Agents in Logic Programming

Jacinto Alfonso Dávila Quintero

April 1997

Submitted to the University of London as a thesis for the degree of
Doctor of Philosophy

Department of Computing
Imperial College of Science, Technology and Medicine

Abstract

The objective of this thesis is to explore ways of describing agents in logical theories.

The contribution is that the logical theories we build are a generalised form of logic programs. Like normal logic programs, these theories have an intuitive declarative reading and a procedural interpretation to guide the implementation of automatic devices and software. Both human beings and machines can reason about these logical theories.

We employ the amalgamation of object and meta-logic programs to model notions such as beliefs, goals and agent's "mental" activities. But we also accommodate less usual notions such as reactivity, openness, activation of goals and preference encoding, that have proved to be essential in realistic models of agents. Four logic programming languages to program agent with those features are introduced. We use an event-based approach to model dynamic universes with changing properties, concurrency and synergistic effects.

NOTE: This is a copy of the thesis with single spacing and smaller font than the original. Please, do not refer to the page numbering in here as this is different from that in the copies submitted to the University of London.

Acknowledgements

Many people helped me to complete this thesis. Thanks and apologies to any whom I do not mention here.

I specially want to thank my supervisor Bob Kowalski, for his systematic guidance and valuable support. Many thanks also to Fariba Sadri, Francesca Toni, Gerhard Wetzel, Murray Shanahan, Rob Miller, Keith Clark, Chriss Moss, Stephen Se, Torbjorn Semb D, Yongyuth Permpoontanalarp and the Temporal Reasoning, Artificial Intelligence and Logic (TRAIL) seminar group for many useful discussions.

Special thanks to Susan Peneycad for her careful proofreading of the thesis and for lending her intuition on the subtleties of the English language. I am also very grateful to Giorgio Tonella and Christoph Jung, for their opportune and useful comments.

My wife Liliana, my mother Gloria, father, family and friends deserve special mention for providing so much love and enthusiasm.

I gratefully acknowledge the support of the CONICIT-Universidad de Los Andes, Venezuela, for this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Statement of the general problem

This thesis presents a language to describe "agents". An agent is, in principle, any entity capable of intelligent and effective behaviour at problem solving.

There is an on-going debate in Artificial Intelligence (AI) and other disciplines as to what is the best way of describing and building agents. Among the recent polarizations in that debate, perhaps the most notable is that between those who support the traditional *unembodied* path and the supporters of the *embodied* approach [Tur50].

The *unembodied* approach assumes that the principles behind intelligent behaviour (and the structural components required for it) can be studied independently of particular realizations. Those defending the *embodied* approach, on the other hand, argue that the actual structural components of an agent determine how well it can achieve intelligent behaviour and that "[t]he intelligence of the system emerges from the system's interactions between its components - it is sometimes hard to point to one event or place within the system and say that is why some external action was manifested" [Bro91a].

In a technical review of the debate [Bro91a], Rodney Brooks explains how the *unembodied* approach has been the mainstream methodology of traditional Artificial Intelligence and how it has been unable to deliver satisfactory solutions. He contrasts it with his own pioneering work on the *embodied* approach to robotics, which has succeeded in building systems that display intelligent behaviour in tasks like locomotion, path-finding and manipulation of objects ([Bro86], [Bro91a]). Brooks' main point is that none of the robots built following this approach had anything that could be considered a representational or symbolic system in the traditional sense. Encouraged by these results, he goes on to suggest the rejection of any representation and symbolic reasoning mechanism in the construction of agents.

In this project, we share with Brooks the belief that some important elements have been missed or confused in traditional approaches to agent construction. However, unlike Brooks, we do not believe there is something inherently wrong in symbolic and representational approaches. On the contrary, we believe that the way forward to the understanding of the principles of agency is by overcoming the limitations of the formal languages used to specify, analyse and program agents. Wrong constraints on the languages may have caused these failures of the disembodied approach, mainly by confusing principles with particular realizations.

So, the problem attacked in this thesis is how to describe the elements of an agent so that they can be reasoned about in logical, implementation-independent terms. The logical description must lend itself to formal analysis, and eventually to some form of automation that

would implement the agent.

This thesis may thus appear to be following the disembodied approach. There is, nevertheless, an effort to incorporate in the descriptions those elements that Brooks and others have identified as lacking in previous attempts to characterize agency. Notions like reactivity, openness, interaction (with the environment and with other agents), indexicality (references to it*self*, the current position '*here*' and the current time '*now*') and bounded rationality can be catered for in the agents' models presented in this document.

However, we probably do just the opposite to that which advocates of embodiment would expect us to do. Throughout this thesis the reader will see a systematic attempt to avoid (postpone) specific involvement with implementational decisions. The intention is to elicit the abstract components of an agent and formalize them in a logical language. This formalization must be sufficiently expressive to guide the implementation, not only of the computational platform (e.g. whether it is a Von-Neuman computer or otherwise) but also of the sensorial and effecting components that the agent might require. It should be an implementation-independent description that can be used to support (machine or human) reasoning about the system.

Thus, the objective of this research is to explore ways of describing agents in logic. We have pursued a logical model that captures the everyday intuition of what an agent is.

1.2 What is an agent?

The "aim of Artificial Intelligence is the creation of artifacts capable of intelligent behaviour" [Isr93]. Although this is the widely accepted objective of AI, the emphasis of most research projects seems to be on obtaining a characterization of intelligent behaviour as independent as possible from the actual physical realization of the artifacts that display such a behaviour. The research community is already employing some terminology that permits reference to those abstract ideas. The term **Agent**, for instance, has been adopted to refer to an abstract entity that "can be viewed as **perceiving** its environment through **sensors** and **acting** upon that environment through **effectors**" [RN95]. It follows that an agent can be a robot, a program being executed, an animal or a human being.

Some researchers have suggested that an interesting resource in the study of agents is the attribution of *mental* capabilities to agents, similar to those possessed by humans. By adopting this *intentional stance* [Den87], [MSae90], one obtains a more expressive theory in terms of its explanatory and predictive powers (See [Lif90a], quoted in [Sho90]).

Consequently, we extend the previous definition to include those concepts which can be useful to the understanding of behaviour of any kind of agent, including human:

> An agent is an entity that can **perceive** its environment, can **assimilate** those perceptions accommodating them into a **memory** device, can **reason** on the information stored in that memory device, can adopt beliefs, **goals** and intentions for itself and can actively pursue the achievement of those intentions, by appropriate **control** of its **effectors**.

This definition commits itself to an embodiment including sensory, memory, processing and effecting devices. However, it sets no constraints on the actual form or structure of those devices.

Also, this definition is consistent the weak and the strong notions of agency discussed in a recent review of the technological practice in Distributed Artificial Intelligence by N. Jennings and M. Wooldrige [WJ95]. According to their *weak notion*, an agent must "have" *autonomy*: being able to set itself in a environment and control its effectors for its own purposes; *social*

ability: being able to perceive messages from other agents and reason and/or act upon those perceptions, perhaps generating messages itself [1]; *reactivity* (discussed below) and *pro-activeness*: being able to display goal directed behaviour and "taking the initiative" (also discussed below, but an immediate consequence of our definition of agent).

Notice that, although the definition above is of a single agent with no explicit social abilities, these abilities can be added to the "architecture" specified by the definition. For instance, Jennings-Wooldridge's strong notion of agency (.ibid) includes such things as *mobility*, which could be considered a consequence of the agent's control upon its effectors, that is, being able to properly move its body; *veracity* and *benevolence*: constraining its behaviour in multi-agent settings which will also require that the agent be able to reason about other agents and their attitudes (goals and beliefs), and *rationality*, a concept that will receive special attention (below in section 1.3.4) because traditional formal models of rationality have problems "capturing" the kind of rational capabilities of realistic, human-like agents.

1.3 Review of basic concepts in agent theories

1.3.1 Reactivity

Reactivity deserves special attention in the discussion about how to model intelligent agents. It has been the centre of the above mentioned debate between advocates and opponents of *Good Old fashioned Artificial Intelligence* (GOFAI). The main concern of the debate is the widespread opinion that the techniques originally used in Artificial Intelligence, characterized by the *representation of knowledge* are inappropriate or insufficient to generate "the swift, dynamic behaviour that implies real intelligence" [MSae90]. It has been suggested in that debate that disregard of the notion of reactivity has been the main cause of a number of problems in implementing effective, intelligent agents.

As has been mentioned, there are opinions suggesting that the best way of eliminating those problems is by avoiding the use of representations ("let the world be its own model" [Bro91a]). However, a purely reactive agent, like Brooks' robots, that simply responds to stimuli in its environment, lacks some of the characteristics that can be attributed to *an intelligent agent* (such as the capabilities for planning, hypothetical reasoning, introspection and reasoning about other agents).

Some researchers have tried to bridge the gap between these two viewpoints. The work of Genesereth and Nilsson[GN88] pioneered these attempts. Work by Kowalski *et al* ([Kow95], [DQ94], [KS97]) originated the project to reconcile reactivity and rationality that inspired this thesis. It was clear from those experiences that one had to make a systematic effort to understand the notion of reactivity.

Reactivity as a property of an agent can be seen as having two facets:

1.3.1.1 Architectural reactivity

First is the aspect of real-time input processing. The system (agent) must be set up so that the time to process the next input has an upper bound. The exhaustive, open-ended computation of traditional AI systems is simply inadmissible because it would prevent the system from responding to new, probably significant inputs.

Real time input processing means that the system must allow the suspension of normal processing in favour of a periodic checking for inputs, and the subsequent re-assuming of pro-

[1] Note that a language is required for that but it does not have to be a sophisticated system of written or spoken symbols. A *body language*, based on "gestures" could be good enough.

cessing, presumably from the point where it was stopped. Algorithms that support this kind of processing are re-entrant algorithms or *any-time* algorithms [DB88]. In the latter case (*any-time algorithms*) there is also the requirement for the "quality" of the outputs being a function of the time available to produce them (more time to compute → better quality in the solution).

Observe that eventual re-entering alone it is not enough. There must be a limit for the time within which the system returns to accept more inputs. At that time, any on-going process will be interrupted to check for data coming from the sensors.

Something similar occurs in time-sharing, computer operating systems. That interruptive strategy, known there as *preemptive scheduling* [Tan87], is not sufficient to support *real time performance*. An upper bound for the length of time between inputs is also required. This kind of reactivity is labelled here as *architectural reactivity*.

1.3.1.2 Knowledge-based reactivity

A second aspect of reactivity is intimately linked to the *knowledge structures* in the agent's knowledge base. As such, it is here called *knowledge-based reactivity*. If the knowledge of the agent is highly optimized and compiled into, for instance, condition → action rules, the agent can react to inputs in a minimal time with a great chance of success because of its timely response. In an extreme configuration, those condition-action rules are stored as a table or as a hardwired array that directly relates inputs from sensors to actions dictated to the effectors.

Unfortunately, success is not, in general, only a matter of timely response. There is also the need for a *proper* response that is a function of the situation (state) in which the agent is and the situation(s) in which it wants (or somebody else wants for it) to be in the future.

Condition → action rules are very efficient because, in that extreme configuration, there are direct links between sensors and effectors. When the rules are "hardwired" or compiled into some form of machine code, the response time is optimal. But this kind of configuration is also very rigid. The agent becomes a slave of its local environment. Inputs are always related to the same basic reactions in the immediate future. An example of a reactive architecture that relies on this strategy is Kaelbling and Rosenschein's situated agent ([KR90], [Ros89], [Kae87], [Kae90], [RK95]).

If one wants to recover the flexibility of run-time decision making and to provide for *thinking ahead* by the agent, the immediate choice is to introduce a forward reasoning system that processes the condition → action rules kept in a memory device and that maintains a record of those conditions that activate the rules. Conditions need not be related to inputs only. They can also be memory records set by special, *internal* actions. This is what is done in *production rules systems* such as those implemented on the OPS5 platform [Bro85].

But internal actions are still not enough. If the system is to support reasoning on configurations of the world at different times, it must include some sort of *time-stamping* of conditions and actions. This is one of the main features of *Agent0* (Agent-Zero) a model of an agent (presented by Shoham in [Sho90], [Sho95]) in which conditions and actions in commitment rules (*Agent0* equivalents of condition → action rules) all have time indexes.

Thus, the system must allow for the "triggering " of the conditions of its rules with newly sensed and previously recorded information, and then it must *mediate* among those rules that get "fired" to select the next actual action for the agent. When actions include changes to its memory device (as in *Agent0*), the agent can explore more complex responses to the environmental conditions by simulating other configurations of the world. The agent can then can obtain complex combinations of actions to be executed in the non-immediate future.

1.3.1.3 Goals in reactive agents

One inconvenience of the methods just described is that when the agent tries to "think ahead" it will face a combinatorial explosion in the number of "branches" to be explored. This problem can be dealt with by making the agent a *goal-oriented entity*. Goals, together with conditions, can be used to select the branches to be searched in the space of alternative plans of action.

Goals, though, have always been related to backward-reasoning systems. Typically, given a goal stating a condition to hold or an action to take place in the future, the reasoning system "backward-chains" related rules until it gets to the list of actions that the agent must execute, from which it chooses the first. This form of processing is called *regression*. Backward reasoning implies that the system must complete a plan to achieve its goals before it can take the the first action in that plan for execution. This is, of course, inadmissible for reactive systems as described above.

In this work we adopt a representational strategy (discussed in chapters 4 and 5) that combines the reactive, forward-directed nature of condition $\rightarrow$ action rules with the goal-orientedness of traditional planners based on regression. Our agent (our agent's planner) will still be reasoning backwards from goals to sub-goals, but the rules in the knowledge base are arranged so that the agent "moves forward" (also called forward planning or *progression* [Sha96]) in its search for actions to execute to achieve its goals. The rules themselves are of a more general form that link conditions, actions and goals.

Kowalski [Kow95] explains that condition $\rightarrow$ action rules can be subsumed by that more general form if one restores the goals that have been put away by *partial evaluation* [2] [Hog90].

1.3.1.4 Combining planning and execution

From what has been said so far one can already deduce some of the functional components of an agent. There must be a reasoning component (a theorem prover, for instance) that performs the reduction of goals to sub-goals. As this process eventually decides the actions the agent will perform to reach its goals, this component could be called *the planner*. There must be a memory device, from which the planner gets the rules and the information to trigger these rules. But the planner must also operate in close association with the component that controls the effectors, which we call *the executive*.

There have been many attempts to combine a theorem prover, a database, a planner and an *executive* to model rational agents ([Gre69], [All87], [GL90], [GN88] (and others mentioned in chapter 6). Some more recent systems have tried to solve the problem of reactive behaviour by combining modules for planning (or reasoning) and execution (for instance in PRS [GL90], [RW91] and [PIB87]). leaving to the *executive* the task of reacting to certain inputs with pre-established actions that require no planning.

However, in some of those systems there is a sharp (and sometimes *ad hoc*) separation between the reactive and the planning components. As the relation between the reactive and the reasoning component remains obscure, one does not know how the elaborate and complex mechanisms of forecasting, communication and cooperation could be smoothly incorporated into a rational and reactive system.

---

[2] For example, Kowalski says that the rule:

$$\textbf{if } \textit{it is raining } \textbf{then } \textit{carry an umbrella} \tag{1.1}$$

can be rewritten as:

$$\textbf{to } \textit{stay dry}, \ \textbf{if } \textit{it is raining } \textbf{then } \textit{carry an umbrella} \tag{1.2}$$

.

1.3.2 Intentionality

The previous section suggests that *goals* can play an important role in an agent's description. Yet, unlike conditions and actions, goals have no "physical grounding"[3]. Goals are "mentalistic" constructs that we attribute to agents when we adopt an *intentional stance* (see above) to explain their behaviour.

As a consequence of adopting the *intentional stance* for the construction of agents, the question arises of which mental constructs are the best to characterize an agent. Several answers are presented in the literature. Perhaps the most widely used are the so-called *Beliefs, Desires and Intentions* (BDI) Architectures ([Bra87], [RG95]).

Without going into detail, let us just say that "intentions" may not be regarded as a primary category as they can be derived from goals (desires) and beliefs. In this work, an intention could be defined as an agent's goal which at a certain moment has the *attention* of the agent. It is the one that if the agent decides to act at a certain time determines the action to be tried first at that time. In chapters 2 and 5, we show what we mean by a goal having the attention of the agent.

If goals represent actions or tasks to be performed in the world, there are two ways for the agent to generate new intentions out of goals that are being reduced to subgoals. An *optimized theory of actions and properties of the world* may allow the agent to reduce a particular task to a a minimal set of plans, the first of which is the *best plan* to accomplish the original task. This is the case in Expert Systems. In these systems, the conditions which provide optimality are compiled into the object level rules that describe the knowledge about a certain domain.

The alternative to that is to allow the agent to reduce its goals to a greater, non-optimal set of subgoals, and later to compare those alternative plans, to choose the best to suit the circumstances. In this case, the rules that cater for optimality are placed at the metalevel, the level of the descriptions that *talk about* the object level theory. Following this alternative, the agent would somehow be reflecting upon its goals, an operation that could be regarded as a sort of *introspection*.

Of course, this description does not explain how the agent gets its goals (and intentions) in the first place. What happens if at some point the agent has no goal at all. Here is where, we believe, inputs from the environment play a key role. Kowalski [Kow94] has explained how observations can be assimilated by agents, generating new goals by using integrity constraints as condition $\rightarrow$ action rules.

Having accepted the necessity of "mentalistic" abstractions, such as goals and beliefs, in the analysis of agents, one then has the problem of how to represent those abstractions. Issues like whether to use an object-level language only or to provide also for metalevel descriptions, and how to represent actions and events, arise. This has turned out to be one of the greatest challenges in Artificial Intelligence research, giving rise to one of its most prolific sub-fields: Knowledge Representation.

The following section introduces a general framework to analyse representational strategies and to justify the one selected in this thesis.

1.3.3 Representation of *problems*

A "problem" is the difference between the current situation and a desired situation. That difference is relative to a particular agency. A problem is always a problem for someone. There must be an agent (or a group of agents) that for some reason designates a problem as such.

[3] It is interesting to note that *actions* (events) were considered problematic in this respect until recent times. In section 1.3.3, we discuss some of the problems related to action and event representation in agency theory.

Moreover, the criteria used to decide what could be a *good* solution to a problem is also subject to that agent's relativism.

To solve a problem generally means to perform a set of actions in order to take some objects from their current situation to the desired configuration.

While acknowledging the elusive nature of the notion of problem, this work explores one possibility for formalising the process of solving problems.

Certainly, this is not an original objective. To design machines to solve general problems (as humans do) has been the main concern of Artificial Intelligence throughout this half of the twentieth century and has involved a lot of work in the formalisation of reasoning.

However, the possibility of having, not just one automatic system, but sets of machines interacting with themselves and with humans in a realistic environment to solve problems, is a more recent concern much more difficult to tackle than the original one. It is well accepted that *Good Old Fashioned Artificial Intelligence* offers no standard solution to the problem of designing a device that can solve problems while embedded in a dynamic environment.

For some people (see, for instance [Hew91]) the first casualties in the switch from centralized, monolithic systems to dynamic and multi-agent systems has been those systems based on deduction. For others any kind of system based on symbolic representation is inappropriate for modeling agents acting in a changing world ([Bro86], [Bro91a]).

In this thesis, we present arguments against those suggestions. An attempt is made to show that logic can be used to describe, model and implement reasoning agents that can function properly while embedded in changing environments.

The second technical aim of this work is to show how to specify a problem related to a *changing universe* and how to obtain its solution as a logical consequence of such specification and the set of rules embodying the knowledge required to solve it.

1.3.3.1 The classical notion of *problem*

The difference between the current situation and the desired or *intended* situation which, as has been said, characterizes a problem, can be made precise in two different ways:

1. One can describe the properties (of objects and agents) that hold at the initial and the final situations (e.g. *holds(at(me, work), now) and holds(at(me, home), later)* or,
2. One can state the intention of performing a particular action (e.g. *go(me, work, home,* $[T_1, T_2]$ *) and* $now < T_1 < T_2 < later$).

As it stands, that is the notion of a problem that has guided Artificial Intelligence since the early works on the subject. There exist many classical approaches to planning and problem solving based on that simple idea (for instance, [Kow79b], [FN71]) all of them built upon an almost standard formalization. Pednault defined it as:

> **Definition 0** *A* **classical planning problem** *is a quadruple of the form* $< \mathcal{W}, \mathcal{A}, \mathcal{I}, \mathcal{G} >$, *where:*
>
> 1. $\mathcal{W}$ *is the set of all possible states of the world.*
> 2. $\mathcal{A}$ *is the set of allowable actions.*
> 3. $\mathcal{I}$ *is the set of possible initial states.*
> 4. $\mathcal{G}$ *is the set of acceptable goal states.*

A solution to a classical planning problem is a sequence of actions $a_1 a_2 \ldots a_n$ that will transform the world from any of the possible initial states into one of the acceptable goal states [Ped87].

Traditional semantics for this kind of formalization say that "a state represents everything that is *true* of the world at a given point in time between one action and the next [Ped87]". This definition of a state leads naturally to the notion of a *situation* as a snapshot of the world, which has become the semantic core of the most widely employed formalism for knowledge representation in AI: The *Situation Calculus* [MH69]. Situations in the Situation Calculus can be defined as "individuals denoting intervals of time over which a fact holds or does not hold, but over which no fact changes its truth value. This latter property allows us to speak unambiguously about [which] facts are true or false in a situation" (Hanks and McDermott in [HM87]). Notice that for this to make sense the world must stay unchanged in order to define a situation.

Thus, the first, normally unstated, traditional assumption is that the relevant world cannot be in a continuous *state of change* [Gal95]. If one ignores this possibility then there is another immediate practical objection to the *situation-based approach*. It is that in any real application this approach is very likely to yield an intractable system, given the number of actions that should be considered even when those included are those that could *change* that world (see [Gin89], also discussed in chapter 6).

Yet, it is not only that the solver has to consider a huge number of world-changing actions or events. The really strong assumption is that the planner can somehow "control" those actions and decide its presence and its *location* within a plan, based on an assumed well-known set of effects that each action must have.

On the contrary, conventional wisdom says that the actual effects of an action are normally unknown or known with a degree of uncertainty before the action is executed and even after its execution. It is precisely because of this that adequate *feedback* is so important for a successful controller.

Furthermore, in classical semantics, actions are not allowed to be concurrent. Paralellism, and with it any *synergistic* effects that might arise from the simultaneous execution of actions, is ruled out or emulated by an early commitment to a sequential ordering.

This is definitely denying the real nature of the interaction between the executor of an action and the circumstances (environment) in which the action is executed or the purposive interaction between two actively independent entities. (consider two agents pushing an object in different directions, each one applying enough force to overcome the friction with the floor. The trajectory of the movement would be different if those forces are applied simultaneously instead of sequentially).

Thus, classical approaches face potentially high complexity in their representations, and yet they seem insufficient to capture primary intuitions about our changing universe.

Of course, these problems are not new. Researchers have been suggesting solutions to the problem of modelling action and change for a long time, no only in AI, but also in philosophy [Res66], computer science [Mil89], and other disciplines.

Within AI, interesting ideas have been presented as ways of patching the semantics of First Order Logic to cater for such dynamic worlds (including those with concurrent activities).

Pednault [Ped87] for instance, suggested, after borrowing the notion from McDermott [McD82], the idea of using a *chronicle* as a record of "all that is true, was true, and will be true of the world, from the beginning of time through to the end of time"(.ibid) What Pednault was doing was an early attempt to reinterpret what situations are, in an effort to follow McDermott in

going beyond the limitations of state-transition models (those that interpret situations as global states or snapshots of the world) and so being able to represent paralellism.

Similarly, Lansky [Lan87] emphasized the "duality between events and states". The work in the GEM concurrency model [LO83] incorporated several original notions to problem modelling. There, Lansky talked about events as *reified* entities, following the proposal of Davidson [Dav67] (see below).

Lansky also emphasized the notion of *location* through which the events are always related to "logical locations of occurrence". Events so structured, he said, help to organize the ways a domain is described.

But, the key contribution in work such as Pednault's and Lansky's is the switch from a conceptualization of events "solely in term of [..] state-changing functions" [Lan87] towards an event-based approach, where "the *state* of the world is represented in terms of the set of events that have occurred up to that moment"(.ibid).

One of the reasons this is helpful is because the resulting representations are inherently partial. One only needs to indicate those events *relevant* to a problem to restore the states of the world that condition the problem.

The influence of the philosopher Donald Davidson is all-pervasive in this "event-oriented" community. In a series of papers (collected in [Dav80]) Davidson presented convincing arguments for treating events as things that can be talked about in logical descriptions. That is *reification* of events. In "The Logical Form of Action Sentences" ([Dav80], page 118 and page 136)[4], it reads:

> [..] For example, we would normally suppose that 'Shem kicked Shaun' consisted of two names and a two-place predicate. I suggest, though, that we think of 'kicked' as a *three*-place predicate, and that the sentence to be given in this form:
>
> $$\exists X(kicked(shem, shaum, X)) \tag{1.3}$$
>
> If we try for an English sentence that directly reflects this form, we run into difficulties. 'There is an event X such that X is a kicking of Shaun by Shem' is about the best I can do, but we must remember that 'a kicking' is not a singular term. [..] nothing now stands in the way of giving a standard theory of meaning for action sentences, in the form of a Tarsky-type truth definition; [..] that is, of giving a coherent and constructive account of how the meanings (truth conditions) of these sentences depend upon their structure. [..] there is a lot of language we can make systematic sense of if we suppose events exist, and we know no promising alternative.

The key contribution of an event-based description is, according to Davidson himself, that "it explains more, and it explains better. It explains more in the obvious sense of bringing more data under fewer rules"(.ibid.). It turned out that explaining more with fewer rules has important computational advantages, as is explained below in the context of the so-called frame problem.

The efforts to develop alternatives to situation-based semantics have continued. Hayes [Hay85] appeals to the concept of a *history* as a way of dealing with the frame problem. Recently, Sandewall [San93] has re-introduced the analogous notion of *a chronicle* as part of a general semantics framework to analyse logics of change. Also, see Pelavin in [Pel91] for another attempt to address the limitations of the state-change model for planning and for a review of logics that were developed for that purpose.

[4] We have changed the notation to a Prolog-like form consistent with the one used in this document.

Advocates of the Situation Calculus [Lif90b] (page 247) have acknowledged its limitations and some have even started to re-interpret "situational" terms in the notation of SC as "histories" [Rei96].

Trying to profit from those experiences, the event-based conceptualization is used in this project, perhaps in a more radical way. Most of the inspiration of this work comes from the experience accumulated with the *Event Calculus* [KS86], a representational framework that has been shown to be useful for reasoning about change.

The *Event Calculus* has been proved to be as expressive as the *Situation Calculus* in some important respects (see [KS94]).

Both formalisms have been used to tackle the **frame problem** (FP): *in modelling a changing world, how to cater for those properties that do not change when some event happens*; the **ramification problem**: *how to deal with the implicit consequences of actions and events*; and the **qualification problem**: *how to define the circumstances under which an action is guaranteed to succeed* [RN95]. (Shoham [Sho89] suggested that these problems are the manifestation of a more general one: the *extended prediction problem*).

The frame problem (and its relatives) has also been characterized as *the problem of inertia* (or *the problem of persistence*): In a given language, how to state the fact that certain properties do not change unless some action makes them do so (*the representational frame problem*) and, once a representation has been adopted, how to infer changes and persistency of properties from it in an efficient way (*the inferential or computational frame problem*). The language of first order logic cannot be used on its own for that purpose because, among other things, it lacks the required *bias towards inertia*. First order logic does provide, however, an often denied resource to deal with non-monotonicity: *if-and-only-if definitions* which we use in this thesis (see chapter 3).

There has been an enormous volume of research concentrated on the Frame Problem since 1969, when it was first stated (See [MH69], [McC86], [HM87], [Bak91], [Lif91], [Kar94], [San94], [Sch94], [Sha97] in that order for a historical perspective). The aim (the desired solution) seems to be a general, domain independent language to model the frame problem and to specify the architecture that solves it. However, the emphasis so far has been more on building that bias for inertia into some logical language (syntactically or semantically) so that a proof procedure for that language can compute only the "intended" results. This has caused the production of a number of languages and nonmonotonic logics that can be used for particular classes of problems, but that do not have enough expressive power to describe other classes.

