does not need yet another specialized logic inference engine implemented on Java, C++, LISP or whatever each time logic gets used in the real world, and might instead reuse nearly three decades of logic programming R&D.

Fortunately Java, arguably the emerging language of choice in many "real world contexts", constitutes an excellent platform for this scenario. Java is multi-platform, object-oriented, dynamic, heavily adopted. InterProlog is an attempt at capitalizing on this opportunity, taking full advantage, among others, of Java's Reflection and Serialization APIs. Reflection allows arbitrary messages to be sent to Java from Prolog, without any specific need for Java programming. Serialization provides a uniform and open mechanism to exchange data between both, by having Java sending and receiving from Prolog serialized byte streams representing Java object state.
InterProlog is a Java front-end and functional enhancement for standard Prologs, running on both Windows and Unix, and currently supporting XSB Prolog (http://xsb.sourceforge.net/). It is implemented as a set of standard Java classes and Prolog predicates, and is available under the terms of the GNU Library License from http://www.declarativa.com/interprolog. It provides Prolog with the ability to call any Java method, and for Java to invoke Prolog goals, by using a communication layer based on standard TCP/IP sockets to pass object/term data among both processes:

Another proprietary version offers the same functionality (except for I/O redirection) but based instead on direct JNI integration.

The InterProlog system was partially and briefly overviewed in [Calejo 98], and mentioned in a previous survey [Calejo 99]; it elaborates on ideas previously explored in a C++ context in [Soares Calejo 97], and in [Calejo Sousa 97]. The evolution of Java as a dynamic language opened the door to the articulation of Java Serialization [Sun 97] with Prolog Definite Clause Grammars:

InterProlog also provides a simple graphical front-end to Prolog using Java's basic subprocess-handling capabilities, by running it under a separate process and redirecting its STDIO to listener objects, such as a Java window.

Object network in memory

Prolog Term

JAVA Serialization API

Stream bytes

InterProlog Object Grammar

object(class(...),...data)