The Event Calculus, on the contrary, provides a natural way of representing the so-called *frame axioms* that address the frame problem (as we show in chapter 4). And because it is defined in first order logic, one can employ *if-and-only-if definitions* to deal with non-monotonicity. This observation and the previous remarks about the limitations of the classical approaches are the preliminary reasons for adopting an event-based approach to modelling for problem solving in this research project. This is done in chapter 4.

Before that, let us explain why we believe this approach is more general than the one based on state-situations only.

1.3.3.2 The notion of *problem* revisited

If one considers the classical idea of what a problem is (discussed in the previous section), it is clear why the state-situation approach was so attractive to the pioneer researchers in AI who tried to design problem solvers.

According to that approach, to define a problem, all that is required is to produce one description of the initial situation and another of the final situation.

Each description is a complete account of everything that is relevant to the problem: objects, attributes of those objects, agencies and especially, properties (of those objects and agents involved in the problem). Actions are simply (state)-*transitions* between pairs of descriptions.

These descriptions have to be "sufficiently" comprehensive to allow the problem solver to "rebuild" the *picture* of the world corresponding to each state-situation. Given these two *pictures* and a set of *transition rules*, the problem solver is responsible for "filling the gap" between them with perfectly ordered, intermediate pictures connecting the original two (i.e. a list of actions).

However, in a more *realistic* setting, when an agent is faced with a problem the following conditions normally hold:

1. Instead of an initial, global *picture* describing the environment at the time when the "*problem starts*", the agent normally has a few records of events scattered over different points in the past. From that record of events and its knowledge about persistence of properties (initiated by those events), the agent itself infers which properties hold and which do not hold in the so-called current state. So will it do with the final state.

2. The problem-solver agent is not forced to generate a perfectly ordered set of situations. A set of actions that *could* (in a normal world) start the desired properties, a *minimal* ordering of those actions and an account of the persistence of those properties would be enough as a plan. The agent *reserves the right* to update that plan while it is trying to execute it (by re-ordering or adding new actions as required by changes and updates from the environment). In this sense, the problem that the problem solver is trying to solve remains open.

To be precise about these two conditions and to clarify the concepts involved (like minimality of ordering, reordering and addition of actions and initiation and termination of properties), a new formalization of the notion of problem is given here:

Definition 1 *A problem $\mathcal{P}$, for an agent acting in an unpredictable environment, can be described as a tuple $\mathcal{P} =< \mathcal{T}, \mathcal{G}, \mathcal{IC} >$ where:*

- *$\mathcal{T}$ is a structured theory consisting of an* **historical record of the relevant world** *(what events have happened, including actions that have been performed) $\mathcal{H}$ and a description $\mathcal{K}$, of how (complex and atomic) actions affect objects and fluents*[5] *and how fluents persist over time.*

- *$\mathcal{G}$ is a set of* **goals** *that must be achieved at given points in time by those agents involved.*

- *$\mathcal{IC}$ is a set of* **constraints** *upon possible extensions Δ_i of $\mathcal{H}$. Being an extension of $\mathcal{H}$, Δ_i will include, in principle, all actions that should be performed by all the agents involved. However, an agent reasoning about $\mathcal{P}$ will have to reduce/link every external event to actions under its own agency, in order to use Δ_i as its individual plan to solve $\mathcal{P}$.*

In the traditional model theoretical semantics of logic, a solution to a problem $\mathcal{P}$ can be characterized by saying that $\mathcal{T}$, $\mathcal{G}$, $\mathcal{IC}$ and some Δ_i extending $\mathcal{H}$ must satisfy:

$$\mathcal{T} \cup \Delta_i \models \mathcal{G} \wedge \mathcal{IC} \tag{1.4}$$

[5] A *fluent* is a property that changes as time passes.

However, because Δ_i represents a plan to achieve $\mathcal{IC}$ and $\mathcal{G}$, one has to be more interested in the proof-procedural characterization of logical consequence that permits the generation of those extensions Δ_i:

$$\mathcal{T} \cup \Delta_i \vdash \mathcal{G} \wedge \mathcal{IC} \tag{1.5}$$

As it was suggested, a plan Δ_i will become the solution to the problem $\mathcal{P}$ once the agent has successfully executed it.

This formalisation is more general that the classical one (see **Definition 0** above) because: 1) the current state can be restored from $\mathcal{H}$, 2) the final, wanted state can also be constructed from $\mathcal{G}$ and $\mathcal{IC}$, 3) plans correspond to extensions Δ_i and 4) transition rules are subsumed by axioms and predicates in $\mathcal{K}$. In addition, the new formalisation allows for the openess and the minimality of ordering mentioned above.

Definition 1 can be further refined by combining $\mathcal{G}$ and $\mathcal{IC}$ in the set $\mathcal{IC}$ (or $\mathcal{G}$) itself. To allow for this, a semantical extension is required. As integrity constraints are normally expressed as conditional sentences, stating conditions that must be fulfilled by a knowledge base (here $\mathcal{T}$), it is intuitively appealing to regard them as conditional goals. Not only the agent has to strive to achieve its non-conditional goals, but it also has to cater for those goals with conditions. This movement also provides a beautiful semantic for *activation of goals*. Whenever all the conditions of a conditional goal hold, according to $\mathcal{T}$ and its extensions, the agent is compelled to pursue the unconditional goal represented by the consequent (the goal being activated).

All these abstractions are given concrete realizations in chapters 3, 4 and 5.

1.3.4 Bounded rationality

The previous formalisation of problem is intended to be part of a dynamic setting for problem solving. A problem-solver agent is regarded as an entity that has to deal with deduction, not for one fixed theory as usual, but from a continuous string of "theories" that change to accommodate new information from the external world.

In such a dynamic setting, an agent must be prepared to abandon a particular plan to solve the problem that has turned out to be infeasible in the light of the latest evidence. The current state can change while the agent is engaged in planning. What was derivable at some point as a feasible extension to the history can become inappropriate at some later point in time.

The agent then needs to reason about what can be derived from its knowledge at particular points in time. What were its *beliefs*, *goals* and *inputs* at certain points, whether a *plan* can be deduced for every goal and what was the outcome of every action that has been attempted to solve a particular problem.

But, what is more important, the agent needs to reason about the process of deriving plans itself: an agent must be able to decide when to stop reasoning because the time has come to act.

Some of those needs have been traditionally addressed at the specification level by extending the logical language with several "modal" operators to refer to, for instance, goals and beliefs (See [Moo95]) Combined with some modal temporal logic, those languages are very attractive due to their naturalness (descriptions are closer to natural language). However, the enriched semantics required by those languages (to accommodate the modal operators) inevitably introduces the omniscient agent problem[6] contradicting any attempt to model a realistic agent.

Kowalski has argued [Kow94] that the real problem is that the notion of logical consequence, establishing what can be derived or deduced from what is, in those logics, disconnected from

[6] An agent must know all the logical consequences of the knowledge it has. This is also known as *Perfect Rationality*.

the computational aspect of actually deriving those consequences. It can be said, for instance, that an agent cannot compute all the consequences of its knowledge simply because it has not got sufficient resources (time or storage) required to carry out the computation. An agent is a resource-bounded entity in the widest sense[7].

All these issues can be elegantly addressed if one distinguishes between an **object language** employed to describe the "constituents" $\mathcal{H}$, $\mathcal{K}$, $\mathcal{G}$ and $\mathcal{IC}$ of the problem $\mathcal{P}$ to be solved, and a **meta-language** that "puts together" those constituents of $\mathcal{P}$ and specifies the agent and its derivability relation ($\vdash$). When the distinction is established, one can *amalgamate* both languages to provide an overall specification of the whole system.

These ideas require further clarification. This thesis builds on a variety of results from logic and logic programming. The most basic of those results are in the theory of problem solving developed in [Kow79b] and [BK82]. In [Kow79b], Kowalski showed how the notion of *provability* of a language L_1 - how to deduce what sentences follow from what other sentences in that language - could be formalised by means of another language L_2, acting as the meta-language of L_1.

> "Given any two languages (i.e. systems of logic with their associated proof procedures) it may be possible to simulate the proof procedure of one language L_1 within the other L_2. The simulation is accomplished by defining in L_2 the binary relationship which holds when a conclusion can be derived from assumptions in L_1. Sentences in L_1 need to be named by terms in L_2 and the provability relation needs to be named by a binary predicate symbol, say **demo**[8], and defined by means of sentences **[DEMO]**[9] in L_2. Provided that the definition [DEMO] correctly represents the provability relation of L_1, simulation by means of [DEMO] in L_2 is equivalent to direct execution of the proof procedure of L_1. L_2, the language in which [DEMO] simulates L_1 is a meta-language for the object-language L_1. To serve as meta-language, L_2 needs to possess sufficient expressive power. For any object language the language of Horn clauses is already adequate" [Kow79b].

Moreover, Kowalski also explains(.ibid) that, although in the amalgamation of an object-language with its meta-language, one has the "impossibility [..] to completely formalise the notion of provability"[10], the amalgamated language "is more expressive than the object language alone".

Here we take advantage of that expressiveness. An amalgamation is used to describe the architecture of a generic agent and the knowledge that agent manipulates. The architecture includes a formalisation of the reasoning mechanism of the agent - the derivability relation that this agent employs - which is described in the meta-language. The knowledge is then "captured" by a description in the object-language. The amalgamation provides the means to describe systems with one or more agents interacting in a changing environment, among other things.

For instance, if *agent* is the "name" of the theory describing the agent *in the meta-language*, and its derivability relation is represented by the (meta-)predicate *demo*, the following provides a criterion for correctness for the agent's reasoning mechanism:

$$\textbf{if } agent \vdash demo(R, \lceil\mathcal{T}\rceil, \lceil\Delta_o\rceil, \lceil\Delta_f\rceil) \textbf{ then } Comp(\mathcal{T}) \models \Delta_o \leftrightarrow \Delta_f \tag{1.6}$$

[7] To distinguish it from the resource-bounding notion suggested by H.A. Simon in [Sim55], sometimes regarded as the narrow sense of resource bounding [HMP92].

[8] In [Kow79b] is *demonstrate*.

[9] In [Kow79b] they are called *Pr*.

[10] A result analogous to Godel's incompleteness of formal arithmetic and whose proof is also shown in [Kow79b].

where $\mathcal{T}$ is the agent's knowledge base ($\mathcal{K} \cup \mathcal{H}$), Δ_o represents a starting set of goals (including $\mathcal{IC}$) and Δ_f represents a new set of goals (which also includes $\mathcal{IC}$) computed from Δ_o with the amount R of resources available for computation. Notice that the expression $\lceil \alpha \rceil$ refers to the name of the object-level sentence α, as is written in the meta-language. $Comp(T)$ refers to the Clark completion [Cla78] of T, which provides a well-defined *closed world assumption*.

Basically, $\mathcal{T}$ and Δ_0 are being "frozen" while *demo* is computing Δ_f. This computation can be suspended when the agent runs out of "resources". While this computation is suspended, new information can be *assimilated* and so *demo* will probably have new parameters $\mathcal{T}'$ or Δ_0' when computation is resumed.

The fundamental point is that all these concepts can be described in the metalanguage itself, giving a declarative, logical account of the agent's cognitive and computational behaviour through time. This is achieved by the definition of the *cycle* predicate given in chapter 2.

In establishing relation 1.6, it is assumed that *demo* correctly represents the agent's derivability relation ($\vdash$). It is worth noting, however, that *demo* represents a *resource-bounded* version of the derivability relation. Thus, re-phrasing the *condition for correct representability* stated in [Kow79b] and [BK82], *demo* correctly represents $\vdash$ **if and only if**:

> for all finite sets of sentences A in the agent's knowledge base ($\mathcal{T}$ in a problem) and single sentence B (A and B in the object language),
>
> $$A \vdash B \textbf{ iff } \exists R \, (agent \vdash demo(R, \lceil A \rceil, \lceil B \rceil, \lceil \{\textbf{true}\} \rceil)) \qquad (1.7)$$

1.4 Statement of the specific problem

After that summary revision of the topics related to the modelling of agents, we are in the position to precisely state what the purpose of this thesis is.

The objective of the thesis is to describe agents by means of logical theories. An important consideration is that these logical theories should be some generalised form of logic programs. Like normal logic programs, the logical theories of agents must have an intuitive declarative reading and also a procedural interpretation to guide the implementation of automatic devices and software.

To solve the problem, we employ the amalgamation of object and meta-logic programs to model notions like beliefs, goals and agent's "mental" activities. We also incorporate less usual notions such as reactivity, openess, activation of goals and preference encoding, that have proved to be essential in realistic models of agents. Four (generalised) logic programming languages to program agents with those features are presented. We also explore an event-based approach[11] to deal with the modelling of dynamic universes.

To illustrate the sort of things this work could be applied to let us introduce an example that has been adopted as a benchmark in this and other research projects.

1.5 A motivating example: A program for an elevator

Benchmark examples are common in Artificial Intelligence literature. Researchers use them to reduce the complexity of some problem, clarify the general aspects involved and then focus the attention on those topics that are relevant for potential solutions.

[11] Which does not necessarily mean the Event Calculus as it is discussed in chapter 4.

The example shown here is borrowed from [LRL+95] where a GOLOG program is used to solve the same problem. We have kept the notation almost[12] unchanged to retain the simplicity of that example and to facilitate comparisons between our solution and theirs.

The solution presented first in chapter 2 is a family of logic programs written in the standard, PROLOG notation, and later (chapter 4) by means of some auxiliary programming languages.

1.5.1 What is the program for?

The purpose of the program is to control an elevator. The problem of controlling elevators has been approached by control engineers in many ways. There are well-known solutions to critical aspects, such as automatically (and safely) parking the elevator when energy power gets interrupted by some unanticipated event (like fire). Yet, it still seems to be an unsolved problem because of the diversity of optimality criteria. There are several variables that can be optimized. Some of them interfere with others, and then the problem is how to balance the arising trade-offs. Observe that this has to be done constantly over the working hours of the device, while the elevator keeps providing an adequate service for a highly uncertain set of *clients*. These are some of the criteria to be optimized [SMM96]:

- minimization of average waiting time per passenger,
- balancing of the boarding rate,
- minimization of the average service time,
- minimization of energy consumption by reducing flight time and the number of elevator starts.

Thus, controlling an elevator seems a natural task for an agent (or set of agents, in a building with more that one elevator). We believe[13] that such agents can be programmed by means of an appropriate logical architecture. In order to build the controller-program, [LRL+95] employs several abstractions that we preserve. The elevator is an agent that **can perform** the following primitive actions:

- *up(N)*: Go up to floor N,
- *down(N)*: Go down to floor N,
- *turnoff(N)*: Switch off the call signal at floor N,
- *open*: Open the door, and
- *close*: Close the door.

So, the agent will keep records like: $do(self, up(5), t_1, t_2)$, meaning that the elevator has gone (if t_1 and t_2 are past time-points) or is planning to go (if t_1 and t_2 are not both past time-points) up to floor five.

In addition, the agent **knows** about the following fluents:

- *currentfloor(C)*: the current floor is C[14],

[12] Except for the $current\_floor$ fluent.

[13] A belief that is shared by some of those who analysed this example before. See [LRL+95].

[14] Levesque et al. use $current\_floor(S) = M$ to say that the current floor is M in situation S. Instead of that, we say $holds(currentfloor(M), T)$ where T could be seen as referring to the equivalent of a situation in our framework.

- *on(N)*: the signal-call is on at floor N. Observe that, for simplicity, we treat all the calling buttons (outside the elevator, to fetch the caller, but also inside, to indicate destinations) as *on(N)* signals. Once the button is pressed, the signal turns on and stays so until it is disabled by *turnoff*.

Accordingly, the agent *believes* that the current floor is the fourth floor if it can demonstrate $holds(currentfloor(4), Now)$ with the information and the rules in its knowledge base at that time. Now is a (meta)variable whose value is equal to the current time according to the agent's internal clock. So, the agent should prove that:

$$demo(R, AgentKB, \{holds(currentfloor(4), Now)\}, \{\mathbf{true}\}) \tag{1.8}$$

holds for some R. Provided that R is small enough (so that the agent can actually finish that computation on time), the agent will then be able to consider the consequences of this belief for its future plans. The fluent $on(5)$, that represents the changing property of the calling button at floor five being on, is treated in the same way..

In [LRL+95], *nextfloor(N)* is also designated as a fluent (the next floor to be visited is N). We choose not to use this fluent. On the other hand, we add the *exogenous*[15] action *turnon(N)* (to press the button at floor N) that "explains" *on(N)*[16].

1.5.2 Policies of optimal behaviour for the elevator

From the list of variables that can be optimized (shown above), we have constructed a progressively more complex set of optimality criteria. The policies are:

1. **Serve each floor for which the call-signal is ON, in a FIFO fashion (first to call, first to be served)** . In chapter 2, section 2.3.2 this elementary policy is embodied in a simple set of integrity contraints and then used to illustrate the agent's architecture. Later, in chapter 4 a more sophisticated solution is described and compared with those given in previous works.

   Notice that this policy could be considered "fair" by some elevator users, but it can be terribly inefficient and energy wasting.

2. **Serve that floor for which the signal is ON and which is closest to where the elevator is at that moment**. This policy is specified by the formula:

$$\begin{aligned} holds(nextfloor(N), T) \equiv \\ & (holds(on(N), T) \ \wedge \ holds(currentfloor(C), T) \\ & \wedge \forall M \ (holds(on(M), T) \ \rightarrow \ (|M - C| \geq \ |N - C|)) \ ) \end{aligned}$$

   which is equivalent to the one that appears in the specification in [LRL+95], but is ignored in their implementation. This policy could be implemented as a program in more than one way in the framework here presented.

   The problem with this policy is that it leads to the possibility of stagnation in the services: A number of neighbouring floors can monopolize the elevator indefinitely if there is high enough demand for them. So, this policy must be rejected.

[15] An action performed by other agents.

[16] Observe that although the elevator agent cannot perform the action $turnon(N)$, it can reason about such an action being performed by other agents.

3. **Move to serve any floor requested so long as that does not imply a change of direction when there are still more floors to serve in that direction**. This more realistic and common policy is very easily obtained by exploiting some non-monotonic deductive mechanisms (as explained in chapter 4). The interesting thing is that this policy can be implemented without using a *nextfloor*, as Levesque *et al* did (.ibid).

4. **Set a deadline for every call and follow a route of "minimal energy", provided that it does not violate any deadline**. This more sophisticated policy requires an elaborate reasoning mechanism. We employ the expressive power of the amalgamated language to offer a general solution of this kind in chapter 4.

Eventually, we will be able to prove the following proposition (chapter 4):

Proposition 1 *Let ELE_PLAN be* { *do(self,up(5),*t_4*,*t_5*), do(self,turnoff(5),*t_6*,*t_7*), do(self,open,*t_6*,*t_8*), do(self,close,*t_9*,*t_{10}*), do(self,down(4),*t_{11}*,*t_{12}*), do(self,down(3),* t_{12}*,*t_{13}*), do(self,open,*t_{14}*,*t_{15}*), do(self,turnoff(3),*t_{14}*,*t_{16}*), do(self,close,*t_{17}*,*t_{18}*),* *done(self,park,*t_{18}*,*t_{100}*)* }.

Let $INEQ = \{t_0 < \ldots < t_{100}\}$, *where the* t_i *are time-points. Let ELE_T be the theory with the rules that tell how the agent should behave to control an elevator. ELE_T also includes information such as:* $holds(currentfloor(4), t_4)$, $do(somebody, turnon(5), t_1, t_2)$, $do(someone, turnon(3), t_2, t_3)$.

Then:

$$ELE\_T \cup ELE\_PLAN \vdash_{\textbf{iff}} done(control, t_4, t_{100}) \qquad \text{[ELEVA]}$$

where $done(control, t_4, t_{100})$ stand for the goal: *starting at time* t_4 *and finishing at* t_{100} *the agent must have* **done** *what is required to keep the elevator under control.* That is, serving every floor that has been requested by a call, according to the policy that has been established by the *employer* of the elevator controller.

Observe, however, that the interesting thing for us is not only to prove [ELEVA], but to specify the program that does the proving so that the agent can use such a program as its planner. That is the objective of chapters 3 and 6. This chapter describes an algorithm for planning that can admit (input) information of the forms $holds(currentfloor(4), t_4)$ and $do(somebody, turnon(5), t_1, t_2)$ to be updated into the the theory (ELE_T) virtually at any time in the planning process. Similarly, the *do* atoms in ELE_PLAN are progressively generated, most of them at different *re-entrances* to the planner process (i.e. after suspending planning to interleave this process with some other activity like the aforementioned assimilation of inputs). Chapter 3 will also explain how the planner algorithm is based on an *abductive* proof procedure.

1.6 Overview of the rest of the document

The information in the rest of this thesis is organized as follows:

Chapter 2 describes a model of an agent based on Kowalski's [Kow95] *cycle* predicate. The aim is to provide an architectural specification of agents that combine reactivity and rationality. The resulting general specification of an agent (called GLORIA), is used to describe a first, simple solution for the elevator controller.

Chapter 3 formalises the agent reasoning mechanism (the predicate *demo*) as an abductive proof-procedure, following the specification of the **iff** proof procedure proposed by Fung and Kowalski ([Fun96], [FK96]).

Chapter 4 presents a systematic approach to program agents. The chapter introduces OPENLOG, a logic programming language for writing programs that can be use for planning for problem solving. This chapter also introduces the idea of a *background theory*, formalising those concepts introduced above and completing a temporal and common-sense reasoning platform for agents' programming. A solution to our benchmark example is discussed in this chapter.

Chapter 5 introduces three more logic programming languages. ACTILOG can be used to describe conditions for activations of goals in an agent. PRIOLOG and USELOG can be used to embed heuristics for assigning priorities and utilities to an agent's goals.

Chapter 6 shows a brief review of the work in automatic planning in AI and explains how the general abductive mechanism presented in chapter 3 can be adapted for planning. The chapter also discusses a prototypical implementation of the agent described in the thesis.

Chapter 7 summarizes the thesis.

Chapter 2

Logic-based Agent Architectures

This chapter presents a specification of an agent architecture written as a logic program. The specification is based on a basic strategy to represent change and time, which we discuss in the first section. We then describe the functional components of an agent that define the architecture. This is followed by a first approximation to our benchmark example: the elevator controller, which serves to illustrate how the architecture works. The chapter ends with an analysis of the limitations and possible extensions of the architecture.

2.1 The representation of time in modelling an agent

Time is always an important element in a description of a system. Agents do not escape this rule. A description of what an agent is or does normally involves an account of time and of how properties change as time passes. A formal description of an agent usually includes a representation of time embedded in the language of the formalization (as in [GN88], [Kow95], [Sho95] and [RG95]).

The representation of time has been the subject of many interesting studies in Logic and Artificial Intelligence. Galton surveys fundamental ideas and the latest contributions in [Gal95]. He says that the simplest account of change can be obtained by attaching a term representing time to every predicate denoting a property that changes with time. For instance:

$$\neg cold(london, '31/07/96') \tag{2.1}$$

and

$$cold(london, '31/12/96') \tag{2.2}$$

are each a (partial) account of the state of the world on the respective dates and, together, an account of the *change* of temperature that has taken place in London *between* these dates. Galton (.ibid) also explains that this viewpoint, in which change is a derived notion (change is always a change of states), the basic notion is "state", and a state is a function of an "independent" time variable, can be traced back to the work of Galileo and Newton.

Despite its limitations (as discussed in chapter 1), the state-based model of time and change can give a good approximation of how objects and agents change. One can have a description of change based on extensional records of states like those above, which say that the temperature in London changed between the two dates. In addition, one can have an account of a *process* of change such that, given an state (say, the temperature of London at some point in time) one can deduce other states. Such an account is provided by the following definition:

$$
\begin{aligned}
&temperature\_london(Temp, T+1) \\
&\quad \leftarrow \ change\_temp\_london(Temp, Temp', T, T+1) \\
&\qquad \wedge \ temperature\_london(Temp', T)
\end{aligned}
\tag{2.3}
$$

together with the proper definition of the predicate *change_temp_london*.

It is important to note that the fundamental notion here is that of **a process**. A process is a sequence of changes of states. The sequence can be finite or infinite and is always related to an *invariant* of an object. In the example in 2.3 the invariant is *temperature_london*, which denotes the attribute *temperature* of the object *weather of London*. The description above is saying how the temperature of London changes from an instant to the next. In this sense, the definition in expression 2.3 above, is a description of the process of change of the temperature of London.

Thus, behind the description of an object there is a description of a process: the process that changes the object. By making explicit the notion of process one can easily introduce time into the representation of the object. And then, with object's states indexed by time, one can build a completely declarative account of how the object changes.

Observe that, if we describe the (state of the) object in a theory (a set of axioms), then the process should describe how that theory changes. And if it is so, a description of the process would be a metalevel description, i.e. a description of how the description of the object evolves.

It is also interesting to observe that, although Object Oriented Modelling and Programming techniques rely on the "simulation" of objects by mean of processes, the processes themselves are rarely accounted for in a formalisation (see, for instance, McCabe's Logic & Objects [McC91]). Even if time is not made explicit, by considering the process of change of the object one can reason about the effect of sequences of messages sent to the object.

Take, for example, a common object like a stack, described as a logic program with the following clauses:

```
% stack_process( StackState, Inputs )

stack_process( _, [] ).

stack_process( S1, I1 ) :-
  accepts_msg( S1, I1, S2, I2 ),
  stack_process( S2, I2 ).

accepts_msg( S1, [ push(Item)|Rest ], S2, Rest ) :-
  S1 = [ stack( Content ) ],
  S2 = [ stack( [Item|Content] ) ].

accepts_msg( S1, [ top(Item)|Rest ], S1, Rest ) :-
  S1 = [ stack( [ Item|_ ] ) ].
```

Any PROLOG system using that logic program will correctly answer the query:

```
? stack_process([stack([])], [push(10), push(20), top(X)] ).
```

affirmatively, with $X = 20$, whereas to the query:

$$process(D_o, T) \leftarrow changes(D_o, D_1, T, T+\epsilon) \\ \wedge\ process(D_1, T+\epsilon)$$

Figure 2.1: A formalisation of a process of change of an object

```
? stack_process([stack([])], [push(20), push(10), top(X)] ).
```

the system will also answer correctly that, in this case, $X = 10$. Time is, in this description, hidden behind the list structure of the inputs. That is, the assumption that inputs are ordered in the list by their "arrival times". However, the point is made that correct reasoning about the evolution of the stack through time is possible with a "theory" like the one in the logic program above.

Thus, a description of an object that displays a behaviour along time can include a definition of its process of change which can be used to explain and to reason about the object's behaviour. Such definition can be stated in the language of logic programming, by including, in the axiomatisation, a clause similar to that in figure 2.1, where D_o and D_1 are descriptions of two different states of the object, T is a time point and ϵ is a positive number. This must include, of course, a description of how state D_o turns into D_1, which is represented in fig. 2.1 by the predicate $changes(D_o, D_1, T, T+\epsilon)$ and that, in general, can be a complex description involving several predicates.

Compared with more general logics of change and actions, such as the Situation Calculus or the Event Calculus, this strategy to describe objects and how they change is more primitive but combines selected features of both calculi. It is close to the Situation Calculus in that it is a state-based description of change[1]. It is close to the Event Calculus in that events (changes) are indexed by time-points ($changes(D_o, D_1, T, T+\epsilon)$ above). However, descriptions like the one in fig. 2.1 do not need axioms for persistence to deal with the frame problem. The reason for this is that all the attributes of the object (the object's state) are explicitly represented as terms (in D_o and D_1 above). So, a change is a transition between total descriptions of states. This is a feasible approach (which does not have to face the frame problem) because one can assume that the object is a finite entity, with a finite number of attributes and components which can be explicitly accounted for in finite and not-so-big descriptions. However, these are one-object descriptions. If one is aiming at multi-object descriptions one would have to explain how changes in one object affects properties (attributes) of other objects. This would require integrating the description of each object into a more general calculus, such as those discussed in chapter 4.

In the following section we explain how these ideas can be applied to the modelling of a special type of object: an agent. To characterize an agent the description of the process of change which it goes through is specially important. Agents are objects with special attributes such as goals and beliefs. The behaviour of an agent is determined by the way in which these goals and beliefs change by observing, learning, planning and acting as time passes. The processes of reasoning/planning, observing and acting can be modelled by a description similar to that in figure 2.1, which turns out to have very useful computational characteristics, like bounded computation and reactive behaviour.

[1] For instance, one can write a description similar to that in figure 2.1 like this:

$$holds(processing(D_o), T_o) \leftarrow T_0 = result(changes(D_o, D_1), T_1) \\ \wedge\ holds(processing(D_1), T_1)$$

which, as it will be shown in chapter 4, could be part of an axiomatization in the form of Situation Calculus.

$$
\begin{aligned}
& cycle(KB, IC, Goals, Input.InRest, try(Act, Result).OutRest, T) \\
& \quad \leftarrow \quad assimilate(KB, IC, Goals, Input, KB', Goals', T) \\
& \quad \wedge \quad demo(KB', IC, Goals', Goals'', R) \\
& \quad \wedge \quad R \leq n \\
& \quad \wedge \quad try - action(KB', Goals'', try(Act, Result), KB'', Goals''', T + R + 2) \\
& \quad \wedge \quad cycle(KB'', IC, Goals''', InRest, OutRest, T + R + 3)
\end{aligned}
\tag{2.4}
$$

Figure 2.2: Kowalski's cycle predicate.

2.2 The *cycle* predicate

2.2.1 Kowalski's agent

In this research project an agent is conceived as a *cycling* process constantly interacting with an independent environment process. As (logic) programs, agents go beyond the *transformational* view that see programs as functions or relations between initial and final state. Instead, agents programs are *reactive systems* because "the purpose for which they run is not to obtain a final result, but rather to maintain some interaction with their environment" [Pnu86]. An agent acting in a dynamic and unpredictable environment has to interleave assimilating new information, planning and reasoning about its actions with their execution. Otherwise, its "survival" or the reliability of the system under its control is in jeopardy. Thus, an agent architecture should be *open* in the sense that it must allow for the addition of new information at any time during the process of solving problems.

A logical formalization of an agent with such capabilities has been proposed by Kowalski in [Kow95] with the definition of the *cycle* predicate. Kowalski is specially concerned with the notion of reactivity and its embodiment in a logical theory. His agent is basically a resource-bounded theorem prover that processes sentences from a knowledge base. This knowledge base contains a representation of the world as the agent perceives it. So, the representation is constantly updated. Preliminary versions of this model have guided the implementation of a multi-agent system simulation [DQ94] where each *agent's brain* is a PROLOG program. In [Kow95], *cycle* is defined[2] as shown in figure 2.2.

The declarative reading of the definition of *cycle* in fig. 2.2 simply specifies which sequences of inputs and outputs (as actions) are acceptable behaviour. This one-clause definition has a procedural reading that "requires the *cycle* predicate to be executed as a process concurrently with an environment process" [Kow95]. One can read *cycle* procedurally as follows:

To cycle at time T, an agent:

1. *assimilate*s (at time T and by means of its integrity constrains IC) an input into its goals $Goals$ (generating a new set of goals $Goals'$) and record the inputs in its knowledge base KB (generating a new knowledge base KB').

2. reduces goals (in $Goals'$) to subgoals (in $Goals''$) taking R units of time (or space) to do so. This reduction process is modelled by the predicate *demo* and is bounded to compute

[2] In this document we employ the usual conventions in clausal form representations. Variables should be seen as **universally quantified** over the whole clause, unless stated otherwise.

for at most n units of time (or space). To impose this constraint is the purpose of the expression $R \leq n$, which would be tested *simultaneously* with the execution of *demo*.

3. tries an action *Act* that it extracts from its goals $Goals''$ (which generates a new set of goals $Goals'''$), adding the outcome to its knowledge base KB' (which generates a new knowledge base KB''). Observe that this "attempt" to execute *Act* occurs at $T + R + 2$, which account for the fact that *assimilate* consumes 1 unit of time, *demo* consumes R and $try - action$ itself consumes 1. The assignment of one unit to both *assimilate* and $try - action$ is a convenient simplification that can be relaxed.

4. *cycle*s (to repeat the processes above) with its new set of "mental" structures $(KB'', IC, Goals''')$ and the remaining streams of inputs and outpus (*InRest* and *OutRest* respectively) at time $T + R + 3$.

Kowalski has suggested that an alternative to model-theoretic semantics for logic could be based on the syntax transformations embodied by those predicates in *cycle* and the notion of *knowledge assimilation* [Kow94].

Also, by appealing to the procedural interpretation of logic programs, these predicates (*cycle*, *demo*, *assimilate* and $try-action$) can also be interpreted as re-entrant coroutines with interleaved executions. In particular, the *demo* predicate (the theorem prover) has a resource argument R that limits the time or space of memory devoted to the deduction processes. Thus, R guarantees an eventual switching of the control from *demo* to $try - action$ in the same way as a scheduler, in a time-sharing operating system, switches the control among processes and/or processors. This interleaving yields an ongoing behaviour that combines reactiveness and rationality.

The definition of *cycle* captures the logical relationships between some of the processes that characterize an agent's "mental" activities. The mental activities mentioned above aim to explain how it is that an agent interacts with its environment (by assimilating *inputs* and acting) and reasons "at the same time".

In the following section *cycle* is described in more detail and definitions are given for all the other predicates except *demo*, the reasoner program, of which a full description is given in chapter 3 (although, a first informal mention of its characteristic is also made in section 2.3.2).

2.2.2 Improving *Cycle*

The *cycle* predicate in [Kow95] has already been given a declarative reading in terms of the manipulation of input and output streams. However, as it is, *cycle* has some shortcomings:

1. This cycle uses streams of inputs and outputs to interact with its environment. It is not clear to what extend inputs and outputs can be given a declarative account.

2. The procedural reading of *cycle* is based on the fact that processing goes on forever. There is no "base-case" that could be used to stop a derivation (computation) on *cycle*, even when there are no more inputs and outputs to process.

3. A critical point is the fact that the time line seems to be reversed. The present (denoted by the term T in the head of the definition in fig. 2.2) is defined in terms of the future (denoted by the complex term $T + R + 3$) in the condition.

In addition to these problems, there was also a mistake in the definition of $try - action$ in [Kow95]. According to that design, after successfully executing an action the agent commits

itself to a particular plan, throwing away alternative future courses of action that may be needed later on[3].

All these problems have been addressed and the resulting solutions are contained in the formalization below (table 2.1). Note that, in this reformulation, integrity constraints (IC) are separated from the "Frontier" of nodes ($Goals$). This is only a syntactic device that does not affect the consideration of integrity constraints as (conditional) goals themselves (See the example in section 2.3.2)[4].

Recall from chapter 1 that $\lceil\alpha\rceil$ is used to name the object-level sentence α. Any sentence surrounded by the symbols $\lceil$ and $\rceil$ is a sentence in the object language. Similarly, $\lfloor\beta\rfloor$ is used, in the object language, to embed the term β from the meta-language. So, any term (typically a variable) surrounded by $\lfloor\ \rfloor$ is a meta-term that is also a term in the object language. This distinguishes this kind of term from terms belonging only to the meta-language (e.g. $Rest$ and Alt in the sentences in the table) that act like *place-holders* for sentences in the object language. This symbolism is only used here to point out the interaction between the meta-predicate *cycle* and the object level specification of the agent's knowledge base in table 2.1. We will avoid its use later.

As final points, notice that, for simplicity, all actions are considered to take only one unit of time for their execution. So $do(Action, T)$ is left with only one time argument that makes its description easier. Also, for simplicity again, the agent can execute only one action at a time.

The formalization in table 2.1 has the following new characteristics:

1. Streams are abolished. A predicate-based formalisation of the mechanism of input and output is given through the definitions of the predicates $observe$ and $try-action$. The semantics of $observe$ is reminiscent of the semantics of the $query-the-user$ facility [Ser83]. The agent is "querying" the environment to ask for new inputs. As in query-the-user, the "knowledge of the world" is now seen as *distributed* across the system, part of it

[3]The problem is located in the first clause of the definition of $try-action$ in [Kow95]. The original version of that definition (The paper [Kow95] has been updated since) looked like this:

$$\begin{aligned}&\textit{try-action}(KB, Goals, try(Act, T, Result), KB', Goals', T) \leftarrow\\ &\qquad Goals \equiv (do(self, Act, T) \wedge Rest) \vee AltGoals\\ &\qquad \wedge\ Result = success\\ &\qquad \wedge\ Goals' = Rest\\ &\qquad \wedge\ KB' = (do(self, Act, T) \wedge KB)\end{aligned}$$

$$\begin{aligned}&\textit{try-action}(KB, Goals, try(Act, T, Result), KB', Goals', T) \leftarrow\\ &\qquad Goals \equiv (do(self, Act, T) \wedge Rest) \vee AltGoals\\ &\qquad \wedge\ Result = failure\\ &\qquad \wedge\ Goals' = AltGoals\\ &\qquad \wedge\ KB' = ((do(self, Act, T) \rightarrow false) \wedge\ KB)\end{aligned}$$

$$\begin{aligned}&\textit{try-action}(KB, Goals, try(nil, T, success), KB', Goals', T) \leftarrow\\ &\qquad \neg\exists Act, Rest, AltGoals\ [Goals \equiv (do(self, Act, T) \wedge Rest) \vee AltGoals]\\ &\qquad \wedge\ Goals' = Goals\\ &\qquad \wedge\ KB' = KB\end{aligned}$$

In the first clause, $Goals' = Rest$ must be $Goals' = Rest \vee AltGoals$ to allow for future actions failing and the need for "backtracking" then. Kowalski discovered the error immediately after the paper went to publication. There is no such error in the PROLOG prototype [DQ94] of the agent. To reconcile this procedural version with a declarative reading turned out to be an interesting challenge.

[4]Observe that for this specification to work, there must be an integrity constraint for every kind of input to be assimilated. Otherwise, axiom [ONL-ASSI] could not be used for all the inputs. One can either add another axiom for inputs for which there are no integrity constraints, or add "dummy" integrity constraints such as $obs(p, T) \rightarrow$ **true** to avoid altering the specification.

Also, notice that, for simplicity, axiom [ONL-ASSI] caters only for *observation of properties* ($obs(P, T)$). To allow for observation of actions (such as $do(A, T)$), the specification must be extended with an axiom in which integrity constraints of the form $do(A, T) \rightarrow C$ are used as well.

inside the agent, the other part outside in the environment. A striking point is that the same interpretation can be applied to $try - action$. In this context, when an agent *tries* an action it is asking the environment for information about such an action. It is asking for *feedback*.

2. A "base-case" for *cycle* is provided. We adopt a specification with a single-atom clause stating that *nothing* (in the agent) changes in a *empty interval (from T to T)*.

3. However, the most important feature of this reformulation is that time references are more intuitive. In this new definition what an agent does within a period of time $[T_o, T_f]$, is defined in terms of what it does in sub-periods included in the time period $[T_o, T_f]$.

Briefly, table 2.1 can be read as follow:

cycle The "mental" structures of an agent (knowledge base and goals) do not change in an empty period of time [CYC-0]. On the other hand, over a non-empty period of time a number of information processing activities will constantly transform those structures [CYC-1]. These activities can be seen as independent, parallel processes, synchronized through data exchange. They can also be seen as processes time-sharing a single processor and therefore, each one with a "slice" of every unit of execution-time. For simplicity, that slice of time is set here to 1 for all the activities except *demo*, whose execution-time boundary is explicitly set by $R \leq n$.

It is important to clarify the role of the constraint $R \leq n$. This constraint is tested "in parallel" with the execution of *demo*, in such a way that reasoning is suspended when the "resources" (counted by R) are consumed.

The value of n could also be dynamically set up to indicate, within a particular cycle, that a significant input has arrived and reasoning must be suspended to deal with the incoming information. This would be the way to simulate "system interruptions" as traditionally understood in operating systems.

These considerations about $R \leq n$ are, of course, relative to the control strategy used to execute *cycle* and do not affect its logical form.

observe The agent gets "new" data by querying the environment once in every period of cycling. The intuition is that the environment's knowledge base must be able to say what "properties" (of the environment) the agent observes.

assimilate As in [Kow95], knowledge assimilation is considered an activity with two processes. There may be an *off-line assimilation* caring for the consistency and integrity of the knowledge base, along the lines discussed in [Kow79b] and [Kow95]. However, the one described here is **on-line assimilation**, the process that queries the environment for an input, *time-stamps* it (with the current value indicated in an "internal clock" T), and then says how that input can "fire" a condition-action, integrity constraint. In addition, the knowledge base records the input and new goals get activated by being put into the set of goals being processed.

try-action Finally, within every period of cycling, there is also a review of the goals searching for an action to be executed. If there is no pending action, then there is no modification to be made to the "mental structures" [TRY-AC3]. On the other hand, if there is one or more pending actions, the system will try that action and collect its feedback from the environment about success or failure of that action. This is done through the meta-predicate *try*. If the action "succeeds", it is recorded into KB [TRY-AC1]. If the action

CYCLE : the locus of control of an agent	
$cycle(AgentKB, Goals, IC, T, T)$	**[CYC − 0]**
$cycle(KB, Goals, IC, T_o, T_f)$ $\leftarrow\ assimilate(KB, Goals, IC, KB', Goals', T_o)$ $\wedge\ demo(KB', Goals', IC, Goals'', R)$ $\wedge\ R \leq n$ $\wedge\ try - action(KB', Goals'', IC, KB'', Goals''', IC', T_o + R + 1)$ $\wedge\ cycle(KB'', Goals''', IC', T_o + R + 2, T_f)$	**[CYC − 1]**
$observe(P, T)$ $\leftarrow\ sensors'\ specific\ conditions..$ $\wedge\ conditions\ tested\ by\ the\ environment..$	**[OBS]**
$assimilate(InKB, InGoals, IC, OutKB, OutGoals, T)$ $\leftarrow\ observe(P, T)$ $\wedge\ IC \equiv\ \lceil\ (obs(\lfloor P \rfloor, \lfloor T \rfloor)\ \rightarrow\ C) \wedge\ RestIC\ \rceil$ $\wedge\ OutKB = \lceil\ obs(\lfloor P \rfloor, \lfloor T \rfloor)\ \wedge\ InKB\ \rceil$ $\wedge\ OutGoals = \lceil\ C \wedge InGoals\ \rceil$	**[ONL − ASSI]**
$try - action(KB, Goals, IC, KB', Goals', IC, T)$ $\leftarrow\ Goals\ \equiv\ \lceil\ (\ do(\lfloor A \rfloor, \lfloor T_1 \rfloor) \wedge Rest\ )\ \vee Alts\ \rceil$ $\wedge\ Goals'\ \equiv\ \lceil\ (\ do(\lfloor A \rfloor, \lfloor T \rfloor) \wedge Rest\ ) \vee$ $(\ do(\lfloor A \rfloor, \lfloor T_1 \rfloor)\ \wedge\ \lfloor T_1 \rfloor \neq \lfloor T \rfloor\ \wedge\ Rest\ )\ \vee Alts\ \rceil$ $\wedge\ try(A, T, succeeds)$ $\wedge\ KB'\ \equiv\ \lceil\ do(\lfloor A \rfloor, \lfloor T \rfloor)\ \wedge\ KB\ \rceil$	**[TRY − AC1]**
$try - action(KB, Goals, IC, KB, Goals, IC', T)$ $\leftarrow\ Goals\ \equiv\ \lceil\ do(\lfloor A \rfloor, \lfloor T_1 \rfloor) \wedge Rest\ )\ \vee Alts\ \rceil$ $\wedge\ try(A, T, fails)$ $\wedge\ IC' = \lceil\ (\mathbf{false}\ \leftarrow\ do(\lfloor A \rfloor, \lfloor T \rfloor)) \wedge IC\ \rceil$	**[TRY − AC2]**
$try - action(KB, Goals, IC, KB, Goals, IC, T)$ $\leftarrow\ \neg\exists A\ (Goals\ \equiv\ \lceil\ do(\lfloor A \rfloor, \lfloor T \rfloor) \wedge Rest\ \vee Alts\ \rceil)$	**[TRY − AC3]**
$try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment..$	**[TRY]**

Table 2.1: A new *cycle* predicate

"fails", this will cause the addition of a new integrity constraint (as shown in [TRY-AC2]), which would invalidate any plan containing the failing action.

demo This predicate (whose definition is not shown in table 2.1) is responsible for reducing goals to subgoals by *unfolding* (a transformation defined in chapter 3). But, *demo* is also responsible for catering for the effects of the success and failure of actions, on the agent's goals. For instance if, as a consequence of a failed attempt to execute $do(a, t_o)$, $try{-}action$ generates the new integrity constraint $\mathbf{false} \leftarrow do(a, t_o)$, then any plan containing the action must be cancelled (because, it leads to falsity i.e. $(do(a, t_o) \wedge (\mathbf{false} \leftarrow do(a, t_o))) \equiv \mathbf{false}$) This is another task for the reasoner (implemented by *demo*).

Notice that this transformation is very similar to the assimilation of inputs described by [ONL-ASSI]. This similarity led us to unify the process of *activation of goals* as described by the proposal in the following section, which is supported by the formalizations in chapters 3, 4 and 5.

2.3 GLORIA

Although the last version of *cycle* (in table 2.1) improves upon the original one, it still lacks some intuitive properties of agents. Some of its limitations are that agents can execute only one action at a time and assimilate one "unit" of input-data per cycle. As a consequence, the specification may be too committed to a mono-processor implementation.

What follows is a new specification that overcomes these limitations. Three basic changes with respect to the specification above are made: 1) Acting and observing are integrated as *one single process*, 2) Activation of goals is no longer a task of the *assimilating process*, but is transferred to the reasoning mechanism (*demo* in chapter 3) where it will be integrated with other goal-transforming operations and 3) The assimilation procedure is changed to have inputs (observations and feedbacks after executing the actions) attached to *the goals* (above they are added to KB). The reason for the subtle change will be discussed and clarified in the following chapters. This change implies a major change of perspective with respect to the content of the knowledge base as usually understood.

All this yields a description that is simpler to analyse and to use (as will be seen in section 2.3.2). To make it even simpler we go back to a definition of *cycle* with no base case, the time order reversed and integrity constrains grouped with the other goals. One can always reformulate the definitions as has been shown. Note that the restriction of actions taking only one unit of time for execution is still maintained.

The new description is called **GLORIA**[5].

2.3.1 GLORIA's specification

GLORIA's cycle is specified in table 2.2. This may be read as follow:

- **cycle** The agent's mental structures are changed by the interaction of two processes: *demo* (explained below) that reduces goals to subgoals and activates new goals and *act* that "queries the environment". The agent posts a query string and the environment answers with an input string. Note that in the agent's query string there may be (and normally are) *observational actions* that command the sensors to observe in a particular way. Thus, this model contemplates that an agent must be capable of focusing its sensors.

[5] A **G**eneral-purpose, **L**ogic-based, **O**pen, **R**eactive and **I**ntelligent **A**gent.

GLORIA's cycle	
$cycle(KB, Goals, T)$	
$\leftarrow\ demo(KB, Goals, Goals', R)$	
$\wedge\ R\ \leq n$	
$\wedge\ act(KB, Goals', Goals'', T+R)$	
$\wedge\ cycle(KB, Goals'', T+R+1)$	**[GLOCYC]**
$act(KB, Goals, Goals'', T_a)$	
$\leftarrow\ Goals \equiv\ PreferredPlan\ \vee\ AltGoals$	
$\wedge\ executables(PreferredPlan, T_a, TheseActions)$	
$\wedge\ try(TheseActions, T_a, Feedback)$	
$\wedge\ assimilate(Feedback, Goals, Goals')$	
$\wedge\ use\_order(Goals', Goals'', R_{use})$	
$\wedge\ R_{use} \leq k$	**[GLOACT]**
$executables(Intentions, T_a, NextActs)$	
$\leftarrow\ \forall A, T(\ do(A,T)\ is\_in\ Intentions$	
$\wedge\ consistent((T = T_a) \wedge Intentions)$	
$\leftrightarrow\ do(A, T_a)\ is\_in\ NextActs\ )$	**[GLOEXE]**
$assimilate(Inputs, InGoals, OutGoals)$	
$\leftarrow\ \forall A, T, T'(\ action(A, T, succeed)\ is\_in\ Inputs$	
$\wedge\ do(A, T')\ is\_in\ InGoals$	
$\rightarrow\ do(A, T)\ is\_in\ NGoal\ )$	
$\wedge\ \forall A, T, T'(\ action(A, T, fails)\ is\_in\ Inputs$	
$\wedge\ do(A, T')\ is\_in\ InGoals$	
$\rightarrow\ (\mathbf{false} \leftarrow do(A, T))\ is\_in\ NGoal\ )$	
$\wedge\ \forall P, T(\ obs(P, T)\ is\_in\ Inputs$	
$\rightarrow\ obs(P, T)\ is\_in\ NGoal\ )$	
$\wedge\ \forall Atom(\ Atom\ is\_in\ NGoal$	
$\rightarrow\ Atom\ is\_in\ Inputs$	
$\wedge\ OutGoals\ \equiv\ NGoal\ \wedge\ InGoals$	**[GLOASSI]**
$A\ is\_in\ B\ \leftarrow\ B\ \equiv\ (A\ \wedge\ Rest\ )$	
$\vee\ B\ \equiv\ (A\ \wedge\ Rest \vee RemDisj\ )$	**[GLOISN]**
$try(Output, T, Feedback) \leftarrow\ tested\ by\ the\ environment..$	**[TRY]**

Table 2.2: **GLORIA**'s *cycle* predicate

Of course, what the agent actually senses depends on the properties of its environment at that time.

- **act** This process changes the agent's mental structures at a given time if, out of its "preferred plan" the agent selects those actions that *can be* executed at that time, posts them (in parallel) to the environment and analyses and assimilates the obtained feedback. The agent's preferred plan is the plan upon which the agent is concentrating its attention, that is the first in a (heuristic) ordering of all the goals. The process that orders the plans is part of the reasoning goal-processing mechanism and will be explained in chapter 5 and chapter 6.

- **executables** These are all the actions in the agent's preferred plan whose execution time can be equated with the current (internal) time. Notice that this is not a simple test of equality. There may be an instantiation involved because the actions in the plan normally do not have a specific predefined time for execution. So, the consistency test will check the constraints on the time arguments of each action to find out whether they can safely take the value indicated by T_a.

- **assimilate**. This new form of assimilation relies on a more "informative" feedback that not only says whether particular actions fail or succeed, but it also transmits observations (see the example in the next section).

  Two details are worth emphasizing:

  1) Observations of actions and properties (*do* and *obs* atoms) are no longer "stored" in the knowledge base, but they are all (irrespective of success or failure in the case of the actions) added to the goals *as new constraints*. This change will make sense after the reasoning mechanism has been explained (in chapter 3);

  2) These definitions of assimilation and act lend themselves to an attractive extension when actions with duration (with starting and finishing times) are introduced into the model. One of the most important items of information that one could expect to obtain *via feedback* is the actual terminating time of an action. The agent may decide when to start executing an action, but the finishing time depends on what is going on in the environment at the time of the attempt. Being able to enter this information at "run-time" is very useful to the models discussed here.

- **use_order** The other component of the specification introduced in this definition is the logic program *use_order*, itself defined and discussed in chapter 5. With this program, the agent will evaluate the usefulness or otherwise of the currently preferred plan. If the feedback indicates a failure on this plan, the agent may want to review its frontier of goals and **focus its attention** on another plan that could be more successful. Alternatively, it could try again the failing string of actions again. This type of analysis is *context-dependent*. That is, what the agent does after this analysis depends, in general, on its current situation and on a domain-specific set of preferences[6].

  Thus, *use_order* implements a form on reasoning different from *demo*'s. It is also a resource-bounded reasoner. The resources, R_{use}, consumed by *use_order* are restricted by $R_{use} \leq k$ where k is some predefined constant value. However, unlike *demo*, *use_order* reasons about preferences of the agent and utilities of the plans.

[6]There are, however, simple $ad-hoc$ solutions like removing those plans with failures from the frontier altogether and then covering their absence with clever programs at the object level. We do something like that in the testing solution for the elevator controller as shown below and in chapter 6.

The presence of *use_order* in *act* is not essential, as it is also called from within *demo* (as shown in chapter 3). However, its inclusion in the definition of *act* serves to emphasize the point that after execution, the agent has feedback information that may determine its immediate behaviour. In particular, a prompt decision about what plan to consider immediately after an action execution can affect the efficiency of the agent, as we discuss in chapter 6.

The following section shows how the GLORIA specification can be used to model the elevator controller.

2.3.2 GLORIA featuring as the elevator controller

2.3.2.1 A first look at activation of goals

As a first approximation for the elevator example, let us imagine that an agent (with an architecture as specified in the previous sections) stores its goals in *Goals*. The content of *Goals* is in general a disjunction in which each disjunct represents an alternative goal of the agent.

Let us also imagine that, at a certain moment, the agent has only one disjunct in *Goals*, which is itself a conjunction of the following integrity constraints:

$$\begin{array}{l}\exists T_1\ \exists T_2\ (\\ \quad (\ T < T_1\ )\ \wedge\ (\ T < T_2\ )\ \wedge\\ \quad (\ cons\_do(up, T_1)\ \leftarrow\ M < N\ )\ \wedge\\ \quad (\ cons\_do(down, T_1)\ \leftarrow\ N < M\ )\ \wedge\\ \quad (\ (\ cons\_do(open, T_1)\ \wedge\\ \qquad cons\_do(turnoff, T_1)\ \wedge\\ \qquad cons\_do(close, T_2)\ \leftarrow\ N = M\ )\ )\\ \qquad \leftarrow\ atfloor(M, T)\ \wedge\ on(N, T)\ ) \qquad \textbf{[ICSERVE]}\end{array}$$

$$\begin{array}{l}\exists T'\ (\\ \quad (\ T < T'\ )\ \wedge\\ \quad (\ (\ do(A, T)\ \wedge\ cons\_do(B, T')\ )\\ \quad \vee(\ do(B, T)\ \wedge\ cons\_do(A, T')\ )\ )\ )\\ \qquad \leftarrow\ cons\_do(A, T)\ \wedge\ cons\_do(B, T)\\ \qquad \wedge\ A \neq B\ \wedge\ incomp(A, B) \qquad \textbf{[ICCONSI]}\end{array}$$

$$do(A, T)\ \vee\ (\ \textbf{false} \leftarrow do(A, T)\ ) \leftarrow\ cons\_do(A, T) \qquad \textbf{[ICDO]}$$

$$\textbf{false}\ \leftarrow\ do(A, T)\ \wedge\ do(B, T)\ \wedge\ incomp(A, B) \qquad \textbf{[ICINCON]}$$

where $cons\_do(A, T)$ can be read as "consider doing action A at time T", while *do(A, T)* is read as "do action A at time T" (the other predicates are explained below).

With respect to the agent's knowledge base, stored in KB, let us assume that it contains the following definition:

$$\begin{array}{l}incomp(A, B)\\ \qquad \leftrightarrow\ interf(A, B)\ \vee\ interf(B, A) \qquad \textbf{[INCOMP]}\end{array}$$

$$\begin{array}{l}interf(A, B)\\ \qquad \leftrightarrow\ (\ (A = up \wedge B = down)\\ \qquad \vee(A = down \wedge B = open)\\ \qquad \vee(A = up \wedge B = open)\\ \qquad \vee(A = close \wedge B = down)\\ \qquad \vee(A = close \wedge B = up)\\ \qquad \vee(A = open \wedge B = close) \qquad \textbf{[INTERF]}\end{array}$$

In addition to the knowledge embodied by the integrity constrains and by the definition above, the agent has access to some built-in mechanism to compute for given numbers N and M, whether $N < M$, $M < N$ or $N = M$ and, for every pair of symbols, A and B whether $A = B$ or $A \neq B$.

Finally, the agent also has access to "changing" information about its physical location ($atfloor(M,T)$) and, as any elevator, about which calling buttons have been pressed ($on(N,T)$)[7].

Let us now imagine that the agent has been *cycling* as the specification of GLORIA prescribes. In its last cycle, the agent collected the information $atfloor(3,0)$ and $on(5,0)$.

At this point, the agent's reasoning mechanism (the reasoner) starts processing that information. A typical derivation (a reasoning sequence) could be:

1. The agent "propagates" the atoms $atfloor(3,0)$ and $on(5,0)$ through the integrity constraint ICSERVE above. This form of processing is using the integrity constraint as a $condition \rightarrow action$ rule. After "proving" the conditions, the "head" of the outermost implication is activated. This yields the new implication ICSERVE2:

$$\begin{array}{l} \exists T_1\ \exists T_2\ (\\ \quad (\ 0 < T_1\ )\ \wedge\ (\ 0 < T_2\ )\ \wedge \\ \quad (\ cons\_do(up, T_1)\ \leftarrow\ 3 < 5\ )\ \wedge \\ \quad (\ cons\_do(down, T_1)\ \leftarrow\ 5 < 3\ )\ \wedge \\ \quad (\ (\ cons\_do(open, T_1)\ \wedge \\ \qquad cons\_do(turnoff, T_1)\ \wedge \\ \qquad cons\_do(close, T_2)\ )\leftarrow\ 5 = 3\ )\ )\ ) \end{array} \quad [\textbf{ICSERVE2}]$$

2. From ICSERVE2, the agent obtains:

$$\exists T_1\ (\ (\ 0 < T_1\ )\ \wedge\ (\ cons\_do(up, T_1)\ )\ )\ \wedge \quad [\textbf{ICSERVE3}]$$

   This is, of course, due to the fact that only the corresponding inequality in ICSERVE2 ($3 < 5$) can be proved.

3. This derivation seems to be culminating with the basic sub-goal $cons\_do(up, T_1)$ being activated. However, $cons\_do(up, T_1)$ is a special type of goal which can be "propagated" through constraint ICDO, above, to yield:

$$\begin{array}{l} \exists T_1\ (\ (\ 0 < T_1\ )\ \wedge \\ \quad (do(up, T_1)\ \vee\ (\textbf{false} \leftarrow do(up, T_1))\ ) \end{array} \quad [\textbf{ICDO2}]$$

4. At this point the reasoner has activated a conjunction, one of whose conjuncts is itself a disjunction. After distributing the conjunct $(0 < T_1)$ over the nested disjuncts, the reasoner "splits" the disjunction, to generate two new alternative goals. Presumably, the agent will "prefer" the new goal:

$$\exists T_1\ (\ 0 < T_1\ )\ \wedge\ do(up, T_1) \quad [\textbf{FOREXE}]$$

   because it contains a *do* atom representing a primitive action that might be executed.

[7] For simplicity we are avoiding the "reified" representations (i.e. $holds(at(M,T))$ and $holds(on(N),T))$ introduced in chapter 1.

So, the derivation leads to the activation of *do* atoms. These atoms, as *consdo* atoms, can themselves be propagated through other constraints (as ICINCON above). However, more importantly, *do* atoms represent atomic actions that the agent can try for execution when the reasoning is suspended, and the control is passed to *act*, as described by the specification of GLORIA.

Thus, a list of *do* atoms ("activated" as above), represents a plan of actions for the agent. They constitute a plan in the sense that the agent will be acting according to them, to satisfy its integrity constraint that, as we discussed in chapter 1 can be seen as goals for the agent.

Of course, to guarantee that the plan (the set of *do* atoms) is consistent with all the integrity constraints and the rest of the agent's knowledge, the reasoning mechanism would have to explore all the possible derivations. This could take a lot of time or memory space. This is why the agent is restricted (by the constraint $R < k$ on *demo* in table 2.2) to interrupt is reasoning and present its "partial" plans (the set of *do* atoms activated so far) to the process *act* that attempts their execution.

The agent needs a *demo* predicate capable of all these different transformations and knowledge manipulations. A proposal in that direction is the subject of chapter[8] 3.

2.3.2.2 A first look at the implementation of the elevator controller

A key element in the implementation is the data structure to store the agent's set of goals (*Goals*). We use lists and tuples.

The set of goals *Goals* of the agent can be kept in a tuple $(IC, Frontier)$, where IC is a list (an and-list) of structures storing sentences like [ICSERVE] above (conditional sentences).

$Frontier$ is a list of tuples, which we call nodes. Each node in $Frontier$ has the following structure:

$$Node = (Uncond, Cond) \tag{2.5}$$

$Uncond$ stores *do* atoms representing actions that the agent must **try** for execution. $Uncond$ can, therefore, be said to contain a plan to achieve certain goals that have been *activated* by processing of IC, as we illustrated above. $Uncond$ also contains *cons_do* atoms used to guide the search for correct and feasible sets of *do*'s, i.e. combinations of actions that satisfies the integrity constraint [ICCONS], [ICDO] and [ICINCON] above.

The other element of each tuple in $Frontier$ is $Cond$. $Cond$ (as IC), contains implications, sometimes also called conditional sentences. The general structure of sentences in IC and $Cond$ is described by the following elementary grammar:

$$\begin{array}{l}
Conditional \ ::= \ Head \leftarrow Body \\
Head \ ::= \ Disjunct^{\vee} \\
Disjunct \ ::= \ Conditional^{\wedge} \\
\quad \mid (Conjunct^{\wedge}) \wedge (Conditional^{\wedge}) \\
\quad \mid Conjunct^{\wedge} \\
Body \ ::= \ Conjunct^{\wedge} \\
Conjunct \ ::= \ Literal \mid \mathbf{false} \mid \mathbf{true} \\
Literal \ ::= \ Atom \mid \neg Atom \qquad\qquad [\mathbf{ICGRAMM}]
\end{array}$$

where $C^{\wedge}$ means either one C or the conjunction of two of more C's; $C^{\vee}$ means either one C or the disjunction of two of more C's; Every identifier in [ICGRAMM] starting with a capital letter is a non-terminating element of the grammar. The symbols ($\wedge$, $\neg$, $\vee$, **false** and **true**) are terminating elements. In addition, [ICGRAMM] must also include rules defining the category

[8] This chapter also presents a formalization of propagation, unfolding, splitting and activation of goals

Atom for a particular application. For instance, above, one might have: $Atom ::= do(up, T)$ ... and so on.

Notice that there are global constraints over all the plans (kept in IC), but also every node maintains (in $Cond$) its own set of local conditional sentences, some of which are obtained by a sort of *partial evaluation* of the global constraints. The reasons for this will be clarified later. However, let us say that the intention is to factor IC out of the nodes, leaving in $Cond$ those constraints that are specific to each plan.

$Uncond$ is a conjunction of literals whose variables are regarded as **existentially** quantified. $Cond$ and IC are conjunctions of conditional sentences like [ICSERVE]. Those variables in IC and $Cond$ that *do not appear* in $Uncond$ are considered **universally** quantified. We will formally describe all these elements in chapter 3 and later in chapter 5. For the moment, an example can illustrate the declarative reading of a node:

Example 2.3.1
$N = ([do(a_1, T')], [cons\_do(a_2, W) \leftarrow \mathbf{true}, do(A, T) \leftarrow c(A, N, T)])$ should be read as $N \equiv \exists T'\ do(a_1, T') \wedge \forall A, N, T, W\ (cons\_do(a_2, W) \wedge do(A, T) \leftarrow c(A, N, T))$.

2.3.2.3 A first look at the functions of the *demo* predicate

This section describes some of the reasoning functions that *demo* (whose formal specification is the subject of chapter 3) performs for the agent.

Some of this functions provided by *demo* are the key to the success of the elevator controller (in particular, the elevator following policy one, as defined in chapter 1). The following is a informal account of those functions:

1. *demo* will (always) consider more recent data from the environment first, to ensure that every goal that must be activated by inputs, is actually activated.

2. *demo* will always process the first node in the $Frontier$ first. Inside the node, the attention is always on the "outmost part" (body) of the first conditional-sentence of $Cond$. For instance, in [ICSERVE] the testing starts with $atfloor(M, T)$ and $on(M, T)$. If all the literals in that part are reduced to **true** the "head" of that sentence will be promoted to a new position in the node, as illustrated above. As we said above, this promotion can be seen as a form of activation of goals.

3. If the head of a sentence is itself an implication, then its promotion means placing it as the new first element of $Cond$. If the head is not an implication but a conjunction of literals, promotion means that it will be transferred to $Uncond$.

   If the head is a disjunction, promotion means the creation of a new node for every disjunct (different to **false**) in that head, while preserving the order of the disjuncts (i.e. first disjunct in the first node and so on).

   Finally, if the head is the symbol **false**, the node must be dropped from the goals.

4. If there are no sentences in the $Cond$ part of the first node of the frontier the program will take a copy of (the whole) IC and will put it as the new $Cond$ of this node. After this processing starts again.

A strong assumption that we should adopt here is that *there is always enough time within every cycle to test all the literals in the body of the conditional sentences or implications.*

A *demo* program with these characteristics will be enough to support a gloria-like agent working as an elevator controller.

2.3.2.4 A first look at an agent cycling: tracing the elevator controller

As an illustration, let us show an imaginary *trace* of that elevator controller at work. The trace shows the evolution of state of the agent as it acts and assimilates observations. For easy reference, we use this notation:

S_i is the overall state at time i.

mental activity is the agent's process that is starting at this stage.

inputs is the record of all the inputs that the agent has received. We use different symbols I, I', I'' to point out the different input sets.

time shows the time indicated by the agent's internal clock.

resource is the amount of resource used in the last call to *demo*. This, of course, is known when the activity *act* is starting to be executed.

goals displays the nodes of the frontier after exiting the mental activity. Notice that the integrity constrains IC are not displayed. IC contains the overall constraints of the agent: [ICSERVE], [ICDO], [ICCONSI] and [ICINCON]. As this information does not change, it is omitted in the trace. Also observe that N_i is the i-th node of the $frontier$. Its content is displayed as: $N_i = \{\ldots\} + \{\ldots\}$. The first set is the list $Uncond$, the second is $Cond$. For simplicity, only the first element of the $frontier$ is displayed.

The trace goes as follows:

S1 =

mental activity	**demo**
inputs	$I : atfloor(3, now), on(5, now)$
time	$T : now$
goals	$N_1 = \{true\} + \{true\}$

At the beginning the elevator is at floor 3 and the term *now* refers to the time at that moment. Let us assume as above that $atfloor(3, now)$ and $on(5, now)$ (the button at floor 5 is on) are known to the agent. The reasoner module (*demo*) is starting to run.

↓

S2 =

mental activity	**act**
inputs	$I : \ldots$
time	$T : now + r_1$
resources	$R : r_1$
goals	$N_1 = \{do(up, T_1), now < T_1\} + \{\ldots\}$

demo has completed its portion of time this cycle, leaving activated the goals of *going up*, from floor 3, presumably to serve floor 5, at some time T_1 after *now*. Observe that, because *demo* took r_1 units of time for reasoning, the new current time is $now + r_1$. Also note that the store of inputs is unchanged w.r.t previous time and that *act* is starting its execution.

↓

S3 =

mental activity	**demo**
inputs	$I' : atfloor(4, now + r_1 + 1), atfloor(3, now), on(5, now)$
time	$T : now + r_1 + 1$
resources	$R' :?$
goals	$N_1 = \{do(up, now + r_1), \ldots\} + \{\ldots\}$

act successfully executed the action of moving *up* at time $now + r_1$. It took *act* 1 unit of time to do so, and then the current time is $now + r_1 + 1$. As part of the feedback, the agent has learnt that it is at floor 4 at time $now + r_1 + 1$. *demo* takes over once again.

$\downarrow$

S4 =

mental activity	**act**
inputs	$I' : \ldots$
time	$T : now + r_1 + 1 + r_2$
resources	$R' : r_2$
goals	$N_1 = \{do(up, T_2), now + r_1 + 1 < T_2,$ $do(up, now + r_1), \ldots\} + \{\ldots\}$

When *demo* suspends its running, time is $now + r_1 + 1 + r + 2$ (*demo* consumed r_2 unit reasoning) and a new goal, similar to the one above, has been activated. The elevator must go *up* again, any time T_2 after $now + r_1 + 1$ which is, of course, already in the past. *Act* re-starts to run.

$\downarrow$

S5 =

mental activity	**demo**
inputs	$I'' : atfloor(5, now + r_1 + 1 + r_2 + 1), atfloor(4, now + r_1),$ $atfloor(3, now), on(5, now)$
time	$T : now + r_1 + 1 + r_2 + 1$
resources	$R'' :?$
goals	$N_1 = \{do(up, now + r_1 + 1 + r_2), \ldots\} + \{\ldots\}$

Once again, *act* succeeded in executing the corresponding action and has learnt its new position. Time is $now + r_1 + 1 + r_2 + 1$ when *demo* starts again reasoning.

$\downarrow$

S6 =

mental activity	**act**
Inputs base	$I'' : \ldots$
time	$T : now + r_1 + 1 + r_2 + 1 + r_3$
resources	$R'' : r_3$
goals	$N_1 = \{do(open, T_3), do(turnoff, T_3),$ $do(close, T_4), \ldots\} + \{\ldots\}$

demo has suspended its processing, this time after activating a more complex set of goals. Two actions in that set are next for parallel execution at time T_3. With that set of instructions, to start serving floor 5, *act* resumes its running at time $now + r_1 + 1 + r_2 + 1 + r_3$.

$\downarrow$

S7 = ...

This trace can be summarized like this: At time *now* the agent, knowing that it is at floor 3, observes that the button at floor 5 is pressed. This input activates, through processing of [ICSERVE] by *demo*, the action "go up". This type of activation happens twice and the execution of these commands takes the elevator to floor 5. Once it gets there, *demo* activates the goals of opening the door and turning off the calling signal.

We are assuming that, in both calls to *demo*, the corresponding r_i (resource or time allocated for computing) is large enough to allow for a complete processing of the goals, as described in the previous section. This, of course, is not always the case. One has to analyse a more complex pattern of interactions to test the flexibility of this architecture. This is done below in chapter 6.

There are many "holes" in this explanation that will be further explored in the following chapters. Critical among them are the operational details of *demo* and of how *demo* manipulates the temporal variables in the representation. The former is the subject of chapter 3. The latter will be considered in chapter 4, where a more systematic approach to program agents of this type is discussed. How to elaborate this models of agent to support more complex and sensible behaviour is also discussed in chapter 4 and in chapter 5. The elevator example is re-taken in chapter 4 and in chapter 6 an actual implementation simulating the elevator is discussed.

To close this chapter, we discuss some limitations of GLORIA that we have identified.

2.3.3 Limitations and shortcomings in GLORIA

There is an important structural limitation and an important simplification in the agent described by GLORIA. Let us discuss first the structural limitation.

2.3.3.1 Thinking and acting

So far, neither GLORIA, nor the previous versions of *cycle* can describe parallelism between "mental activities". As part of the inheritance from the state-based models of change, the *cycle* predicate relies on the **interleaving** of reasoning and acting (i.e. one activity is strictly performed after the other). This, together with a proper set of contraints on the times that *demo* and *act* (specially *demo*) are left to run before passing control to the other "activity", creates the "impression" of parallelism between acting and thinking. However, as any user of a time-sharing operating system (like UNIX) knows, as soon as the "time-slice" of each activity becomes too big, the illusion of parallelism disappears. To prevent that, one has to impose tight limits on the resources (time) allocated to each activity.

The problem with tight resource limits for each activity is that they may conflict with the opennes of the architecture. As we said while formalising GLORIA, the ending time of an action can be regarded as part of the feedback information that the agent receives, and that could not have set before the attempt. The durations of the actions are part of the things unknown before execution. In GLORIA, *act* could only pass control back to the next cycle, after the longest action among the *executables* has terminated. In general, one does not known when that will happen and it is, therefore, difficult and counter-intuitive to set strict constraints on that time.

This problem is even more serious for GLORIA. It is even more counter-intuitive to have an agent (like GLORIA) that can execute several "physical", external actions strictly *at the same time* and yet, it cannot execute those actions and "think" simultaneously (but only pseudo-simultaneously as a time-sharing operating system). If the agent is, as in Allen's example [All91], *holding the lock open* **and** *pulling the door at the same time* (to open a door), then it does not make sense to say that agent *stops thinking* meanwhile.

$$
\begin{aligned}
&cycle(M, T) \\
&\quad \leftarrow\ demo(M, M', T, R) \\
&\quad \wedge\ R\ \leq n \\
&\quad \wedge\ M_1 + M_2 = M' \\
&\quad \wedge\ initiate\_actions(M_1, M_1', T + R) \\
&\quad \wedge\ terminate\_actions(M_2, M_2', T + R + 1) \\
&\quad \wedge\ M_1' + M_2' = M'' \\
&\quad \wedge\ cycle(M'', T + R + 2) \qquad\qquad [\textbf{FUTCYC}]
\end{aligned}
$$

Figure 2.3: A cycle for simultaneous thinking and acting

A solution within the current framework would be to break the *holding-pulling* action into (sufficiently) many small atomic actions. This would be an application specific solution and would require much more intervention by the programmer or designer of the (object-level) description of the problem.

One would like to ascertain that the logical description of the architecture can accommodate this more subtle intuition: that one can think and act *really* simultaneously. In fig. 2.3, we present a simplified proposal[9] that satisfies this intuition.

The important aspects of this proposal are:

1. Two new "activities" substitute *act*: 1) *initiate_actions* which takes actions from a plan, set their initial times, "post" them to the environment and suspends further processing of that plan; and 2) *terminate_actions*, which checks the actions "under execution" for termination at that time and sets the final time for those that do. One, or both activities, could include the checking and assimilation of inputs.

2. We use a "partition" mechanism, represented in the figure by $+$, to split the mental data structure (Goals or Knowledge base or both) into two "mutually exclusive" parts, and later to restore the new parts into a new global data structure. Ensuring that the two parts, presumably containing subgoals from the same overall plan, do not interfere each other, may require access to domain-specific knowledge which could be stored in the mental structures as well. How to do this requires further investigation.

An agent like [FUTCYC] would use *initiate_actions* to "launch" a set of primitive actions (such as $do(hold\_lock\_open, t_o, T_f) \wedge do(pull\_door, t_o, T_f)$), for simultaneous execution as *act*, in the specification of GLORIA, would do.

However, these actions do not need to terminate for *initiate_actions* to pass the locus of control to the other processes. The execution of the actions can continue until some subsequent call to *terminate_actions* (presumably in some future iteration of cycle) terminates it. In particular *demo* can run while the actions are being executed.

Thus, the architecture might accommodate simultaneous reasoning and acting, provided that one can find a sound and efficient "partition mechanism".

2.3.3.2 How to assign resources for reasoning

An important simplification in the agent specified by GLORIA (and Kowalski's *cycle*) is related to the management of reasoning resources. The condition $R \leq n$ in [GLOCYC] (table 2.3.1)

[9]For simplicity we "collapse" all the agent "mental" state (KB and $Goals$, which of course, includes the integrity constraints) into one single term. Thus, M, M' and the corresponding M_i store the agent's mental state at different points in the cycle.

is setting an upper bound for the amount of time that *demo* can consume (i.e. can use for processing goals) in a cycle. Because this bound is set to n, a constant value, *demo* will be allowed to run for at most n unit of times in *every* cycle.

Setting a constant upper-limit for the time for reasoning is a good modelling simplification. It allows for a clear description of the idea of time-sharing and interleaving mental activities in the agent. However, it will be one of the first simplifications to be reviewed before an implementation. The value cannot be kept constant in an implementation of the agent without losing flexibility and performance. If the constant n is too small, the agent will have little chance to process goals and inputs, before a new, probably not very significant, set of inputs arrive. On the other hand, if n is too big, the agent will spend relatively long periods of time reasoning, ignoring events and stimuli in its environment that could be very important. So, a big constant n will affect the reactivity of the agent. These comments apply even if the agent has an extended architecture along the lines of [FUTCYC] in previous section.

The obvious step forward is to make the resource boundary variable, say with $R \leq f(\overline{X})$, where $f(\overline{X})$ is a function of some parameters $\overline{X}$ to the natural numbers. The problem is that very little can be said, in general, about the form of $f(\overline{X})$. This function could, for instance, incorporate an analysis of the physical and psychological state of the agent, so that moods, pains and other factors determine the allocation of thinking time. It could also evaluate the agent's model of the world and history of events and observations, to decide if it is a "good time" to *pace down* and devote more time for reasoning. It could even take into account preferences and *desires* so that, as part of the response to certain stimuli, the agent "decides" (or it is imposed) to think less and observe more. Another possibility that we suggested above is to use the boundary (n or $f(\overline{X})$) to indicate the ocurrence of some external events (e.g. the arrival of a certain input), that demands the "interruption" of reasoning. Most of this considerations are specific for particular realizations and implementations of agents. For this reason, we do not discuss "resource allocation functions" (functions like $f(\overline{X})$) or interruptions anymore in this thesis.

2.4 Conclusion

In this chapter we have described first, how to model an agent as a logic program, using an approach that is useful in general to model processes that change objects. We started with a formalisation due to Kowalski [Kow95] and extended it so that one has an agent's description more suitable for automatic reasoning and also a formalization of an agent capable of performing parallel actions. We then showed one way of incorporating knowledge into the agent, so that it knows how to provide services like those required in an elevator. Using this benchmark example, we showed a simple trace of the execution of the agent, to illustrate the interleaving of acting, reasoning and observing.

In this chapter we also sketched, very roughly, some of the characteristics of the reasoning mechanism required by the agent, which will be implemented as the *demo* predicate. We already said, for instance, that the execution of *demo* is resource bounded and that *demo* will transform goals into sub-goals, using the information obtained from the environment and from its knowledge base. We also illustrated how some of the agent's goals processed by *demo* have the form of $condition \rightarrow action$ rules or implications, and we said that *demo* is responsible for obtaining unconditional goals from the implications in the process of "activation of goals".

In the following chapter, we describe a general-purpose proof procedure for abductive logic programs. The key contribution of that chapter is that the operational requirements for *demo* (those mentioned above and others) are "captured" by the logical transformations specified by the proof procedure, including a logical account of the principles of "activation of goals".

Chapter 3

The Agent's Abductive Reasoning Mechanism

3.1 What is abduction?

Abduction is a defeasible, non-valid form of reasoning. The philosopher Charles Sanders Peirce first characterized it, together with the other known forms of reasoning, as follows [Pei55]:

Deduction [is] an analytic process based on the application of general rules to particular cases, with the inference of a result.

Induction [is] synthetic reasoning which infers the rule from the case and the result.

Abduction [is] another form of synthetic inference, but of the case from a rule and a result.

Abduction is not a valid rule of inference because *it is not the case* that, for instance, from: $h \leftarrow (b \vee c)$ and h one can validly infer $(b \vee c)$. It all depends on whether $h \leftarrow (b \vee c)$ is a *complete* account of the knowledge about h. Only then one can assume to know $h \rightarrow b \vee c$ as well. However, for a well-informed agent embedded in a not-so-chaotic environment, abduction can be an useful tool for effective decision-making.

As Console et. al. explain in [CDT91], the assumption of *complete* knowledge does not have to imply the *permanent closure* of the information (related to h, in this case), "but simply that one is reasoning to the best of the given knowledge" (.ibid). If new information arrives, one should be able to relax the assumption and to review one's knowledge base, perhaps withdrawing conclusions reached with the previous knowledge base. Thus, abduction can be regarded as a form of defeasible, non-monotonic reasoning. A process of decision-making that appeals to abduction would then be subject to inherent uncertainty.

As has been explained by Reiter in [Rei96], abduction is a *meta-level* task. It is at the *meta-level* that the *abducible* character of the explanations (b or c, above) is established. Notice that this is similar to the treatment of negation as *negation as failure* (NAF [Cla78]) where *that which can not be proven* is assumed to be **false**. As in NAF, the *if-and-only-if* rules obtained from, for instance, the **completion** [Cla78] of the knowledge base ($h \leftrightarrow b \vee c$, in our example) provides a way to reduce that "meta-level task" to pure, object-level, deductive reasoning. It is doing *abduction through deduction* [Fun96].

A substantial effort has been made to formalize abductive reasoning. Poole's Theorist [Poo89] was the first to incorporate the use of abduction for non-monotonic reasoning. Eshghi

and Kowalski [EK89] have exploited the similarities between abduction and negation as failure and provided a proof procedure based on a transformation of logic programs with negation into logic programs with *abducible atoms*. de Kleer incorporates abduction into the so-called truth maintenance systems to obtain the ATMS [dK86]. Also, in [CDT91], L. Console, D. Theiseider and P. Torasso analyse the relationships between abduction and deduction and define what they call an **abduction problem** as a pair $< \mathcal{T}, \phi >$ where:

1. T (the domain theory) is a hierarchical propositional logic program[1] whose abducible atoms are the ones not occurring in the head of any clause.

2. ϕ (the observations to be explained) is a consistent conjunction of literals with no occurrence of abducible atoms.

This definition resembles Poole's definition of the *Theorist Framework*, and both of them can be mapped into our **Definition 1** in chapter 1. The idea of imposing these structures and distinctions upon a reasoning problem is to create **frameworks** in which the semantics of each component and its relationships with other components can be established in a declarative manner. A framework is a structure that distinguishes between types of elements in a formalisation. For instance, the framework $< T, \phi, Ab >$ could be used to say that one has a theory T, a set of of observations ϕ and that these observations can be explained by abducing predicates in T whose names appear in Ab (abducible predicates). These distinctions are then used to justify differential treatment of each type of element. In the cases here considered, for instance, abducible predicates and non-abducible predicates, so separated by the framework, are processed differently. The distinction captures the fact that the former, unlike the latter, denote uncertain information.

The use of *frameworks* has been taken further by Kakas and Mancarella [KM90], Denecker and De Schreye [DDS95], Toni [Ton95], Fung [Fun96] and more recently, Wetzel *et al* [WKT95], [Wet97] in the context of incorporating abduction into constraint logic programming. In [KKT93] there is an overview of the first efforts to incorporate abduction into logic programs. In [FK96] there is a preliminary description of the abductive framework that we have used to formalize the agent reasoning mechanism. The following sections will clarify the reasons for this selection.

3.2 Abduction for planning

Abduction has been applied to a variety of problems (see [CM85] for a general description of applications). When abduction is applied to planning, normally "abducibles" are actions that may be performed to achieve a given goal. So, for instance, given a goal g and the rule $g \leftarrow a_1 \wedge \ldots \wedge a_n$, the plan $a_1 \wedge \ldots \wedge a_n$ may be abduced, provided that every a_i is regarded as an abducible. We use this and other rule-forms where the strategy of reducing goals to abducible sub-goals is also employed.

Planning by abduction is an increasingly popular technique in logic-based systems. It has been used in Allen's temporal logic [All91] and several times in logic programming systems (see [Esh88b], [Sha89], [Eva89], [MBD95] and [Poo95]). Most of these systems use the Event Calculus or some event-based formalism to deal with the temporal reasoning required by the planning problem. According to Reiter [Rei96], there is no alternative but to use an "abductive account" of reasoning when an event-based temporal logic is involved. However, as we have suggested in the previous section and will show later in this chapter, an abductive, computational

[1] A hierarchical logic program is a logic program without recursive rules.

framework may well be based on a deductive account of reasoning. We discuss the advantages and disadvantages of the Event Calculus approach in chapter 4. For easy reference, however, let us summarize the attributes of the abductive, event-based approach that makes it attractive for planning applications:

1. The history of planning systems in AI (see [RN95] and the review in chapter 6) shows a shift from systems and algorithms that produce **complete**, perfectly ordered and well defined sequences of steps as plans, to systems that do **partial planning**, where plans are only partially described leaving the *executive* to choose a particular set and ordering of actions at runtime[2]. This **least commitment strategy** [RN95] can be achieved by means of an abductive, event-oriented system that keeps the *planning problem* open for updates at execution time (as we show below).

2. Conceptually speaking, "Meta-level abstractions" are desirable. Meta-level descriptions can be amalgamated with object-level ones to obtain a highly expressive specification. Notions like "agent's mental activities" can be easily captured by that amalgamation, as we have shown in chapter 2. It has been argued that, intuitively, planning is a meta-level (abductive) activity and hence, it should be represented as such[Sha93].

3. An event-based framework lends itself to planning by: traditional goal *regression*, where the next action to be performed by the agent is determined last for execution (having identified all the actions back from the goal); or by *progression*, where the next action to be performed by the agent is determined first for execution (and where a complete identification of all the actions is not required, in principle). The system can then be tailored to a particular planning problem. We argue that in the case of reactive planning, a planner based on progression is more appropriate. This is discussed in chapter 4 (also, recall the discussion about reactivity in chapter 1).

4. Finally, because the "time line" need not be explicitly represented, as in the Situation Calculus, an event-based planner can be naturally implemented as an *any-time algorithm*. This type of algorithm can be interrupted at any stage of processing (to be interleaved with other processes) and, when provided with more resources, will produce better (refined) plans (as shown below).

It has been traditionally understood that abduction involves a test of consistency of the generated explanations "with respect to the object level axiomatization of the domain"[Rei96]. This normally implies having a "meta-level" theorem prover that 1) identifies the abductive explanations[3], 2) builds a theory consisting of the original axiomatization plus the set of explanations and 3) checks that the new theory is consistent.

If those explanations are written as arbitrary sentences in first order logic, that test would be equivalent to a test of consistency of a set of sentences in FOL which is, in the general case, not even *semi-decidable* (see discussion in [RN95]). Therefore, that test is *non-computable*

[2]There is an confusion of terminology in the planning community in AI. Partial planners are misleadingly called *non-linear planners* by some researchers to contrast them with the *linear*, complete planner of the first generations. The label *non-linear* is misleading because it suggests that the process of generating the plans is non-linear, whereas the intention is to say that the product of the process: a plan, is non-linear, meaning, in turn, that the ordering of the time-points or intervals in the plan is not explicitly defined. Similar objections apply to the label *linear*, of course. We use [RN95] terminology that is closer to usage in Knowledge Representation.

[3]This identification is only possible at the meta-level, i.e. using a meta-language to talk about the object level axiomatization. That is the reason one have to refer to a "meta-level" prover. However, one could say that the other two steps also require "meta-level" capabilities in the prover, to "talk about" a theory being built and its consistency being checked. Probably, that is what Reiter means when he mentions a "meta-level test of consistency" [Rei96].

in the general case. However, one can impose restrictions on the form of the theories and of the "abduced explanations" to avoid the general case. For instance, by assuming that the original theory is consistent, one can concentrate the checking of consistency on the abduced explanations. This kind of "sub-test of consistency" can be shown to be computable for certain form of "abducibles" (see [Mil96] for an analysis of these topics in the context of the Event Calculus).

Consistency tests are carried out to ensure the integrity of the explanations. There are, however, many ways of guaranteeing the integrity of a theory. The designer of the axiomatization can state *integrity constraints*, sentences describing conditions with which the information in a theory (knowledge base) must be consistent. This is known [EK88] as the *consistency view* of integrity constraints, which requires the theory consisting of the original axiomatization, the explanations and the integrity constraints, to be consistent. In principle, integrity constraints do not add anything but complexity to the consistency test problem (more sentences to be tested). However, if one assumes that the original axiomatization is internally consistent then the test of consistency could involve only the explanations and the integrity constraints.

Alternatively, integrity contraints can be interpreted as goals to be "achieved" by the system. This is the *theoremhood* interpretation (of integrity constraints) [EK88], which requires the integrity constraints to be entailed by the new theory (axiomatization plus explanations).

The *theoremhood* interpretation is, in general, "stronger" (i.e. more restrictive) than the *consistency* view, although the two views coincide for theories of the form of the if and only if completion of hierarchical logic programs [FK96]. The intesting point is that *theoremhood* is always semi-decidable and can be used 1) to prevent incorrect plans (as those mentioned by Pelavin in [Pel91], that arise in the context of planning with concurrent actions) and 2) to devote all computations to goal-entailment proofs, thereby cancelling the need for a separate, consistency-testing procedure.

For the **iff** proof procedure (**iff**PP hereafter), introduced below, the theoremhood view of integrity contraints has been adopted. The description of an algorithm for this abductive proof procedure is the subject of the rest of this chapter and of chapter 6 where the algorithm is adapted to be used as a planner. In this chapter, we introduce the specification of **iff**PP which constitutes the kernel of the planner.

A remark about programs and specifications is in place here. Normally, specifications are logical descriptions that guide the implementation of programs. In logic programming, however, there is the possibility that "*logic programs can just as well be regarded as executable specifications*"[Kow84]. This is related to the fact that logic programs have a declarative reading in addition to the procedural reading that any program has. So, a logic program can bridge the gap between a specification and its implementation. In this chapter, for instance, we show *an implementation* of the inferences rules of the proof procedure as a logic program (the *demo* predicate). However, this *implementation* can be seen as the specification for another implementation (in non-pure PROLOG, in any other lower-level programming language or even in hardware).

3.3 Preliminaries for the iff Proof Procedure

3.3.1 Abductive logic programs, queries and semantics

3.3.1.1 What is an Abductive Logic Program?

In [FK96] Fung and Kowalski define an **abductive logic program** as follows[4]:

[4]For basic notions like **atom**, **literal** and **if-and-only-if form** we follow the common usage in logic programming, as explained in [Kow79b] and [Hog90].

Definition 2 *An abductive logic program is an tuple* $< T, IC, Ab >$ *where*

1. *T is a set of* **definitions** *in if-and-only-if form:*

$$p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \ldots \vee D_n \quad k, n \geq 0 \tag{3.1}$$

where p is a **defined** *predicate symbol different from equality (=) and from any predicate symbol in Ab, the variables* $X_1, \ldots, X_k$ *are all distinct, and each* D_i *is a conjunction of literals. When* $n = 0$*, the disjunction is equivalent to* **false**. *There is one definition per predicate symbol p.*

2. *IC (the set of integrity contraints) is a consistent set of implications [of the form]:*

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n \quad m, n \geq 0 \tag{3.2}$$

where each A_i *and* B_i *is an atom. When* $m = 0$*, the disjunction is equivalent to* **false**. *When* $n = 0$*, the conjunction is equivalent to* **true**.

3. *Ab is the set of all predicate symbols, called* **abducibles**, *different from* $=$ *and any predicate symbol defined in T.*

The atom $p(X_1, \ldots, X_k)$*, defined in T above is said to be the head of the definition. The variables* $X_1, \ldots, X_k$ *are implicitly universally quantified, with scope the entire definition. Any variable in a disjunct* D_i *of a definition, which is not one of the* $X_1, \ldots, X_k$ *is implicitly existentially quantified, with scope the disjunct.*

Fung [Fun96] also adds the restriction that definitions must be "range restricted", i.e. every variable occurring somewhere in a D_i in a definition must occur in a non-negative literal within the same D_i or in the head of the definition.

With respect to [FK96], we can refine the constraint that "all variables in an integrity contraint are implicitly universally quantified" by allowing explicit quantification in the heads of implications, as exemplified by the integrity contraints in section 2.3.2 in chapter 2. Thus, in the general case, variables in implications are universally quantified, unless otherwise stated. This means, of course, that the language must be extended with the existential symbol ($\exists$) and also with the universal symbol ($\forall$). Implicit quantifiers are always "outside" the formulae (i.e. the formula $\exists X F(\ldots, Y_1, \ldots, Y_n, X, \ldots)$ should be read as $\forall Y_1, \ldots, \forall Y_n, \exists X, F(\ldots, Y_1, \ldots, Y_n, X, \ldots)$, where $Y_1, \ldots, Y_n$ are all the universally quantified variables in the formula). We return to this discussion in chapter 5 as part of the presentation of the ACTILOG language.

3.3.1.2 What is a Query?

To *extract* information from a logic program, a user can submit a query to the proof procedure operating on the program. In logic programming, a query can be seen as a **call to procedures** defined in a logic program. A query can provide *input data* to those procedures and also collect *output data* in its variables. Declaratively, however, a query is a logical sentence, defined here (as in [FK96]) as follows:

Definition 3 *A* **query** *is a formula of the form:*

$$B_1 \wedge \ldots \wedge B_n \quad m, n \geq 0 \tag{3.3}$$

where each B_i *is a* **literal**, *i.e. an atom or the negation of an atom. All variables in the query are regarded as existentially quantified (No other form of quantification is allowed).*

To improve the efficiency of the proof procedure, variables in the query can be **marked** to distinguish them from other existential variables that may appear later in the computation to answer the query. This distinction is required in order to extract answers from the frontier and is further clarified below.

In the planning context, a query is a conjunction of goals whose joint *achievability* the proof procedure tries to prove. The information to prove that the goals are achievable is normally extracted from the knowledge base. The abductive procedure, however, allows for another source of *contextual information*, particularly suited to the reactive planning problem because it can be updated at anytime. This new source of information is constituted, in principle, by the set of *abducibles* or assumptions made up to a point, that can be used to justified or to prevents further assumptions. We expand on this argument in chapter 4.

3.3.1.3 What is the semantics of an Abductive Logic Program?

To define the semantics of a logic program, one needs to establish what it is that the proof procedure **computes** from the program given a query, i.e. what an answer is.

Definition 4 *Given an abductive logic program,* $< T, IC, Ab >$*, an* **answer** *to a query* Q *is a pair* (Δ, θ) *where* Δ *is a finite set of ground abducible atoms and* θ *is a substitution of ground terms for the variables in* Q*, that satisfies:*

$$T \cup Comp(\Delta) \models Q\theta \tag{3.4}$$

and

$$T \cup Comp(\Delta) \models Cond\theta \tag{3.5}$$

for every conditional sentence $Cond$ *in* IC

$Comp(\Delta)$ refers to the Clark completion [Cla78] of Δ. Completing a theory C in Clark's sense, technically means strengthening all the *if* definitions in C to *if-and-only-if* definitions, negating any undefined atom in C, and adding the Clark equality theory (CET) that defines the necessary and sufficient conditions for syntactic equality between terms. An important consequence of the completion of Δ is, therefore, that abducible predicates which do not occur in Δ are defined as equivalent to **false**. Clark completion is a common form of *closed world assumption* in logic programming. It was designed to provide a semantics for logic programs extended with **negation as failure**, i.e. *normal logic programs*.

What completion does is to transform a logic program with negation as failure, into a first order logical theory with classical negation. This dissolves the semantical problem because, the resulting theory can be understood model theoretically as is traditional in first order logic. This happens in most cases. However, there are *pathological* cases where completion causes problems. A well-known example is the program $p \leftarrow \neg p$, transformed by completion into $p \leftrightarrow \neg p$, which is obviously inconsistent.

Completion semantics does not allow either "for the possibility that the truth value of a query [..] be **undefined** because the [prover] interpreter fails to halt" [Kun87]. To formalise the **iff** proof procedure, Fung and Kowalski [FK96] have used a semantics described by Kunen in [Kun87] and also used by Denecker and De Schreye in their formalisation of SLDNFA [DDS92], [DDS95]. Kunen's semantics is a refinement of Fitting's three-valued semantics for logic programs [Fit85] and they are both based on Kleene's idea above, (first suggested in 1938 [Kle38], [Kle52]), of using three-valued logic to deal with nonterminating computations. In his PhD. thesis [Fun96], Fung has proved that the **iff** proof procedure is sound and complete with respect to Kunen's three-valued (**t**rue, **f**alse and **u**ndefined) semantics. So, $\models$ above (in 3.4 and 3.5) should be read as truth in all three-valued models of that kind.

The following section describes Fung and Kowalski's **iff proof procedure**.

3.3.2 Fung and Kowalki's iff proof procedure

3.3.2.1 Derivations and Frontiers

The **iff** proof procedure is a rewriting method for logical formulae. It consists of a set of **inference rules**, each of which replaces a formula by another formula that is equivalent to the former in the context of the theory and the integrity contraints. One can characterize the rewriting process by means of *derivations* [FK96].

Definition 5 *A* **derivation** *of a formula F_n from a formula F_1 is a nonempty sequence of formulae $F_1, \ldots, F_n$ such that each F_{i+1} in the sequence is obtained from the previous F_i after applying one of the inference rules. A derivation then, satisfies:*

$$T \cup CET \cup IC \models F_1 \leftrightarrow F_n \tag{3.6}$$

Definition 6 *A* **frontier** *in a derivation is any formula F_i within the sequence that defines the derivation. The general form of every F_i is a disjunction:*

$$F_i \equiv D_1 \vee \ldots \vee D_n \quad n \geq 0 \tag{3.7}$$

which in the case $n = 0$ implies $F_i \equiv$ **false**. *There is also the degenerate frontier when $n = 1$, for which $F_i \equiv D_1$ holds.*

Definition 7 *A* **node** *is any formula D_j in a frontier F_i. The general form of a node is a conjunction such that:*

$$D_j \equiv C_1 \wedge \ldots \wedge C_n \quad m \geq 0 \tag{3.8}$$

and whenever $m = 0$ the conjunction is equivalent to **true**. *Every conjunct C_k can be either* an atom, a disjunction of conjunctions, or an implication as in equation 3.2.

Definition 8 *An* **implication** *is any formula of the form:*

$$A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_r \quad m, r \geq 0 \tag{3.9}$$

where $A_1 \vee \ldots \vee A_m$ is known as the **head** *of the implication and $B_1 \wedge \ldots \wedge B_n$ is its* **body**. *If $m = 0$ the head is* **false**. *If $r = 0$ the body is* **true**.

In the planning context, every F_i can be used to represent the set of alternative courses of action (plans) the agent has.

As discussed at the beginning of the chapter, one can allow for a more general form of implications in C_k (as described by [ICGRAMM] in chapter 2). Neither this extension, nor the use of explicit existential quantification suggested above, alter the soundness or completeness of the proof procedure with respect to Kunen's semantics. For the sake of simplicity, however, we maintain Fung's syntax in the presentation of the proof procedure. We consider the extensions in the special version of the procedure discussed in chapters 5 and 6.

3.3.2.2 The form of queries

As explained at the beginning of this section (equation 3.3), a query is a conjunction of positive and negative atoms.

When a query Q is submitted, the proof procedure starts by transforming it into its **initial frontier** F_o. This initial frontier is a degenerate disjunction i.e. just one disjunct formed, in turn, by the conjunction of:

1. the positive atoms in Q.

2. an implication with the form $\mathbf{false} \leftarrow Q$ for every $\neg A$ in Q.

3. the implications in IC.

This initial transformation is not required if one assumes that instead of being queried from time to time, the system is *permanently* deriving a new F_{i+1} from a previous F_i. The proof procedure goes from a *frontier of derivations* F_i to a new one (F_{i+1}).

In the following section, we describe the inferences rules including its operational details. Before that, however, let us extend the data structures introduced in section 2.3.2, chapter 2 to store frontiers of nodes.

Definition 9 *A frontier of nodes F is implemented by an or-list. Every Node N in the frontier is implemented by an structure such as:*

$$(\Delta, UC, CN, HF, M) \tag{3.10}$$

where Δ contains abducibles atoms only, UC contains unconditional goals still to be processed, CN is a conjunction of implications including those in IC. Each implication CN_i in CN has a history of propagation HP_i attached to it. HF above is the history of factoring for atoms in this node. M is the list of "marked variables", i.e. the list of existentially quantified variables in the initial frontier.

M has an important role to play, as illustrated by the following revisiting of the example 2.1 in [Fun96]:

Example 3.3.1 The query $Q \equiv \exists X \exists Y(p(X) \wedge \neg q(Y))$ is represented by the node:

$$(\{\}, \{p(X)\}, \{\mathbf{false} \leftarrow q(Y)\}, \{\}, \{X, Y\}) \tag{3.11}$$

Notice that:

$$(\{\}, \{p(X)\}\{\mathbf{false} \leftarrow q(Y)\}, \{\}, \{X\}) \tag{3.12}$$

would be representing $Q' \equiv \exists X, \forall Y(p(X) \wedge \neg q(Y))$ instead.

The meaning and use of each one of these elements will be explained by the description of the inference rules and clarified by the algorithms presented in the following sections.

3.3.2.3 The inference rules

The inference rules of the **iff** proof procedure are explained by Fung in [Fun96] and by Fung and Kowalski in [FK96]). The following description is intended as a introduction to the logic programs that implement the proof procedure, presented and discussed in the following sections. Notice, however, that some important details for the implementation of the rules are discussed in this section. The inference rules are[5]:

- **Unfolding** creates a new node by replacing an atom in UC or in the body of some implication in CN by its definition in T. Consequently, one type of unfolding is **unfolding in UC**:

[5] An observation about notation. We are using X, T, etc. in capital letters as "meta-variables" here. They can be regarded as placer holders for variables or terms in the object language. So, when we say $\overline{X}$, this is a vector of X_i's, each of which can be a variable or a term. Also, $\overline{X} = \overline{T}$ is an abbreviation for the conjunction of atoms $X_i = T_i$ built from elements of $\overline{X}$ and $\overline{T}$, respectively.

1. select an atom $p(\overline{T})$ from UC in a node N.
2. obtain the definition $p(\overline{X}) \leftrightarrow D_1 \vee \ldots \vee D_n$ from the knowledge base.
3. build a new node N' identical to N except for having $p(\overline{T})$ replaced with $(D_1 \vee \ldots \vee D_n)\theta$, where θ is the substitution $\{X_1/T_1, \ldots, X_m/T_m\}$, where the X_i and T_i are the elements in $\overline{X}$ and $\overline{T}$, respectively.

The other type of unfolding is **unfolding in CN**:

1. select an atom $p(\overline{T})$ from the body of an implication $H \leftarrow p(\overline{T}) \wedge Rest$, in CN (in a node N).
2. obtain the definition $p(\overline{X}) \leftrightarrow D_1 \vee \ldots \vee D_n$ from the knowledge base.
3. build a new node N' identical to N except for having that implication in CN replaced with the conjunction: $(H \leftarrow \overline{X} = \overline{T} \wedge D_1 \wedge Rest) \wedge \ldots \wedge (H \leftarrow \overline{X} = \overline{T} \wedge D_n \wedge Rest)$.

- **propagation** resolves an atom in Δ with an implication in CN of a node N, to create a new node N' that has an additional implication in its CN. More precisely, the rule says:

1. select an atom $p(\overline{T})$ from UC and an implication CN_i of the form $(H \leftarrow p(\overline{X}) \wedge Rest)$ in N.
2. **if** $p(\overline{T})$ has not been previously *propagated* with $p(\overline{X})$ **then** build a new node N' identical to N except that the implication $(H \leftarrow \overline{X} = \overline{T} \wedge Rest)$ is added to CN. Also, add $p(\overline{T})$ to the history of propagation of $p(\overline{X})$, HP_i, attached to the original CN_i'.

The test in the second step verifies that the atom has not been used previously to resolve against the same atom in the implication. This is the reason to maintain a *history of propagation*: to ensure that atoms are only used once to be propagated with a particular implication. Otherwise, loops could easily occur. One could avoid having a history of propagation by doing exhaustive propagation of all suitable atoms $p(\overline{T})$ in UC through each $p(\overline{X})$. But this is only possible when one has all these $p(\overline{T})$ when propagation is applied to an implication. In an anytime implementation, where the prover may have "new" atoms in UC on reentering, that would not be the case, as we show below.

propagation is conceptually attractive as an inference rule for the agent reasoning mechanism because it supports, (together with the second type of unfolding), the mechanism of **activation of goals**. A goal in the head of an implication can be seen as activated by atoms in Δ that are resolved against atoms in the body of the implication. Eventually, the body of the implication is reduced to **true** and the goal in the *head* can be promoted to UC as an unconditional goal, as the following example illustrates:

Example 3.3.2 Suppose one starts with the formula:

$$o(1) \wedge \forall N(s(N) \leftarrow o(N)) \tag{3.13}$$

propagation will transform it into:

$$o(1) \wedge \forall N'(s(N') \leftarrow N' = 1) \wedge \forall N(s(N) \leftarrow o(N)) \tag{3.14}$$

which, after rewriting yields:

$$o(1) \wedge s(1) \wedge \forall N(s(N) \leftarrow o(N)) \tag{3.15}$$

Thus, one ends up with a new unconditional goal to be proven.

Notice that the history of propagation is crucial to ensure that the prover will not loop. Without a record of that type in the example above, $o(1)$ could be used again to "trigger" the implication and activate the goal $s(1)$.

The history of propagation has the same structure and content type as the history of factoring discussed below. In the general case, it is a list of equalities, refering the assignments required to unify the abducible in Δ ($o(1)$ above), with the abducible in the implication ($o(N)$). So, after the operation illustrated in example 3.3.2, the history of propagation of the implication will probably[6] contain the equality $N = 1$.

When abducibles are ground atoms, one can dispense with the need to maintain or update the history of propagation. For that, one must adopt a slightly different version of the propagation rule which do not add a new implication to the node, but replaces the original one. The following example 3.3.3 illustrates this.

Example 3.3.3 Given the node: $p \wedge\ q \leftarrow p$, where p is an abducible, one can:

- either, derive the node: $p \wedge q \wedge q \leftarrow p$, with the history of propagation of $q \leftarrow p$ registering the p has been "propagated" (and, therefore, cannot be propagated again).
- or, derive the node: $p \wedge q$, in which the implication has been removed (and, obviously, cannot be used for propagation anymore).

Another important detail of the propagation rule is about the treatment of existentially quantified variables in the implications. Variables of this kind must be renamed at propagation, according to the following criterion:

> When the propagation rules is being applied to an implication I, every existentially quantified variable X appearing in I must be renamed **if and only if** the propagating abducibles contain universally quantified variables on which X depends (i.e. X is within the scope of any of this variables in I).

The following examples clarify the criterion:

Example 3.3.4 Consider the node:

$$o(1) \wedge \forall N\ \exists T(s(N,T) \leftarrow o(N)) \tag{3.16}$$

When propagating $o(1)$ through $o(N)$, one must obtain:

$$o(1) \wedge \exists T'(s(1,T')) \wedge \forall N\ \exists T(s(N,T) \leftarrow o(N)) \tag{3.17}$$

where the variable T' is introduce as a renaming of T. If this were not done, one would obtain:

$$o(1) \wedge \exists T(s(1,T)) \wedge \forall N\ \exists T(s(N,T) \leftarrow o(N)) \tag{3.18}$$

where a posterior assignment of, say, $T = 0$, would allow the rewrite rules (explained below) to produce an incorrect rendering of the implication, such as:

$$o(1) \wedge s(1,0) \wedge \forall N\ (s(N,0) \leftarrow o(N)) \tag{3.19}$$

which, clearly, is not the intended meaning of the implication.

[6] These lists can have one of several possible structures, with different implications for the implementation. These possibilities are discussed below for the historical list required by the factoring rule, but the discussion equally apply to the history of propagation.

However, renaming of existentially quantified variables is not always required, as this example shows:

Example 3.3.5 Consider another node:

$$o(1) \wedge \exists T \forall N (s(N,T) \leftarrow o(N)) \tag{3.20}$$

In this case, renaming of T as, say T' would be incorrect, because it would yield:

$$o(1) \wedge \exists T' (s(1,T')) \wedge \exists T \forall N (s(N,T) \leftarrow o(N)) \tag{3.21}$$

from which the system will be unable to verify whether or not *it is the same T for all the N's.*

The criterion above covers cases with a more complex combination of quantifiers, as in:

Example 3.3.6

$$o(1) \wedge r(0) \wedge \forall N \ \exists T \ \forall M (s(N,T,M) \leftarrow o(N) \wedge r(M)) \tag{3.22}$$

where renaming is done when one propagates $o(1)$ (because T is within the scope of N), but not when one propagates $r(0)$ (because, T is not within the scope of M).

The treatment of existentially quantified variables makes propagation a complicate operation. One can avoid these complications by restricting the language so that, for instance, universally quantified variables are not allowed in implications or are restricted to appear in particular positions in the implication.

- **splitting** distributes conjunctions over disjunctions. This rule has two variants: **simple splitting** and **splitting a head**. *simple splitting* is the usual distribution of conjunctions over disjunctions applied to the elements of UC. Whenever one has $UC \equiv (E_1 \vee \ldots \vee E_k) \wedge RestUC$, one replaces the node containing this UC with k nodes, one for each E_i. Every new node has an UC' such that $UC' \equiv E_i \wedge RestUC$ for the E_i corresponding to that node. Recall that the E_i's are obtained by unfolding of "range restricted" definitions. This ensures that no E_i contains universally quantified variables.

  On the other hand, in *splitting a head* (in node N), an implication of the form $(H \vee RestH \leftarrow Body)$, where H is an atom with *no universally quantified variables*, is selected and two nodes, N_1 and N_2, replace N. N_1 is identical to N, except it has H added to its UC, and the original implication disappears from its CN. N_2 is also identical to N, except for the replacement of the original implication by $(RestH \leftarrow Body)$

  Observe that whereas *simple splitting* exhaustively distributes the rest of UC over the disjuncts of D, the second form of splitting takes only one element of the head apart, creating two nodes.

  There is a relation between splitting and unfolding and the notion of equivalence of logical sentences. For instance:

$$(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C) \tag{3.23}$$

$$(H \leftarrow (A \vee B) \wedge C) \equiv (H \leftarrow A \wedge C) \wedge (H \leftarrow B \wedge C) \tag{3.24}$$

formalise splitting (3.23) and unfolding in CN (3.24), respectively. However, they can also be seen as "defining" the relation $\equiv$ for sentences with these forms. This approach has been used by Kowalski in [Kow95] to specify logical equivalence between formulae and also to describe control strategies for a proof procedure.

We use a similar device in the implementation of the proof procedure. That is the reason why *splitting* is not mentioned in any of the logic programs shown below but is assumed to be "embedded" in the definition (and implementation) of the predicate $\equiv$.

- **case analysis** for an equality, $X = T$, creates two nodes to replace the one which contains the equality. One node for the case when $X = T$ and the other for $X \neq T$. The application of this rule is restricted to equalities appearing in the body of an implication. This rule is useful to extract information from implications. The rule is applied as follows:

  1. select an implication of the form $(H \leftarrow X = T \wedge Rest)$ in N.
  2. replace N with nodes N_1 and N_2. N_1 is identical to N, except it has $X = T$ added to UC, and the implication above is replaced in CN by $(H \leftarrow Rest)$. N_2 is also identical to N, except for the replacement of the implication by $(\mathbf{false} \leftarrow X = T)$

  Observe that there are restrictions on H and $Rest$ in the selected implication (H cannot be **false** and $Rest$ **true** at the same time). In other words, one cannot select a disequality for case analysis.

- **factoring** considers two abducible atoms (with the same predicate) in $\Delta \cup UC$ in a node, and creates two new nodes to replace this node; one for the case when the abducibles are identical, and the other for case when they are different.

  For ground atoms, one can decide whether the two atoms are or are not the same immediately, without having to create two nodes for the alternatives. This is also the case when the atoms involved are identical up to variable identifiers (e.g. $p(X) \wedge p(X)$ can only be transformed into $p(X)$ disregarding the actual value of X).

  In general, factoring matches the arguments in both atoms to find out whether these atoms can be considered identical. If the arguments cannot be matched (e.g in $a(1) \wedge a(2)$), factoring must fail. It must also fail, if the atoms involved were used for factoring earlier on. To support this, a record of which pairs of arguments have been considered is kept in the *history of factoring* (HF) of each node. As with the history of propagation, this record, HF, is kept to avoid repeatedly performing the same operation. And as with propagation, one cannot by-pass the history of factoring by doing exhaustive factoring because, in the context of an open architecture, no all "factors" are known at any given point in time. Factoring is done as follows:

  1. Consider two abducible atoms in a node N, $p(\overline{T_1})$ and $p(\overline{T_2})$. (The algorithm below shows how one abducible, $p(\overline{T_1})$, is taken from Δ and other, $p(\overline{T_2})$, from UC).
  2. **if** these atoms have not been jointly considered before and they arguments match **then** replace N with two nodes N_1 and N_2 such that:
     (a) N_1 is identical to N except that $p(\overline{T_2})$ is not in N_1's UC and the equalities in $\overline{T_1} = \overline{T_2}$ are added to Δ.
     (b) N_2 is identical to N, except the set of disequalities $\overline{T_1} \neq \overline{T_2})$ is added to history of factoring HF of the N_2, to prevent more factoring of atoms involving these terms and variables. Alternatively, these disequalities could be added to CN as new implications of the form $\mathbf{false} \leftarrow T_{1i} = T_{2i}$. This would yield a declarative

reading of the history of factoring. But it would be more complex to use the whole CN to test for previous factoring. On the other hand, one could use disequalities produced by other rules.

The addition of $\overline{T_1} \neq \overline{T_2}$ to HF[7] implies some extra processing. For instance, if $\overline{T_1} = \{W, a, f(Z), g(a), h(T)\}$ and $\overline{T_2} = \{c, X, d, g(R), t(a)\}$ [8] then, strictly speaking, one should add $(W \neq c)$, $(X \neq a)$ and $(R \neq a)$ to HF. The others do not add new information about the variables, in which case it is neither necessary nor useful to keep them. With respect to $g(R) \neq g(a)$, this is not necessary provided that the mechanism for matching (used in step 1) always reduces the arguments to their innermost elements.

This extra processing can cause overload because it involves, among other things, a variant of the *occur-check* of the unification algorithm (If $\overline{T_1} = \{f(f(Y))\}$ and $\overline{T_2} = \{Y\}$, then the matching program will have to detect that $f(f(Y))$ contains Y and, therefore, $\overline{T_1}$ and $\overline{T_2}$ do not match).

One can reduce the overload by relaxing the factoring discipline. For instance, one could simply store the two atoms involved, in the history of factoring. This would prevent incorrectly repeating the same operation on the same atoms, but it would still allow factoring in cases where it could be prevented. The following example will clarify these observations.

Example 3.3.7 Assume one has $a(1) \wedge b(1,2) \wedge a(X) \wedge b(X,Y)$. One can apply the rule to $a(1)$ and $a(X)$ to yield: $a(1) \wedge b(1,2) \wedge X = 1 \wedge b(X,Y) \ \vee \ a(1) \wedge b(1,2) \wedge a(X) \wedge b(X,Y) \wedge \mathbf{X \neq 1}$, where $\mathbf{X \neq 1}$ represents the history of factoring HF. This history would prevent the application of the rule to $b(1,2)$ and $b(X,Y)$ in the second disjunct, because this would cause $X = 1$.

If, instead of the disequality, one stores "$\mathbf{a(X) \neq a(1)}$" in HF, then factoring of $b(1,2)$ and $b(X,Y)$ would not be prevented although, of course, that would not make any difference to the logic of the sentences.

Thus, it seems that some computationally useful simplifications are possible in the factoring rule, but only at the cost of adding logically irrelevant nodes to the frontier.

As in the case of propagation, one can ignore the history of factoring in the case of ground abducibles.

- **rewrite rules** for equalities, is a set of rules that could be regarded as the specification of an algorithm similar to Robinson's unification algorithm [Rob79] [Hog90], widely used in logic programming systems. The main difference between standard unification and the processing prescribed by the *rewrite rules* is the special treatment of variables, for these can be existentially or universally quantified. Equalities in implications, as defined below, cannot be processed by standard unification.

  One important consideration for the efficiency of the implementation of the proof procedure is that *rewrite rules* have priority over any other inference rule. They have to be applied exhaustively and as frequently as possible, to process any equality that may have been produced by the application of the other rules.

  The rewrite rules are described by the algorithms below in tables 3.6, 3.7, 3.8 and 3.9 and can also be found in [Fun96].

[7] Or to HP, as these arguments also apply for histories of propagation

[8] Recall that we are using PROLOG convention. Term names starting with capital letter represents variables, otherwise they are constants or functors.

In addition to these inference rules, we must add to the proof procedure the conventions about treatment of quantifiers, already mentioned at the beginning of this section: Variables in Δ and/or UC are implicitly existentially quantified. Variables in implications are *either* implicitly universally quantified with scope the whole implication in which they appear *or* existentially quantified because the also appear in Δ and/or UC.

Although we could allow for explicit (existential and universal quantification in implications (as shown chapter 5 as part of the presentation of the ACTILOG language), we follow Fung's conventions for quantifiers (stated above) in order to simplify the presentation of an algorithms. This presentation is the subject of the remaining sections of this chapter. Other operational details of the proof procedure are discussed with the algorithms.

3.4 An Any-time Algorithm for the iff Proof Procedure

Anytime algorithms were first mentioned by Dean and Boddy [DB88] to refer to programs that can be interrupted at any time during their operation. When provided with more resources or time to compute, these algorithms will produce results of higher quality. This description is particularly applicable to the reasoning mechanism of a reactive agent. As explained in chapter 1, an agent must be able to interrupt its *thinking* in order to receive information from the environment, assimilate it, and more importantly, to act in response to these inputs. If there is no pressure to act, the agent may well devote more time to goal processing, for instance, obtaining more refined plans of action to achieve its goals, possibly avoiding actions that can be predicted to result in eventual failure. In this sense, its plans would be improved.

We must add to Dean and Boddy's description an extra condition: the requirement for **re-entering**. After interrupting its reasoning mechanism to act, the agent must be able to resume its reasoning process *from the point where it left it.*

We have already explained (in chapters 1 and 2) that the interruption of reasoning is "controlled" by the resource parameter R provided as a argument for *demo*. In the algorithms shown below, R *counts* the derivation steps performed by the application of the inference rules. When that counting *hits* a predefined (defined before calling *demo*) value N, processing is suspended. An interesting topic for further research is how that value N is set. Most probably, it would be a function of the previous "mental state" of the agent (see chapter 2).

For *re-entering* (carry on the processing of goals from where it left it), the system needs to maintain its "state of computation" between interruptions. The frontier of nodes is already a sufficient structure to preserve the "state of computation" between successive calls to *demo*, the logic program that implements the agent's reasoning mechanism. The effects of re-entering on the agent's previous goals and plans will be discussed in chapter 4 in the context of the treatment of inputs.

The following sections present the derivability relation (*demo*) of an agent as a logic program. The choice of language for the specification (logic programming) is justified by the following considerations:

1. We will end using only one language, normal logic programming, to describe the agent architecture (chapter 2), the agent reasoning mechanism (this chapter) and the agent's procedural knowledge (chapter 4).

2. Nondeterminism and certain degree of paralellism in the applications of the inference rules, can be modelled by a logic program. Different behaviour can be obtained by changing the strategy of control of the same logic program. Both possibilities would be fixed in a procedural description.

3. Pure normal logic programs do not suffer from the ambiguity of destructive assignment of values to variables. Variable assignments are unique. One always refers to the values in certain data structure before an after some major transformation by using different terms. This discipline is particularly useful for modelling interleaving of processes. One will always be able to refer to the particular set of values of data structure at a time-point.

4. Control mechanisms can themselves be represented inside logic programs in different ways and other extra-logical resources (such as cuts) could be used to optimize the logic code, should it be necessary.

In what follows, *demo* is described in detail. Tables 3.1, 3.2, 3.3,3.4, 3.5, 3.6, 3.7, 3.8 and 3.9 contain the definition of *demo* and its subsidiary predicates. We discuss each table afterwards, to clarify the operational details of these logic programs.

First, some remarks about the notation:

- To make easier the reading of the logic programs below, we attach a "nesting index" to every $\vee$ and $\wedge$ operator. This is to indicate which operators are at the same level of nesting (marked with the same index) within a complex formula;

- The operator $=$ is used to indicate syntactic equality as defined by some suitable equality theory left implicit. The operator $\equiv$ denotes logical equivalence and it "encapsulates" the **splitting** rule of inference and other logical transformations such as $\mathbf{false} \wedge A \equiv \mathbf{false}$ and $\mathbf{true} \wedge A \equiv A$. Thus, if $F_1 \equiv F_2$ holds for frontiers F_2 and F_2, it means that either they are identical or one can be transformed into the other by the splitting rule or one of these logical transformations.

- Instead of the logical rendering of nodes (as conjunctions of literal and implications), we use the structure that implements them. The intention is to have immediate access to the control information related to each node, namely histories of propagations and factoring and the distinction between Δ, UC and CN that will be explained below. All these elements can be described in the logical notation, but the description would be considerable longer and potentially confusing. Thus, when we write

$$N_i \equiv (\Delta, UC, CN, HF, M) \tag{3.25}$$

  we mean that **node** N_i has Δ as its set of abducibles, UC as the rest of the unconditional goals expecting processing, CN the conjunction of all the implications in the node, HF the history of factoring for abducibles in this node and M is the set of variables in the initial query. CN will have some internal structure as well because we need to distinguish each implication and its history of factoring. So, when we write:

$$CN \equiv (Imp, HP) \wedge CNRest \tag{3.26}$$

  we mean that CN's first implication is Imp the history of which is HP. The rest of CN is, of course, $CNRest$.

3.4.1 The main routine: demo

demo (in table 3.1) is one of the two topmost predicates of the specification (the other is **demo_impl**). It encapsulates tests for termination and selection of nodes for further processing and it invokes the lower level routines to apply the inference rules (**demo_abd** as explained below) on those nodes.

DEMO	
$demo(KB, InGoals, OutGoals, R)$	
$\leftarrow$ ($R = 0 \ \wedge_1 \ OutGoals \equiv InGoals$)	**[DEMO – BAS1]**
$\vee_1$ ($empty(InGoals)$	
$\wedge_1 \ R \neq 0$	
$\wedge_1 \ empty(OutGoals)$)	**[DEMO – BAS2]**
$\vee_1$ ($\neg rule\_apply(InGoals)$	
$\wedge_1 \ R \neq 0$	
$\wedge_1 \ OutGoals \equiv InGoals$	
$\wedge_1 \ \neg empty(InGoals)$)	**[DEMO – BAS3]**
$\vee_1$ ($rule\_apply(InGoals)$	
$\wedge_1 \ R \neq 0$	
$\wedge_1 \ \neg empty(InGoals)$	
$\wedge_1 \ rewrite\_disj(InGoals, NextGoals)$	
$\wedge_1$ (($NextGoals \equiv (FirstPlan \vee AltGoals)$	
$\wedge_2 \ urg\_order(FirstPlan, OrderedPlan, R_{urg})$	
$\wedge_2 \ R_{urg} < k$	
$\wedge_2 \ NewGoals \equiv (OrderedPlan \vee AltGoals)$)	
$\vee_2$ ($empty(NextGoals) \wedge_2 \ empty(NewGoals)$))	
$\wedge_1 \ demo\_drop(KB, NewGoals, OutGoals, R-1)$	**[DEMO – RECU]**
$demo\_drop(KB, InGoals, OutGoals, R)$	
$\leftarrow$ ($InGoals \equiv (FirstG \vee AltGoals)$	
$\wedge_1$ (($false \equiv FirstG$	
$\wedge_2 \ demo(KB, AltGoals, OutGoals)$)	
$\vee_2$($false \not\equiv FirstG$	
$\wedge_2 \ demo\_abd(KB, InGoals, OutGoals)$)))	**[DEMO – DROP]**

Table 3.1: The *demo* predicate

demo has four arguments: the knowledge base KB, containing the definitions used to unfold goals into subgoals; $InGoals$ contains the frontier of nodes provided as input to *demo*; $OutGoal$, contains the final frontier of nodes, i.e. the frontier when *demo* is suspended; and R is the argument responsible for the *any-time* character of the program. It functions as a counter of the steps in the derivation starting with the initial frontier.

The *demo* program is described by four sub-clauses:

- **[DEMO-BAS1]** ... **[DEMO-BAS3]** gather the *base cases* when *demo* will suspend processing. Declaratively, these clauses can be read: No frontier can be derived from the initial frontier ($InGoals$ remain unaltered) if either, 1) the prover has spent all its resources ($R = 0$), 2) the $frontier$ is empty or 3) no rule can be applied to the frontier.

- **[DEMO-RECU]** is a more complex clause. It says: To derive $OutGoals$ from $InGoals$ 1) apply rewrite rules to the frontier **and if** the frontier is not empty after this **then** 1.1) select the first node in the new frontier, 1.2) order the literals (and implications) in the nodes (this task is further explained below and it is also discussed in chapters 4 and 5) and 1.3) create a new frontier with the ordered first node, **else** the new frontier will also be empty; 2) call *demo_drop* with the new frontier.

The routine *demo_drop* **prunes** the frontier, removing those nodes that entail falsity (i.e. are equivalent to **false**).

The role of *urg_order* above is related to the goal selection strategy which, as explained in [Kow95], could be embedded in the definition of the predicate $\equiv$. However, we argue, in chapter 4, that a more general priority mechanism, also resource bounded (the reason for $R_{urg} < k$, where k is a pre-defined constant value), could be used to increase the efficiency of the theorem prover with domain specific information. For the time being, *urg_order* can be omitted without affecting the theorem prover.

Another aspect that needs clarification is the role of R. As has been mentioned several times above, the *resource* argument counts the steps of the derivation from the input frontier to the output frontier. The main assumption is that each one of these steps (the application of one rule of inference), represents a *minimal* unit of processing in the agent's reasoning mechanism. Once this atomic unit of processing has been activated, it can not be stopped until it has been completed. There is no such thing as a *partial* application of an inference rule. Moreover, we further assume that all the inferences take exactly the same amount of resources. These two assumptions allow us to add one unit to the counting per applied rule. For further simplicity, however, resource counting for rewrite rules and case analysis is not considered. These are, of course, working simplifications. Conceptually, every derivation step has its own particular operational latency (duration), presumably highly dependent on the actual physical implementation of the computing device that performs the derivations.

3.4.2 The abductive procedure: demo_abd

We have been taking liberties with the form of the logic programs presented so far (including those in chapter 2 describing **GLORIA**). Some of these logical descriptions are not in clause form. However, they can be easily transformed into PROLOG clauses following the standard procedures described, for instance, in [Kow79b] and [Hog90]. We carry on with this non-clausal style for the sake of the clarity and brevity of the presentations. Labels are used to highlight **sub-clauses** (conjunctions in the body of a main clause) that are related to certain tasks.

demo_abd contains the procedures that perform the operations described in previous sections, and some additional ones that we explain below. The processing starts with the selection

DEMO_abd : The abductive procedure	
$demo\_abd(KB, InGoals, OutGoals, R)$	
$\leftarrow\ InGoals \equiv\ FirstNode \vee AltGoals$	
$\wedge_1\ FirstNode = (\Delta, (G \wedge Rest), CN, HF, M)$	
$\wedge_1$ (($G = \neg G'$	
$\wedge_2\ NewCN = ((\mathbf{false} \leftarrow G'), \{\}) \wedge CN$	
$\wedge_2\ NewNode = (\Delta, Rest, NewCN, HF, M)$	
$\wedge_2\ NextGoals = (NewNode \vee AltGoals)$	
$\wedge_2\ demo\_impl(KB, NextGoals, OutGoals, R-1)$	**[DMAB − NEG]**
$\vee_2$ ($G \neq \neg G'$	
$\wedge_2$ (($unfoldable(G)$	
$\wedge_3\ definition(KB, G, D)$	
$\wedge_3\ NewNode = (\Delta, (D \wedge Rest), CN, HF, M)$	
$\wedge_3\ NextGoals \equiv NewNode \vee AltGoals$	
$\wedge_3\ use\_order(NextGoals, OrdGoals, R_{use})$	
$\wedge_3\ R_{use} < k_{use}$	
$\wedge_3\ NextGoals = OrdGoals$)	**[DMAB − UNF]**
$\vee_3$	
(($equality(G) \vee_4 inequality(G)$)	
$\wedge_3\ \Delta' = G \wedge \Delta$	
$\wedge_3\ NewNode = (\Delta', Rest, CN, HF, M)$	
$\wedge_3\ NextGoals = (NewNode \vee AltGoals)$)	**[DMAB − EQU]**
$\vee_3$	
($abducible(G)$	
$\wedge_3\ factorable(\Delta, G, HF)$	
$\wedge_3\ factoring(InGoals, NextGoals)$)	**[DMAB − FAC]**
$\vee_3$	
($abducible(G)$	
$\wedge_3\ \neg factorable(\Delta, G, HF)$	
$\wedge_3\ \Delta' = G \wedge \Delta$	
$\wedge_3\ NewNode = (\Delta', Rest, CN, HF\ )$	
$\wedge_3\ NextGoals = (NewNode \vee AltGoals)$))	**[DMAB − ABD]**
$\wedge_2$ ($demo(KB, NextGoals, OutGoals, R-1)$)))	
$\vee_1$ ($\neg rule\_apply\_to\_uc(FirstNode)$	
$\wedge_2\ demo\_impl(KB, InGoals, OutGoals, R)$)	**[DMAB − NRA]**
$rule\_apply\_to\_uc(Node) \leftarrow\ Node \not\equiv \mathbf{false}$	
$\wedge_1\ Node = (\Delta, UC, CN, HF, M)$	
$\wedge_1\ literal(G) \wedge_1\ G \in UC$	**[NRA − UC]**

Table 3.2: The abductive procedure

of a node ($FirstNode$) and then, from the UC component of this node, the program selects a literal on which the application of a rule will be attempted. The subsequent operations are:

- **[DMAB-NEG]**: *If* the selected literal is a negative atom, *then*:

  1. Add the corresponding implication to CN. Note that the new implication is initiated with an empty history of propagation ($\{\}$).
  2. Rebuild the node, excluding the original literal from UC.
  3. Call the routine *demo_impl* (described below) with a new frontier that has the new node as its first node. This maneuver is intended to activate the processing of the just-added implication.
  4. Eventually *demo_impl* will return the control to *demo* to carry on processing other literals in UC.

  The transformation performed by this sub-clause is similar to Fung and Kowalski's transformation of [FK96] "initial" queries and within the splitting rule. Observe that the algorithm emulates the resolution mechanism of a SLDNF theorem prover like PROLOG. Here, as in SLDNF, once a negative atom has been selected for reduction a separate procedure (in this case *demo_impl*) is called to process it. The main difference, with respect to SLDNF, is that this new process may end up not only proving or disproving the atom, but also *suspending* the reduction of the body of the implication, without making a final decision on the truth or provability of every atom in it.

  Further explanations of this novel strategy are provided below. One can add here, however, that this suspension strategy can be profitably combined with abduction. Processing of an implication is suspended when the reduction hits an *abducible atom* that can not be resolved by propagation. Abduction is not performed on abducibles atoms in the body of implications. In doing so, the abducible atoms are *minimized*. The reasons and consequences of this are further discussed below, in the description of the program *demo_impl*.

  If the selected literal is **not** a negative atom, one of the following (mutually exclusive) operations will be performed.

- **[DMAB-UNF]**: If the selected literal can be unfolded (*unfoldable(G)* i.e. not abducible or equality or inequality, as explained below) then its definition is retrieved from the knowledge base (if there is no definition **false** will be provided as proxy definition). The new definition takes the place of the literal in the new version of the first node, which is then placed in the new frontier.

  Recall that the predicate $\equiv$ (i.e. the program or routine that implements the equivalence relation), has the *splitting rule* embedded in it. As a consequence, after the call in this clause the new first node, that contains disjunction D (the definition of the literal G) in UC, will be processed by $\equiv$ so that the conjunction $Rest$ is distributed over the disjunction, yielding a new *flat* frontier in $NextGoals$.

  The predicate *use_order*, as *urg_order*, is further discussed in chapter 4. As the syntax suggests, this predicate performs an *ordering* of the nodes (unlike *urg_order* that orders the conjuncts "inside" a node) and can be omitted or trivially implemented as a program that makes a copy of $NextGoals$ into $OrdGoals$. We explain in that chapter how this predicate is part of the (resource bounded) mechanism to incorporate *useful*, domain-specific heuristics for goal processing into the agent's programs. Its appearance here serves to illustrate the location of this sort of procedure within the reasoning mechanism.

- **[DBMAB-EQU]**: This clause simply decides whether the literal being processed is an equality or a disequality, in which case it transfers it to Δ for later processing. In the **iff** proof procedure, equalities require special treatment, as explained before, due to the need of distinguishing between existentially and universally quantified variables. This distinction is critical when one wants to retrieve an answer from the final frontier in a derivation. We have chosen to extend the treatment in order to deal with inequalities ($<$) in a similar form. That is, a set of rewrite rules has been built to reduce conjunctions with inequality as well as with equalities. The set required by the planning application is simply one that "suspends" the reduction of inequalities when there are variables (unknown values) involved, and processes them in the corresponding manner when they are instantiated. These rules for equalities and inequalities are shown in tables 3.8 and 3.9 and are explained below.

  It is worth noticing, however, that equalities and inequalities can be regarded as abducible predicates or predicates with no definition in the knowledge base. They can, therefore, be incorporated into the abductive framework $< T, IC, Ab >$, without altering its semantics. Nevertheless, the predicate $abducible(G)$ in our description is intended to refer to abducibles other than equalities and inequalities.

- **[DMAB-FAC]**: This clause implements the **factoring** transformation described in the previous section. Two tests restrict the application of the factoring rule to the first node of the current frontier (done by $factoring(InGoals,\ NextGoals)$). The first test ($abducible(G)$) establishes whether the literal under consideration is an abducible and not an equality or inequality.

  The second test is specific to factoring. It checks whether there is an atom in Δ that can be used to factor with the atom G, according to the history of transformation kept in the node. If there is such an atom, then the factoring is carried out.

- **[DMAB-ABD]**: If the literal G is an abducible but there is no atom with which G has not been factored before, then G must be suspended. As a transformation of the data structures, this implies removing G from UC and placing it in Δ. *Suspension* is this sort (moving a predicate from UC into Δ) is equivalent to *abduction*).

- **[DMAB-NRA]**: If none of the alternatives above applies, which will be the case when UC is empty or equivalent to **false**[9] then control passes to *demo_impl*, to check whether any of the rules involving implications can be apply.

  One can increase the efficiency of an implementation by dropping, at this stage, a node that is equivalent to **false**, instead of waiting until the control returns to *demo_drop*. To make the presentation simple, we have not include this possibility in the program in table 3.2.

3.4.3 Processing implications: demo_impl

The program *demo_impl* deals with the application of inference rules (those that can be applied) to literals in implications. It could be regarded as the topmost procedure of the prover. One can start processing an initial frontier either by calling *demo_impl* or *demo*. It all depends on whether one wants attention first on activation of goals (done by *demo_impl*) or on goal reduction (done by *demo*). These processes, however, will call each other and processing will stop when *neither* can modify the frontier anymore.

[9] (as tested by $\neg rule\_apply\_to\_cn(FirstNode)$).

DEMO_IMPL : Processing implications	
$demo\_impl(KB, InGoals, OutGoals, R)$	
$\leftarrow$ ($R = 0$ $\wedge_1$ $InGoals = OutGoals$)	[**DMIM – BAS1**]
$\vee_1$ ($empty(InGoals)$	
$\wedge_1$ $R \neq 0$	
$\wedge_1$ $empty(OutGoals)$)	[**DMIM – BAS2**]
$\vee_1$ ($InGoals \equiv (FirstNode \vee AltGoals)$	
$\wedge_1$ $R \neq 0$	
$\wedge_1$ $\neg rule\_apply\_to\_cn(FirstNode)$	
$\wedge_1$ $InGoals = OutGoals$))	[**DMIM – BAS3**]
$\vee_1$ ($InGoals \equiv (FirstNode \vee AltGoals)$	
$\wedge_1$ $exist\_quant\_vars(FirstNode, ExQVars)$	
$\wedge_1$ $FirstNode = (\Delta, UC, CN, HF, M)$	
$\wedge_1$ $R_o + R_r \leq R$	
$\wedge_1$ $demo\_each\_impl(KB, ExQVars, \Delta, CN, \{\}, CN', R_o)$	
$\wedge_1$ (($case\_analysis(ExQVars, \Delta, CN', Eq, CN_1, CN_2)$	
$\wedge_2$ $NewNode_1 = (\Delta, (Eq \wedge UC), CN_1, HF, M)$	
$\wedge_2$ $NewNode_2 = (\Delta, UC, CN_2, HF, M)$	
$\wedge_2$ $NextGoals \equiv (NewNode_1 \vee NewNode_2 \vee AltGoals)$	$[DMIM - CA]$
$\vee_2$ ($promotion(CN', Head, CN'')$	
$\wedge_2$ $NewNode = (\Delta, (Head \wedge UC, CN'', HF, M)$	
$\wedge_2$ $NextGoals \equiv (NewNode \vee AltGoals)$)	$[DMIM - PRO]$
$\vee_2$ (($\neg case\_analysis(ExQVars, \Delta, CN', Eq, CN_1, CN_2)$	
$\wedge_2$ $splitting\_head(ExQVars, CN', Head, CN_1, CN_2)$	
$\wedge_2$ $NewNode_1 = (\Delta, (Head \wedge UC), CN_1, HF, M)$	
$\wedge_2$ $NewNode_2 = (\Delta, UC, CN_2, HF, M)$	
$\wedge_2$ $NextGoals \equiv (NewNode_1 \vee NewNode_2 \vee AltGoals)$	$[DMIM - SPL]$
$\vee_2$ ($\neg case\_analysis(ExQVars, \Delta, CN',$ $Eq, CN_1, CN_2)$	
$\wedge_2$ $\neg splitting\_head(ExQVars, CN', Head, CN_1, CN_2)$	
$\wedge_2$ $\neg promotion(CN', Head, CN'')$	
$\wedge_2$ $NewNode = (\Delta, UC, CN', HF, M)$	
$\wedge_2$ $NextGoals \equiv (NewNode \vee AltGoals)$))	$[DMIM - NRA]$
$\wedge_1$ $demo(KB, NextGoals, OutGoals, R_r)$)	[**DMIM – RECU**]
$rule\_apply\_to\_cn(Node)$	
$\leftarrow$ $Node = (\Delta, UC, CN, Hf, M)$	
$\wedge_1$ $some\_rule\_applicable(CN)$	[**NRA – CN**]

Table 3.3: The demostration procedure for implications

DEMO_EACH_IMPL : Processing each implication	
$demo\_each\_impl(KB, ExQVars, \Delta,$	
$\qquad InImps, PreviousImps, OutImps, R)$	
$\leftarrow\ (\ R \neq 0 \wedge_1\ empty(InImps) \wedge_1\ OutImps = PreviousImps)$	
$\vee_1\ (\ R = 0 \wedge_1\ OutImps = (PreviousImps \wedge InImps))$	
$\vee_1\ (\ R \neq 0 \wedge_1\ InImps = (FirstImp \wedge Rest)$	
$\wedge_1\ demo\_one\_impl(KB, ExQVars, \Delta, FirstImp, NewImp)$	
$\wedge_1\ (\ (\ suitable\_for\_ca\_pro\_spl(ExQVars, \Delta, NewImp)$	
$\wedge_2\ RestImp = (PreviousImps \wedge Rest)$	
$\wedge_2\ OutImps = (NewImp \wedge RestImp))$	**[DMEA – CP]**
$\vee_2\ (\ suitable\_for\_deletion(NewImp)$	
$\wedge_2\ demo\_each\_impl(KB, ExQVars, \Delta,$	
$\qquad Rest, PreviousImps, OutImps, R-1)\ )$	**[DMEA – DEL]**
$\vee_2\ (\ suitable\_for\_splitting(NewImp)$	
$\wedge_2\ NextImp \equiv NewImp$	
$\wedge_2\ NewRest = (NextImp \wedge Rest)$	
$\wedge_2\ demo\_each\_impl(KB, ExQVars, \Delta$	
$\qquad NewRest, PreviousImps, OutImps, R-1))$	**[DMEA – SPL]**
$\vee_2\ (\ suitable\_for\_equality\_treatment(NewImp)$	
$\wedge_2\ NewRest = (NewImp \wedge Rest)$	
$\wedge_2\ demo\_each\_impl(KB, ExQVars, \Delta,$	
$\qquad NewRest, PreviousImps, OutImps, R-1)\ )$	**[DMEA – EQU]**
$\vee_2\ (\ \neg suitable\_for\_ca\_pro\_spl(ExQVars, \Delta, NewImp)$	
$\wedge_2\ \neg suitable\_for\_deletion(NewImp)$	
$\wedge_2\ \neg suitable\_for\_splitting(NewImp)$	
$\wedge_2\ \neg suitable\_for\_equality\_treatment(NewImp)$	
$\wedge_2\ some\_rule\_applicable(NewImp)$	
$\wedge_2\ NewRest = (NewImp \wedge Rest)$	
$\wedge_2\ demo\_each\_impl(KB, ExQVars, \Delta,$	
$\qquad NewRest, Previous, OutImps, R-1)\ )\ )\ )$	**[DMEA – NRA]**
$\vee_2\ (\ \neg suitable\_for\_ca\_pro\_spl(ExQVars, \Delta, NewImp)$	
$\wedge_2\ \neg suitable\_for\_deletion(NewImp)$	
$\wedge_2\ \neg suitable\_for\_splitting(NewImp)$	
$\wedge_2\ \neg suitable\_for\_equality\_treatment(NewImp)$	
$\wedge_2\ \neg some\_rule\_applicable(NewImp)$	
$\wedge_2\ NewPrevious = (PreviousImps \wedge NewImp)$	
$\wedge_2\ demo\_each\_impl(KB, ExQVars, \Delta,$	
$\qquad Rest, NewPrevious, OutImps, R-1)\ )\ )\ )$	**[DMEA – REC]**

Table 3.4: Processing each implication

DEMO_ONE_IMPL : processing one implication	
$demo\_one\_impl(KB, ExQVars, \Delta, InImp, OutImps)$	
$\leftarrow$ ($noimp(InImps)$ $\wedge_1$ $empty(OutImps)$)	**[DMON – BAS1]**
$\vee_1$ ($\neg rule\_apply\_imp(InImp)$	**[DMON – BAS2]**
$\vee_1$ ($InImp \equiv (H \leftarrow (G \wedge Rest), HP)$	
$\wedge_1$ ((($equality(G) \vee_3$ $inequality(G)$)	
$\wedge_2$ (($process\_equalities(ExQVars, InImp, NewImp)$	
$\wedge_3$ $demo\_one\_impl(ExQVars, \Delta, NewImp, OutImps)$)	
$\vee_3$ ($\neg process\_equalities(ExQVars, InImp, NewImp)$	
$\wedge_3$ $OutImps = (NewImp)$)))	**[DMON – EQU]**
$\vee_2$ ($G = \neg G'$	
$\wedge_2$ $H' = (H \vee G')$	
$\wedge_2$ $NewImp = (H' \leftarrow Rest, HP)$	
$\wedge_2$ $demo\_one\_impl(ExQVars, \Delta, NewImp, OutImps)$)	**[DMON – NEG]**
$\vee_2$ ($unfoldable(G)$	
$\wedge_2$ $definition(KB, G, D)$	
$\wedge_2$ $OutImps \equiv (H \leftarrow (D \wedge Rest), HP)$)	**[DMON – UNF]**
$\vee_2$ ($abducible(G)$	
$\wedge_2$ $propagation(\Delta, InImp, OutImps)$))))	**[DMON – ABD]**
$rule\_apply\_imp(Imp)$	
$\leftarrow$ $Imp \equiv (H \leftarrow (G \wedge Rest), HP)$	
$\wedge_1$ ($equality(G) \vee_2$ $inequality(G)$	
$\vee_2$ $unfoldable(G) \vee_2$ $G = \neg G'$	
$\vee_2$ ($abducible(G) \wedge_2$ $propagable(\Delta, G, HP)$)	**[NRA – IMP]**
$some\_rule\_applicable(Imps)$	
$\leftarrow$ $Imps \equiv (Imp \wedge RestCN)$	
$\wedge_1$ $rule\_apply\_imp(Imp)$	**[NRA – IPS]**

Table 3.5: Processing one implication

Within *demo_abd*, *demo_impl* itself is invoked by [DMAB-NEG], which rightly suggests that the program is involved in the treatment of negations. This is due to the fact that the proof procedure transforms negation into implication with **false** by head. This should not, however, obscure the fact that the processing explained by *demo_impl* is applicable to any implication in a node. Actually, because one has to process all these implications (integrity contraints) in the goals to safeguard the correctness of the proof procedure, *demo_impl* will try to process *all* the implications in the node. The process will only stop if no implication admits further rule application or the system runs out of resources ($R = 0$).

Notice that although *demo_impl* is used to reduce negative literals in the node its role is more general and includes the aforementioned process of *activation of goals*, as will be discussed below.

Having *demo_abd* spawning a (*demo_impl*) process to deal with negation suggests a close paralellism between SLDNF and **iff**PP. As mentioned above (in the description of the rules of inference), the **iff**PP seeks to generalize the treatment of negations for those cases where no commitment can be made with respect to the truth or falsity of an atom, because it depends on the truth value of some *undefined* atoms. The evaluations of these latter atoms must be *suspended*, waiting until "evidential" or contextual support allows for a decision to be made.

In SLDNF, something that can not be proven is regarded as false. The proof procedure implicitly closes the knowledge base, which is tantamount to state that the prover has all the knowledge related to the problem being handled. This *closure* is what completion semantics seeks to capture.

The **iff** proof procedure, on the other hand, does not *fix* the information related to those predicates whose definitions changes or are known to be incomplete in some way. It is capable of doing this because the abductive framework that defines each program, has been given a semantics where undefined atoms can be made sense of.

Abductive programs, however, involve some subtleties related with negation that require more clarification. For instance, at the beginning of the chapter we said that from a goal g and the rule $g \leftarrow a_1 \wedge \ldots \wedge a_n$, the "explanation" $a_1 \wedge \ldots \wedge a_n$ may be abduced. This means that the proof procedure, when queried with g about a theory containing only that rule, answers back with a conjunction $a_1 \wedge \ldots \wedge a_n$, provided that all the a_i belong to the set of abducible predicates Ab[10]. If one of the a's, say a_j is a literal such that $a_j = \neg b_j$, where b_j is an abducible atom (it belongs to Ab) then, despite being an abducible, *one would not want to abduce* b_j because that would invalidate the explanation provided by the other a's. Of course, if b_j has already been abduced (outside the negation), or it is known to be true, then there is no choice but to drop the explanation as invalid.

So, at first glance, it seems that abducible atoms require different treatment depending on whether they are "inside" or "outside" a negation, i.e. in the body of an implication with head **false** (*inside*) or somewhere else (*outside*). Whether an atom is inside or outside a negation can always be established by applying a well-defined set of rules.

Thus, abduction (i.e. shifting an atom to Δ) should not be performed inside a negation. This could be regarded, we believe, as a form of *context-dependent abduction*, where the context is set by the body of the implication that contains the abducible atom. It is interesting to note that this context-dependent, syntax-driven restriction of abduction has a semantic justification. The justification is the **minimality** of the abduction that the restriction brings about, which is better explained by an example:

[10] Notice that what belongs to the set Ab is the predicate name of each a_i, not the actual atom.

Example 3.4.1 Consider the abductive logic program $< T, IC, Ab >$, where:

$$\begin{array}{ll} T: & a \leftrightarrow c \wedge \neg b \\ Ab: & \{c, b\} \\ IC: & \{\} \end{array}$$

Trying to answer the query a, the proof procedure will unfold a into $c \wedge \neg b$. If the criterion were strictly "add abducibles to Δ whenever you cannot factor or propagate them", then the answer (as extracted from Δ) would be $\{c, b\}$. This "explanation" is consistent with $a \leftarrow c \wedge \neg b$ (easier to observe by rewriting it as $a \vee b \leftarrow c$), the "if" side of the definition of a above, but not with the "only-if" side which forces the exclusion of b.

Thus, if the proof procedure is to correctly capture the semantics of the definitions, then it must *inhibit* abduction inside negations. Moreover, in general and for the same reason, abduction should not be performed on atoms in the body of any implication.

Observe that this discussion is consistent with the interpretation of *abduction* as *deduction* on the *only if* part of the definitions in an abductive logic program, as explained by Fung [Fun96].

We argue for another form of context-dependent abduction in chapter 6. Meanwhile, the discussion above is sufficient as conceptual support for the description of the *demo_impl* program in table 3.3, which follows:

- **[DMIM-BAS1]** ... **[DMIM-BAS3]** are the base cases of the definition, defining the conditions under which processing of implications will be suspended. Lack of resources is stated by **[DMIM-BAS1]** and processing on an empty frontier is prevented by **[DMIM-BAS2]**. **[DMIM-BAS3]** states that no further processing is possible because no rule can be applied to any implication in the *first node*, even though there may be resources for further computation.

  Observe that **[DMIM-BAS3]** calls the program *rule_apply_to_cn* to establish whether a rule of inference can be applied to some implication in the current first node. This program is specified by the clause **[NRA-CN]** which is not as simple as **[NRA-UC]** in table 3.2. An extra condition of this clause caters for the case when there is an abducible in the body of an implication that could be used to propagated some of the previously abduced atoms in this node (kept in this node's Δ). In that case, we say that Δ is **propagable** through G, and the history of propagation is employed in establishing it.

  The definition of *rule_apply_to_cn* implies that processing must stop, not only when CN is empty but more importantly *when there is no abducible in an implication that can be used for propagation.*

  An important implementational detail about **[NRA-CN]** in table 3.3 and **[NRA-UC]** in table 3.2 is that these tests coincide with the conditions of the inference rules and therefore need not be repeated. Instead, extra-logical devices, like "cuts" in PROLOG or "flags" in general, can be used to indicate that no rule has been applied since the last iteration of the the corresponding recursive programs (the other clauses in *demo_abd* and *demo_impl*, respectively). We have followed this strategy in a prototypical implementation.

- **[DMIM-RECU]**: This part of the specification of *demo_impl* describe a more complex process. The intended sequence of execution (which could be enforced with a more complicated logical description and that is carried on by a PROLOG interpreter) is as follow:

  1. The first node of the frontier is retrieved ($FirstNode$).

2. The existentially quantified variables in this node are collected ($ExQVars$).
3. The program *demo_each_impl*, described in table 3.4 and explained below, is invoked with KB, $ExQVars$, Δ, CN, $\{\}$ and R_o as input arguments. The program computes and output CN' which is the revised version of CN after exhaustively applying some of the inference rules to it (see below). We exploit the "double-modality" of arguments in logic programming (parameter of a procedure can be used in both inputs and output mode) with argument R_o. One does not really known the value of R_o when *demo_each_impl* is called. But one knows that whatever the value, it must total R when added to R_r. This restriction is somehow "inputted" to the program. Thus, R_o is bounded by R. If *demo_each_impl* consumes all the resources (in R) then $R_r = 0$. Otherwise, R_r is assigned the difference $R - R_o$. However, this is just one possible strategy. Other assignments (satisfying $R_o < R$ and $R_r > 0$) are also allowed by this any-time algorithm.
4. After the call to *demo_each_impl* and before calling *demo*, one of the following alternatives is chosen:
   - *[DMIM-CA]*: The first condition of this clause tests whether **case analysis** can be applied to the *first implication* Imp in CN' ($CN' \equiv (Imp \wedge CNRest)$). If case analysis is indeed applicable then the program *case_analysis* will output Eq, CN_1 and CN_2 satisfying:

     $$Imp \equiv (H \leftarrow Eq \wedge Rest, HP)$$
     $$CN_1 \equiv ((H \leftarrow Rest, HP) \wedge\ CNRest)$$
     $$CN_2 \equiv ((\mathbf{false} \leftarrow Eq, \{\}) \wedge\ CNRest)$$

     The other two conditions introduce the two nodes resulting from the case analysis into the frontier of nodes, as substitution for the original $FirstNode$.
   - *[DMIN-PRO]*: If the first implication, Imp, in CN' (if any) has a empty body (i.e. $Imp \equiv (Head \leftarrow \mathbf{true}, HP)$ then *promote* will remove the head in $Head$ and will return it as output. It will also return the remaining implications of CN' in CN''.

     Promotion means that $Head$ is shifted to UC. It also can be seen as the activation of all the goals in $Head$. However, promotion of a $Head$ is only allowed when that $Head$ contains no universally quantified variables. The $Head$ is, in general, a disjunction of atoms which would have eventually been split. So, this restriction on promotion is just an extension of the restriction on the splitting rule, as established by Fung and Kowalski [FK96].

     To preserve the *depth first search* strategy for selection of goals, common in SLDNF systems like PROLOG, the program must add the goal being activated as the first element of UC. This and other strategies can also be accomplished by a mechanism (implemented as *urg_order* in [DEMO_RECU]) for setting priorities for goals. This mechanism would favour more recently unfolded goals and it would be called after promoting a literal in H to UC.
   - *[DMIN-SPL]*: If neither case analysis, nor promotion are possible, the program will try to split the the implication, separating the body from the head into different nodes.

     Splitting of a head, as we call the rule implemented by this clause, is restricted by the following conditions:

     (a) The head of the implication to be split must not contain universally quantified variables. This is to avoid introducing universal variables into UC, which would then treat those variable as existentially quantified.

(b) The head of the implication must not be **false**. Splitting an implication of this sort would generate exactly the same frontier and could, therefore, plunge the system into loops.

– *[DMIN-NRA]*: When neither case analysis, nor promotion are applicable, this third clause will build a new frontier the first node of which is identical to the one in $FirstNode$ except that CN' substitutes CN.

5. The last call is to *demo* with a new frontier in $NextGoals$.

The logic program in Table 3.3 specifies the top-most procedures in the processing of implications. The rules for treatment of implications have been separated into two sets. One set is "encapsulated" by the program *demo_each_impl*. The other set is constituted by the rules for case analysis, splitting and the promotion mechanism (which is a sort of degenerated splitting). This separation is intended to give priority to those rules that do not increase the side of the frontier of goals. The rules in the second set, that do increase the frontier's nodes are left as last resource as suggested by [Fun96]. This program, together with *demo_abd*, intend a depth first search of the tree of derivations. The search strategy can be improved by reordering the nodes in the frontier, so that the "best" node is always the first node. This is the purpose of the programs *use_order* and *urg_order*, presented and discussed in chapter 4.

The program in table 3.4, on the other hand, extracts as many implications from the current node's CN as the resources allow. Every one of these implications is processed by *demo_one_impl*, which encapsulates the rules of unfolding, propagation and for treatment of negative literals, equalities and inequalities. The inputs to the program are the knowledge base, KB, the set of existentially quantified variable in this node, $ExQVars$, the abducibles in Δ and CN as $InImps$ together with the set of implications processed in previous iterations (which is, of course, initially empty). The output of the program is processed by the following clauses:

- **[DMEA-CPS]**: The first condition of this clause *suitable_for_ca_pro_spl*, tests whether the newly obtained implication, $NewImp$, is suitable either for **case analysis**, **promotion** or **splitting a head**. If any of this rules is applicable, the control returns to *demo_impl* with $NewImp$ as the first node of CN.

- **[DMEA-DEL]**: If the implication in $NewImp$ has a body equivalent to **false**, then it can be dropped all together. Process resumes with the next implication in CN, if any.

- **[DMEA-SPL]**: After unfolding and propagation, the body of $NewImp$ will possibly contain a disjunction of atoms as one of the conjuncts. The call to $\equiv$ will distribute the conjuncts over the disjunction, generate a new set of "flat" implications. $((H \leftarrow (D_1 \vee \ldots \vee D_n) \wedge C) \equiv (H \leftarrow D_1 \wedge C) \wedge \ldots \wedge (H \leftarrow D_n \wedge C)$ Processing continues on the first of this newly generated implications.

- **[DMEA-EQU]**: If $NewImp$ contains an equality or an inequality that has not been processed before, the implication is installed as first implication in CN and *demo_each_impl* is recursively called.

- **[DMEA-REC]**: Because of *demo_one_impl* (explained below), the returning $NewImp$ may contain implications that have not been exhaustively processed. In that case, *demo_each_impl* recursively calls itself. This is what this clause specifies.

- **[DMEA-NRA]**: Finally, if none of the previous rules apply, it means that the original implication in CN has been exhaustively processed by this set of inference rules on CN. The resulting set of implications is then stored in the bag of already processed implications ($NewPrevious$) and the process resumes with the next unprocessed implication.

Observe that *demo_each_impl* only stops, as the name suggests, when each and every implication in CN has been processed *or* when resources are exhausted. In this latter case, nothing guarantees that the answers given by the proof procedure will be correct, because there may be rules no applied at any time.

Sudden lack of resources is also a problem when the system is processing one application in particular (*demo_one_impl*). However, for simplicity, we have not included resource restriction into the specification of the program for that task (i.e. processing one implication is regarded as an atomic activity). Table 3.5 presents such a program. The program is called with KB, $ExQVars$, Δ, and one implication $InImp$ as inputs. As output, the program returns a conjunction of implications $OutImps$ as the result of processing $InImp$. The structure of the program is explained as follows:

- **[DMON-BAS1]** and **[DMON-BAS2]** states the terminating conditions of *demo_one_impl*. **[DMON-BAS1]** says that processing an implication is impossible if no implication is provided. In an implementation where implications in CN are kept in a list, this conditions will test whether the end of the list has been reached. The output list of implications would, therefore, be empty. **[DMON-BAS2]** establishes whether processing must stop because none of the rules implemented by this program can be applied to the body of $InImp$.

- **[DMON-EQU]**: The second part of the definition starts with the selection of a literal G, if any, from the body of $InImp$. Then, the condition in **[DMON-EQU]** tests whether G is an equality or inequality pending processing in the body of $InImp$. If G is that type of literals, the program invokes *process_equalities* to deal with it. As processing of one equality can enable other equalities for processing, the program *demo_one_impl* recursively calls itself, to pursue processing of other elements of the body of the same implication.

  If the equalities and inequalities in $InImp$ do not admit further processing, the program stops returning $InImps$ as the only element of in $OutImps$.

- **[DMON-NEG]**: If the selected literal G is a negative literal, this clause will attach its negation, $\neg G \equiv G'$, to the head of the implication. The program recursively calls itself to continue processing the same implication.

- **[DMON-UNF]**: If G is an unfoldable predicate, then the program retrieves its definition from the knowledge base (in D). Processing stop, and control is returned to *demo_each_impl* which will distribute the remaining conjuncts in the body of $InImp$ over the disjuncts in D **or** it will drop the implication all together, if $D \equiv$ **false**.

- **[DMON-ABD]**: Finally, if G is an abducible, then propagation will be attempted on it by using the abducibles in Δ.

  The rule of factoring (implemented in [DMAB-FAC] above) is intended to minimize the number of abduced atoms in Δ. It does so by using every (previously obtained) abducible atom to explain as many goals as possible, instead of throwing newly found abducibles into Δ.

  Similarly, propagation, as described in this clause, tries to use every abducible atom already in UC to resolve with an abducible in the body of an implication. It is interesting to observe that, despite their similarities, while propagation is a resolution step (as resolving b and $a \vee \neg b$ into a), factoring is not.

  Propagation is also similar to **unfolding**, if one regards the list of abducibles in Δ as definitions of the abducible predicates in the body of the implications. The program

exploits this similarity: instead of *propagating* one atom from Δ with the one in the body of the implication, the program *builds a definition* for the atom in the body, with all those abducibles in Δ that match it. This "definition" then *takes the place* of the original atom in the *resolvent* of the propagated atom and the original implication, as [DMON-UNF] indicates.

Observe that **iff**PP preserves the original implication subjected to propagation. In an open architecture, the system will be able to use the same implication to propagate new data in future calls to *demo_impl*. It also worth observing that the preservation of the original implication could be understood as an extension of traditional resolution-based theorem provers (such as SLD and SLDNF), which do not keep ancestors of the resolvent. Hogger suggests that the efficiency of those provers could be due to that strategy of no preservation of ancestors ([Hog90] pages 127 and 208). It would be interesting to test the practical impact of challenging that restriction as the **iff**PP does.

Also, notice that the construction of the "definition" is packing up several applications of the propagation rule in one operation. It could be called, therefore, *multiple propagation*.

This replacement operation is guarded by a test ($propagable(HP_i, \Delta, G)$) analogous to that in factoring. Only if a non-empty definition can be built for the atom (G) under consideration, propagation operations will continue. What that test does is to ensure that no abducible is used more than once for propagation, so avoiding the risk of nonterminating computations. The test relies, of course, on the existence of the **history of propagation** (HP_i) of all the abducible atoms in the body of implication i, in the node. Every time a non-empty definition is successfully built, those atoms in Δ involved in the operation are marked as used in (HP_i), as explained in the description of the propagation rule above.

This operation, together with unfolding and promotion, constitutes the set of operations required for *activation of goals*.

This completes the description and analysis of the routines of the proof procedure that manipulate non-abducible and most of the abducible predicates. In the following section the attention is focused on two special classes of abducibles: equalities ($=$) and inequalities ($<$).

3.4.4 Rewrite rules for equalities and inequalities

The original specification of the **iff** proof procedure does require a special treatment of equalities. On one hand, $=$ is an abducible predicate, so that no definition of it should be part of the knowledge base. This leaves the predicate "open" for changes. On the other hand, the predicate is not treated as other abducibles by the *factoring* and *propagation* rules. Instead, the proof procedure is furnished with a special set of rules for equality treatment i.e. those that simulate the unification algorithm and those for case analysis of equalities.

The discussion of the programs that implement these rules for equalities is the subject of this section. The other subject is the introduction and discussion of a similar set of rules to deal with inequalities ($<$, in particular). With the planning application in mind, inequalities are also regarded as special abducible predicates. As will be seen below, scheduling and temporal reasoning in planning are highly dependent on a flexible and efficient treatment of inequalities. As in the case of equalities, this is possible by means of predicate-specific manipulations.

We have chosen to implement treatment of equalities and inequalities within the same routines because most of the processing is the same for both kinds of predicates. Conceptually, one could see an inequality as any other abducible that requires no special treatment, except for appropriate integrity contraints. The effects of this special treatment on the soundness and efficiency of the proof procedure is discussed below.

Predicates for rewriting equalities and inequalities

$rewrite\_disj(D, D')$
$\leftarrow$ ($D \equiv (C \vee DRest)$
$\wedge_1$ $rewrite\_conj(C, C')$
$\wedge_1$ (($C' \not\equiv$ **false**
$\wedge_2$ $D' \equiv C' \vee Rest$)
$\vee_2$ ($C' \equiv$ **false**
$\wedge_2$ $rewrite\_disj(DRest, D')$))
$\vee_1$ ($D \equiv$ **false** $\wedge_1$ $D' =$ **false**)) [**REW − DIS**]

$rewrite\_conj(C, C')$
$\leftarrow$ ($C \equiv (A \wedge RestC)$
$\wedge_1$ (($literal(A)$
$\wedge_2$ $rewrite(A, A', \Theta)$
$\wedge_2$ $C'' \equiv (A' \wedge (RestC\ \Theta))$
$\wedge_2$ $rewrite\_conj(C'', C')$)
$\vee_2$ ($implication(A)$ $\wedge_2$ $existVars(C, X)$
$\wedge_2$ $rewrite\_implication(X, A, A')$
$\wedge_2$ $rewrite\_conj(Rest, Rest')$
$\wedge_2$ $C' \equiv (A' \wedge Rest')$))
$\vee_1$ ($C \equiv$ **true** $\wedge_1$ $C' =$ **true**)
$\vee_1$ ($C \equiv$ **false** $\wedge_1$ $C' =$ **false**)) [**REW − CON**]

$rewrite\_implication(XVars, I, I')$
$\leftarrow$ $I = ((H \leftarrow B), HP)$
$\wedge_1$ $rewrite\_conj'(XVars, B, B', \Theta)$
$\wedge_1$ $I' = ((H \leftarrow B')\Theta), HP)$ [**REW − IMP**]

$rewrite\_conj'(X, C, C', \Theta')$
$\leftarrow$ ($C \equiv (A \wedge RestC)$
$\wedge_1$ $rewrite'(X, A, A', \Theta)$
$\wedge_1$ $C'' \equiv (A' \wedge (RestC\ \Theta))$
$\wedge_1$ $\Theta' = \Theta \cup \Theta''$
$\wedge_1$ $rewrite\_conj'(X, C'', C', \Theta'')$)
$\vee_1$ ($C \equiv$ **true** $\wedge_1$ $C' =$ **true** $\wedge_1$ $\Theta' = \{\}$)
$\vee_1$ ($C \equiv$ **false** $\wedge_1$ $C' =$ **false** $\wedge_1$ $\Theta' = \{\}$)) [**REW − COX**]

Table 3.6: Top-level predicates for rewriting of equalities

Rewrite Rules	
$rewrite(A, A', \Theta)$	
$\leftarrow$ ($A \neq (\ X\ =\ Y\ )$	
$\wedge_1\ A \neq (\ X\ <\ Y\ )$	
$\wedge_1\ A \neq (\ X\ \leq\ Y\ )$	
$\wedge_1\ A = A'\ \wedge_1\ \Theta = \{\}\ )$	$[OTHERPR]$
$\vee_1$ ($A = (\ X\ =\ Y\ )$	
$\wedge_1$ (($var(X)\ \wedge_2\ var(Y)$	
$\wedge_2\ \Theta = \{X/Y\}\ \wedge_2\ A' =$ **true**)	$[UNIF1]$
$\vee_2$ ($X = F(\overline{W})\ \wedge_2\ Y = F(\overline{Z})\ \wedge_2\ \Theta = \{\}$	
$\wedge_2\ A' = (\overline{W} = \overline{Z})\ )$	$[UNIFI2]$
$\vee_2$ ($constant(X)\ \wedge_2\ constant(Y)$	
$\wedge_2\ X = Y\ \wedge_2\ \Theta = \{\}\ \wedge_2\ A' =$ **true**)	$[UNIFI3]$
$\vee_2$ ($var(Y)\ \wedge_2\ \neg var(X)$	
$\wedge_2\ \neg occurs(Y, X)\ \wedge_2\ \Theta = \{Y/X\}\ \wedge_2\ A' =$ **true**)	$[UNIFI4]$
$\vee_2$ ($var(X)\ \wedge_2\ \neg var(Y)$	
$\wedge_2\ \neg occurs(X, Y)\ \wedge_2\ \Theta = \{X/Y\}\ \wedge_2\ A' =$ **true**)	$[UNIFI5]$
$\vee_2$ ($none\_of\_the\_above1(X, Y)$	
$\wedge_2\ A' =$ **false** $\wedge_2\ \Theta = \{\}\ )\ )$	$[UNIFI6]$
$\vee_1$ ($A = (\ X\ <\ Y\ )$	
$\wedge_1\ constant(X)\ \wedge_1\ constant(Y)$	
$\wedge_1$ (($X < Y\ \wedge_2\ \Theta = \{\}\ \wedge_1\ A' =$ **true**)	
$\vee_2$ ($\neg(X < Y)\ \wedge_2\ \Theta = \{\}\ \wedge_2\ A' =$ **false**)))	$[LESSTH]$
$\vee_1$ ($A = (X < Y)$	
$\wedge_1$ ($var(X)\ \vee_2\ var(Y)\ )$	
$\wedge_1\ \Theta = \{\}\ \wedge_1\ A' = A\ )$	$[SUSINEQ]$
	[REWRITE]

Table 3.7: Rewrite Rules

Rewriting process on implications	
$process\_equalities(ExQVars, C, C')$	
$\leftarrow\ rewrite\_implication(ExQVars, C, C')$	
$\wedge_1\ C \neq C'$	**[PRO – CON]**
$case\_analysis(ExQVars, InCN, Eq, CN_1, CN_2)$	
$\leftarrow\ (InCN \equiv (((\ H \leftarrow\ X = T\ \wedge\ Rest), HP_i) \wedge\ RestC)$	
$\wedge_1\ var(X)$	
$\wedge_1\ X \in ExQVars$	
$\wedge_1$ ($atomic(T)$	
$\vee_2$ ($compound(T) \wedge_2\ \neg occurs(X, T)\ )\ )$	
$\wedge_1\ (no\_contain\_Uni\_Vars(ExQVars, H)\ )$	

Table 3.8: Rewriting implications

Rewrite Rules for atoms in implications	
$rewrite'(ExV, A, A', \Theta)$	
$\leftarrow$ ($A \neq (X = Y) \wedge_1 A \neq (X < Y)$	
$\wedge_1 A \neq (X \leq Y) \wedge_1 A = A' \wedge \Theta = \{\}$)	$[XOTHERP]$
$\vee_1$ ($A = (X = Y)$	
$\wedge_1$ (($var(X) \wedge var(Y)$	
$\wedge_2 \neg(X \in ExV) \wedge_2 \neg(Y \in ExV)$	
$\wedge_2 \Theta = \{X/Y\} \wedge_2 A' =$ **true**)	$[XUNIF1]$
$\vee_2$ ($X = F(\overline{W}) \wedge_2 Y = F(\overline{Z}) \wedge_2 \Theta = \{\}$	
$\wedge_2 A' = (\overline{W} = \overline{Z})$)	$[XUNIF2]$
$\vee_2$ ($constant(X) \wedge_2 constant(Y)$	
$\wedge_2 X = Y \wedge_2 \Theta = \{\} \wedge_2 A' =$ **true**)	$[XUNIF3]$
$\vee_2$ ($var(X) \wedge_2 var(Y)$	
$\wedge_2 \neg(Y \in ExV) \wedge_2 (X \in ExV)$	
$\wedge_2 \Theta = \{Y/X\} \wedge_2 A' =$ **true**)	$[XUNIF4]$
$\vee_2$ ($var(Y) \wedge_2 var(X)$	
$\wedge_2 \neg(X \in ExV) \wedge_2 (Y \in ExV)$	
$\wedge_2 \Theta = \{X/Y\} \wedge_2 A' =$ **true**)	$[XUNIF5]$
$\vee_2$ ($var(Y) \wedge_2 \neg(Y \in ExV) \wedge_2 \neg var(X)$	
$\wedge_2 \neg occurs(Y, X) \wedge_2 \Theta = \{Y/X\} \wedge_2 A' =$ **true**)	$[XUNIF6]$
$\vee_2$ ($var(X) \wedge_2 \neg(X \in ExV) \wedge_2 \neg var(Y)$	
$\wedge_2 \neg occurs(X, Y) \wedge_2 \Theta = \{X/Y\} \wedge_2 A' =$ **true**)	$[XUNIF7]$
$\vee_2$ ($none\_of\_the\_above2(X, Y)$	
$\wedge_2 \Theta = \{\} \wedge_2 A' =$ **false**))	$[XUNIF8]$
$\vee_1$ ($A = (X < Y)$	
$\wedge_1 constant(X) \wedge_1 constant(Y)$	
$\wedge_1$ (($X < Y \wedge_2 \Theta = \{\} \wedge_2 A' =$ **true**)	
$\vee_2$ ($\neg(X < Y) \wedge_2 \Theta = \{\} \wedge_2 A' =$ **false**)))	$[XLESST]$
$\vee_1$ ($A = (X < Y)$	
$\wedge_1$ ($var(X) \vee_2 var(Y)$)	
$\wedge_1 \Theta = \{\} \wedge_1 A' = A$)	$[XSUSINQ]$
	[**REWRITE**$'$]

Table 3.9: Rewrite rules for implications

The programs for rewriting are shown in tables 3.6, 3.7, 3.8 and 3.9. Table 3.6 contains the topmost routines for the rewriting process on atoms in the Δ element of a node. Table 3.8 contains the topmost routines for processing equality and inequality atoms in the body of implications, including case analysis. Tables 3.7 and 3.9 contains the rewrite rules that simulate the unification algorithm for equalities and reduce inequalities. In table 3.9 the rules of unification are extended to cater for universal and existential quantification of variables.

In these descriptions, we take for granted built-in mechanisms for:

1. Identifying whether a term(X is a variable ($var(X)$);
2. Identifying whether the term is grounded ($constant(X)$);
3. Collecting the set S of existentially quantified variables in a given node C ($existVars(C, S)$);
4. Identifying whether a given variable X, occurs within a term T, the occur-check of Robinson'n unification algorithm [Rob79], ($occurs(X, T)$) and
5. Establishing whether a given variable X belongs to a given set S ($X \in S$).

What follows is a description of each clause and sub-clause in the tables, using its label as reference:

- **[REW-DIS]**: Rewriting equalities and inequalities in a node means transforming that node into a new one. The new node must be consistent with what can be deduced from the equalities—inequalities in the original node, and an equality theory (such as Clark Equality Theory [Cla78]) that formalizes syntactic relations between terms.

  This clause describes the topmost program of the rewrite mechanism. From the frontier D (a disjunction), provided as input, the program selects a node C (conceptually a conjunction) and rewrites it into C'. If the new C' is not equivalent to **false**, it is attached to the rest of the original frontier and returned as output. If C' is equivalent to **false**, then the program calls itself recursively to rewrite the rest of the original frontier.

  If the original frontier is equivalent to **false** the program indicates that the new frontier is empty by returning **false** as output.

- **[REW-CON]**: This program searches a node, checking every literal and implication in it. If an equality—inequality literal is found, the program will apply the rules specified below in table 3.7 and collect a *unifier* Θ that must be applied to the rest of the conjunction. ($RestC\Theta$). This ensures that the substitution of variables by terms, implied by the original equality, is actually performed on the other occurrences of the same variables in the node. Observe that because this operation of substitution is potentially performed for every equality in the node, the final effect is the application of a *composition* of unifiers to the whole node.

  As, in general, composition is not commutative (see [Hog90], page 77), if it is not completely executed then it can yield different results for different search strategies, as the following example shows:

  Example 3.4.2 Consider a conjunction A such that $A = (X = a \wedge p(X) \wedge X = b)$. This could be partially rewritten as $A = (X = a \wedge p(a) \wedge X = b)$ or as $A = (X = a \wedge p(b) \wedge X = b)$, depending on the selection strategy. But, if the rewriting is completed, it would end with $A \equiv$ **false** in both cases.

This is the reason to make of the rewrite process an atomic, non-interruptible process, with, in addition, the highest priority within the agent's reasoning mechanism.

If the selected conjunct (A) is a literal and is not an equality or disequality, then no alteration of the node takes place.

On the other hand, if the selected conjunct in C is an implication, a special routine *rewrite_implication* will be called to rewrite equalities and inequalities in the body of the implication. $existVars(C, X)$ is called to collect all the variables in the node before rewriting of the body can proceed.

Note that, as rewriting of equalities in implications is carefully controlled by the higher level programs of the proof procedure (see table 3.5), rewriting of implications need not be part of this clause [REW-CON]. It is left here to emphasized that rewriting must take place on literal and implications.

- **[REW-IMP]**: This is the definition of the rewriting of implications. Notice that the program must collect the unifier Θ to apply it to the head of the implication This propagates the instantiations of variables cause by the rewritings in the body of the implications.
- **[REW-COX]**: This program is similar to [REW-CON] except that it only deals with conjunctions of literals (no implications) and it calls a routine that rewrites equalities taking quantification of variables into consideration. This routine is described below. This program is, as one can deduce from the description of the inference rules, devoted to process equalities in the body of implications.

The program in table 3.7 contains the code that rewrites equalities in Δ. Its components are:

- **[OTHERPR]**: This sub-clause *skips over* those atoms that are not equalities or inequalities.
- **[UNIFI1]** .. to **[UNIFI6]** implement Robinson's unification algorithm [Rob79]. Observe that this meta-level description uses some new notational devices. In $X = F(\overline{X}) \wedge X = F(\overline{Y})$, $F(\overline{X})$ is syntactic sugar for $term(F, \overline{X})$, where F and $\overline{X}$ are term themselves (the name of the compound term and its list of arguments, respectively). The constructed term is used to indicate that the functions in the equalities have the same name. Recall that $\overline{X}$ denotes a vector with all the arguments of the function. Thus, $\overline{X} = \overline{Y}$ is an abbreviation for the conjunction of the equalities between arguments in the same position in each vector. Of course, that conjunction can only be built if both vectors have the same number of elements. Otherwise $(\overline{X} = \overline{Y})$ will fail.
- **[LESSTH]** reduces inequalities between known values, while
- **[SUSINEQ]** avoids any commitment on inequalities where at least one of the terms in unknown.

To support the processing equalities inside an implication (tables 3.4) Table 3.8 contains:

- **[PRO-CON]**: This is the topmost program for processing of equalities and inequalities in the body of implications. The added functionally with respect to *rewrite_implications* is a test to detect whether the incoming implication C is rewritten into a different C'. This test is a control device. It is used by the program to decide that equality rewriting has been exhaustively applied.

- **[CAS-ANA]** This clause describes the application of the rule for case analysis of equalities. Whenever an implication contains an equality such as $X = T$, where X is an existentially quantified variable and T is an atomic or "compound" (a function) term, case analysis is applicable. The rule also tests that in the case of a "compound" term this does not contain the variable X. Another condition is that the head of the implication must not contain universally quantified variables.

Finally, the program in table 3.9 contains the code that rewrites equalities in the body of implications in CN. Its components are:

- **[XOTHERP]**: This sub-clause, as [OTHERPR] above, *skips over* those atoms that are not equalities or inequalities.
- **[XUNIF1]** .. to **[XUNIF8]** implement an extension of Robinson's unification algorithm [Rob79] that caters for variables with existential and universal quantification. Basically, normal unification is performed on those terms that contain universally quantified variables, while it is "suspended" for those terms containing existentially quantified variables. Unification involving the latter is only possible after to assignments in Δ, outside the implication, have been considered.

  Observe that the notational devices introduced in table 3.7 are also used here. Major assumptions in this implementation are that ExV contains all the existentially quantified variables in the node and that *any variable not in this set (ExV) is universally quantified.* That explains the use of the operator $\in$ in **[XUNIF4]** ... **[XUNIF7]**.
- **[XLESST]**, as [LESSTH] in table 3.7, reduces inequalities between known values, while
- **[XSUSINQ]** avoids any commitment on the truth of inequalities where at least one of the terms in unknown.

This concludes the description of the logic programs that implement the **iff** proof procedure.

3.4.5 The special treatment of inequalities

The special treatment of inequalities $<$, implemented by [LESSTH], [SUSINEQ] and their analogues in table 3.9, has an important effect on the soundness results obtained by Fung [Fun96] for the **iff** proof procedure.

The net effect of these extensions is to yield an implementation that *is not sound*, because the prover will end up with *pseudo-leaf* nodes to which it can not apply any inference rule. For the program, these *pseudo-leaf* nodes will be undistinguishable from real *leaf* node. Yet the *pseudo-leaf* could contain $<$ atoms that are being prevented from processing by the factoring and propagation rules.

Example 3.4.3 The node $Node \equiv 2 < W \wedge W < 2 \wedge$ (**false** $\leftarrow$ $X < Y \wedge Y < X$ would be offered by the algorithms in this chapter, as a leaf node. This is because we are refraining from propagating $<$.

So, in order to justify the use of these extensions, one must recall the following points:

1. Our implementation of the proof procedure is forced to ignore the strong notion of *soundness* proved by Fung [Fun96]. The reason for this is the parameter R in *demo* and the other programs. As the execution of the prover is resource bounded, the system is doomed to produce potentially incorrect answers when it consumes all its resources before finishing

a proof. In these cases, there would still be some rules of inference to be applied, but the prover would ignore them because of the lack of time or space for further computations.

Nevertheless, the prover still preserves a **weak** notion of soundness, expressed by the fact that any pair of frontiers F_i and F_j in a derivation always satisfies:

$$T \cup CET \cup IC \models F_i \leftrightarrow F_j \tag{3.27}$$

This *weak* soundness is not affected by the special treatment of inequalities, as these can be seen as goals still to be processed.

2. Any useful account of the relation $<$ (or $\leq$) includes the transitivity rule: $T < T' \leftarrow T < T'' \wedge T'' < T'$. If one chooses to consider $<$ was a normal predicate with a definition containing that rule, then when invoked with variables as parameters, this rule could plunge the prover into a loop, consuming resources in unuseful computations. Very little is gained if the predicate is considered as an abducible and the rule is added as an integrity constraint. In this case, even though there would be no loop, the use of propagation will not be adding useful information in most cases. What one needs is a mechanism that uses the transitivity rules and other rules over $<$ *just when it is necessary*. We describe one of such mechanisms in chapter 6.

3. So, the main reason for the selective treatment of inequalities is to avoid too early commitments on the ordering of variables with unknown absolute values. This is particularly useful in planning applications, when $<$ refers to the ordering between time points. As has been pointed out by Denecker et al [DMB92], the use of an explicit *linear order theory*, describing the relationships between points in a linear order, can cause inefficiency in a proof procedure. To deal with this kind of inefficiency, Fung introduces a specialization of the proof procedure (**iff-LO**) that combines a special form of propagation (**resolving**, see [Fun96] pg. 123) with a linear order theory LINORD, containing the transitivity rule above, among others. Fung then proves that **iff-LO** is sound using a new definition of *answer* that extends a non-failure, leaf node $< \Delta, UC, CN >$ with all the possible *linearizations*[11] of the set of time-points in the node.

   These extensions, especially the new definition of an answer, can be useful in the planning problem. However, we consider a more pragmatic approach to deal with time ordering for planning in the reactive agent architecture, which is discussed in chapter 6.

3.5 Examples of the proof procedure at work

To illustrate the way the proof procedure works, we show some solutions produced by a PROLOG implementation of the programs above:

3.5.1 The faulty lamp example

This example is shown by Fung and Kowalski in [FK96], where they do all the computations manually. Below there is the description of the example and the output of program to answer the query. Notice that we have simplified some of the definitions. The programs take a PROLOG database as input. Also, notice that the predicate $=$ is written as **eq**.

[11] a linearization is a particular complete ordering of all the points in a set, i.e. if the set contains the points t_1 and t_2, the two possible linearizations are $t_1 < t_2$ and $t_2 < t_1$.

Example 3.5.1 There is a fault in the lamp when the lamp is broken or when there has been a power failure and there is no backup battery. A backup battery is available when the battery has a nonempty deposit. The system knows about a lamp "a" and a battery "b" with deposit "c".

$$
\begin{array}{ll}
T: & lamp(a) \\
 & battery(b,c) \\
 & faulty\_lamp \leftrightarrow lamp(X) \wedge broken(X) \\
 & \qquad \vee\ power\_failure(X) \wedge \neg backup(X) \\
 & backup(X) \leftrightarrow battery(X,Y) \wedge \neg empty(Y) \\
Ab: & \{broken, power\_failure, empty, eq\} \\
IC: & \{\} \\
Query: & faulty\_lamp
\end{array}
$$

The program's output is:

```
%Query:  demo(40, rule_app, true, [[true,(faulty_lamp, true), true,
 [],[]],L).

Frontier 1

 % UC1 = { | lamp(G2904), broken(G2904) };     CN1 = {  }

 % UC2 = { | power_failure(G2840), not backup(G2840) }; CN2 = {  }

Frontier 2

 % UC1 = { | G2904 eq a, broken(G2904) };     CN1 = {  }

 % UC2 = { | power_failure(G2840), not backup(G2840) }; CN2 = {  }

Frontier 3

 % UC1 = { broken(a) | };     CN1 = {  }

 % UC2 = { | power_failure(G2840), not backup(G2840) }; CN2 = {  }

% L = [[(broken(a), true),true,true,[],[]],
       [true,(powerfailure(G2840), not backup(G2840), true),
        true,[],[]]]
```

Thus, for the program, the "first" explanation of the fault in the lamp is that only lamp "a" is broken. There are alternative explanations that have not been fully explored at this stage because this agent is performing a kind of depth-first search.

3.5.2 Reasoning about the elevator position

One can define a theory to be used by the elevator controller to reason about its own position in the building, with the programs here described. Although this is not exactly reasoning

for planning, there are certain analogies between this mode of reasoning and that used by the planner, discussed later in chapter 6. This is also a simple introduction to the kind of formalisations discussed in the following chapter. (Notice that $<$ and $\leq$ are written as "lt" and "le", respectively).

Example 3.5.2 The elevator is at floor "X" at time "T" if it moved (up or down) to that floor at some earlier time "T_1" and it has not moved since. The elevator moves away from location "X" in the interval between "T_1" and "T_2", if it goes (up or down) to a different floor "Y" at some time "T" between "T_1" and "T_2".

$$
\begin{array}{ll}
T: & at(X,T) \leftrightarrow do(up(X),T_1) \wedge T_1\ le\ T \wedge \neg move(T_1,X,T) \\
 & \quad \vee\ do(down(X),T1) \wedge T_1\ le\ T \wedge \neg move(T_1,X,T) \\
 & move(T_1,X,T_2) \leftrightarrow do(up(Y),T) \wedge \neg(XeqY) \wedge T_1\ le\ T \wedge T\ lt\ T_2 \\
 & \quad \vee\ do(down(Y),T) \wedge \neg(XeqY) \wedge T_1\ le\ T \wedge T\ lt\ T_2 \\
Ab: & \{do, lt, le, eq\} \\
IC: & \{\} \\
Query: & at(1,3)
\end{array}
$$

The program's output is:

```
%Query2: demo(40, rule_app, true, [[true,(at(1,3), true), true,
               [],[]]],L).

Frontier 1

 % UC1 = { | 1 eq G2872, 3 eq G2876, do(up(G2872), G2880),
   G2880 le G876, not move(G2880, G2872, G2876) };   CN1 = {  }

 % UC2 = { | 1 eq G2744, 3 eq G2748, do(down(G2744), G2752),
   G2752 leG2748,  not move(G2752, G2744, G2748) };  CN2 = {  }

Frontier 2

 % UC1 = { do(up(1), G2880),  G2880 le 3 |  };  CN1 = {  [] if
   (move(G2880, 1, 3), true)  }

 % UC2 = { | 1 eq G2744, 3 eq G2748, do(down(G2744), G2752),
   G2752 leG2748, not move(G2752, G2744, G2748) };   CN2 = {  }

Frontier 3

 % UC1 = { do(up(1), G2880),  G2880 le 3,  |  };  CN1 = {
   [1 eq 1] if  (G2880 le G14268, G14268 lt 3, true),
   [] if (do(up(1), G14268),  not 1 eq 1, G2880 le G14268,
   G14268 lt 3, true),  [] if (do(down(G14104),  G14108),
   not 1 eq G14104, G2880 le G14108,  G14108 lt 3, true)  }

 % UC2 = { | 1 eq G2744,  3 eq G2748,  do(down(G2744), G2752),
   G2752 le  G2748,  not move(G2752, G2744, G2748),  };  CN2 = {  }
```

To explain its position (the 1rd floor) at time 3, without further information, the agent assumes that it is due to an earlier movement upwards. Observe that, once again, this is only one among 2 possible explanations, one of which (UC2) is still to be further explored.

Chapter 4 explains how to go beyond this kind of simple reasoning to more complex forms of reasoning for planning. There we also describe a logical platform to program agents with a reasoning mechanism as formalised in this chapter.

Chapter 4

An Agent oriented Programming Language and Knowledge Representation

So far, we have described the architecture of an agent and how that architecture processes and transforms the agent's knowledge and goals. We formalised the agent's architecture in chapter 2 and the agent's reasoning mechanism in chapter 3, using the language presented in the introductory chapter 1. Because the agent's knowledge and goals can themselves be described in a logical language, and we have been describing how these descriptions change, we can say that the language used in the previous chapter is a *meta-language* (for the object language in which the agent's knowledge and goals are represented). In this chapter the attention is switched to the *object language* to describe problems about which the agent reasons.

The leading intention in this chapter is to *program an agent to solve a particular set of problems*. Recall that a preliminary attempt at programming the agent has already been made. In section 2.3.2 of chapter 2, a set of *integrity constraints* was presented and it was briefly explained how they could be used to provide a first solution for the elevator controller, our benchmark example. It could be said that integrity contraints, with the structure prescribed by [ICGRAMM] in chapter 2, already constitute an *object level language* for agent programming.

When designing a language for knowledge representation and agent programming, one should consider what Muggleton and Michie has called the *duality principle*[1]. Sometimes a declarative description of "what" the problem is all about can be the best input to give to the agent that will try to solve it. At other times, listing a *recipe* of "how" to solve the problem is the most straightforward way of conveying the knowledge required by the solver. Indeed, logic programming is an attempt to combine both types of descriptions in the same formalism, so giving the programmer the means to cover the spectrum of possibilities between the procedural and the declarative extremes of knowledge representation. This flexibility of logic programming can be very useful in agent programming, where one wants the agent *to plan* the solution of a given problem but where one does not want to provide a detailed and customized account of the problem's context and relevant procedures. That is, one wants the agent to "deduce" what

[1] Muggleton and Michie state the duality principle as follows:

> "Software involved in human-computer interaction should be designed at two interconnected levels: a) a declarative, or self-aware level, supporting ease of adaptation and human interaction and b) a procedural, or skill level, supporting efficient and accurate computation" [MM96]

.

the relevant solution is.

It is interesting to observe how this dilemma of *procedures vs. definitions* (imperative vs. declarative descriptions) takes many different forms. In [AC90], Agre and Chapman explain that "what plans are like depends on how they are used". They contrast two views of plan use: "On the plans-as-program view, plan use is the execution of an effective procedure. On the plan-as-communication view, plan use is like following natural language instructions [..] it requires an account of *improvisation*." (.ibid, their italics). One can find similarities between this discussion and one of the ideas advanced in logic programming: *algorithm = logic + control* [Kow79b], [Kow79a]. One wants the programmer/instructor (who knows about certain domains of knowledge) to provide the logic of the problem, i.e. a general description or set of instructions that *any* agent could accept and understand. The agent (machine or human) receiving the instructions should then build and execute a particular plan, using the control strategy that best suits the circumstances at execution time (i.e. improvisation). The control strategies must have been fed to the agent's *common- sense* knowledge base in advance, perhaps as part of the development of the agent's basic architecture.

Thus, the multifaceted problem of agent programming seems to require a very rich language. In this chapter and in the following, a family of languages for agent programming is introduced. The family of languages is intended to provide a programming platform flexible in various ambitious respects. First, the platform should allow for trade-offs between declarative and procedural knowledge, in principle by supporting both kinds of programming descriptions. It should also support the specification of diverse control strategies by means of domain-specific heuristics. This means that the agent's control strategies could be tailored to particular domains to have a more efficient reasoner. Finally, it should allow the programmer to describe efficient strategies for problem solving, along the lines of *knowledge-based reactivity* (defined in chapter 1, section 1.3.3), as opposed to exhaustively performed means-end analysis. We start the presentation of the family of languages in this chapter, with the introduction of a structured, logic-programming language: OPENLOG.

4.1 OPENLOG: from structured to logic programming

A program can be seen as a scheme that an agent uses to generate plans to achieve some specified goal. These plans ought to lead that agent to display an effective, goal-oriented behaviour that, nevertheless, caters for changes in the environment due to other independent processes and agencies. This means that, although the agent would be following a well-defined program, it would stay *open* to the environment and allow for changes in its circumstances and the assimilation of new information generated by these changes.

In the following, a well-known programming language (STANDARD PASCAL) is extended with language constructs to support this kind of *open* problem-solving and planning. The semantics of the resulting language (OPENLOG) is based on a non-monotonic logic of actions and events that caters for input assimilation and reactivity. In combination with the reactive architecture described in chapter 2, where the interleaving of planning and execution is clearly defined, the language supports a solution to the problem of agent specification and programming.

OPENLOG supports the principle of **progression** for the decomposition of goals into subgoals. In this mode of planning, the first action to be performed is generated first[2]. Proper heuristic information can help the agent to choose an action that is appropriate for the achievement of its goals. We say that this strategy is reactive because it allows the agent to start

[2]as opposed to *regression* "where the first action to be performed is generated last" [Sha96].

acting within a short period of time, even if it has not completed a plan to achieve its goals. This strategy fits nicely in an agent's architecture where planning can be interrupted at *any time* to be interleaved with execution and sensing, as described in [Kow95] and in chapter 2.

OPENLOG is aimed at the same applications as the language GOLOG of Levesque *et al* [LRL+95] i.e. agent programming. Our approach differs from Levesque *et al*'s in that there is no commitment to a particular logical formalism. One can employ the Situation Calculus or the Event Calculus depending on the requirements of one's architecture. However, the Event Calculus has turned out to be more expressive and useful for the reactive architecture described in chapter 2.

Like GOLOG, our approach also regards standard programming constructs as macros. However, here they are treated as special predicates or terms[3]. There is no problem with recursive or global procedures. Procedures are like predicates that can be referred to (globally and recursively or non-recursively) from within other procedures. Interpreting these macros is, in a sense, like translating traditional structured programs into normal logic programs.

To present the language, the information in this chapter is organized as follows: Section 4.2 describes the syntax of the language which is, basically, a subset of PASCAL [WJ84] extended with operator for parallel execution. Section 4.3 explains the semantics of OPENLOG by means of a logic program. This novel strategy to describe the semantics of a programming language is also discussed. We interrupt the description of the OPENLOG to introduce in section 4.4, *background theories*, the temporal reasoning platform on which the whole family of languages in this thesis is based. Section 4.8 resumes the presentation of OPENLOG and the background theories by showing how they can be applied to the problem of the elevator controller. Subsections 4.8.1 and 4.8.2 explain how to program policy 1 and 3 of the elevator example into the elevator controller. Finally, section 4.10 compares our solution with GOLOG's and discusses related work.

4.2 The Syntax of OPENLOG

The syntax of OPENLOG is described in BNF[4] form in table 4.1.

The syntax is left "open" to accommodate, in suitable syntactic categories, those symbols designated by the programmer to represent *fluents, primitive actions* and *complex actions*. These notions are part of the semantics and are explained below. In addition to the syntactic rules, the system must also provide translations between the "surface syntax", that the programmer will use to write *queries*, and the underlining logical notation.

In this initial formalization, PASCAL syntax is *limited* to the least number of structures required for structured programming: (*";", "if.. then.. else..", "while"*). On the other hand, the syntax supports the representation of parallel actions through the compositional operators *par* [5] and + [6].

[3] See *[DN01]* in table 4.1 below, **proc** can be regarded as a two-argument predicate, the following symbol is a term, and **begin** and **end** are bracketing a more complex term.

[4] In the table, S_j means an instance of S of sub-type j. (A)* indicates zero or more occurrences of category A within the brackets.

[5] Unlike those semantics of interleaving [Hoa85], [Mil89] this is a form of real parallelism. Actions start simultaneously, although they can finish at different times. Notice that when all the actions have the same duration (or when they all are "instantaneous") this operator is equivalent to +. Also, observe that the *cycle* predicates in [Kow95] and in chapter 2 only handle actions which last for one unit of time. We relax this limitation in chapter 6.

[6] used as well to express real parallelism. Actions start and finish at the same time. This allows the programmer to represent actions that interact with each other so that the finishing time of one constraints the finishing time of the other. For instance, taking a bowl full of soup with both hands and avoiding spilling [Sha97].

Table 4.1 OPENLOG: Syntax			
Program	::=	*Proc (Program)**	*A program*
Proc	::=	**proc** $Func_{proc}$	
		begin *Commands* **end**	*Procedure definition*
Block	::=	**begin** *Commands* **end**	*Block*
Commands	::=	*Block*	*Block call*
	|	$Func_{proc}$	*Procedure call*
	|	$Func_{action}$	*Primitive action call*
	|	*Commands* **;** *Commands*	*Sequential composition*
	|	*Commands* **par** *Commands*	*Parallel composition*
	|	*Commands* **+** *Commands*	*Strict parallel composition*
	|	**if** $Expr_{boolean}$ **then** *Commands*	*Test*
	|	**if** $Expr_{boolean}$ **then** *Commands*	
		else *Commands*	*Choice*
	|	**while** $Expr_{boolean}$ **do** *Block*	*Iteration*
Query	::=	...	*Logical expressions*
$Expr_j$	::=	$Func_j$*(Func, Func, ..., Func)*	*Expressions (as function applications)*
Func	::=	$Func_{proc}$	
	|	$Func_{action}$	
	|	$Func_{fluent}$	
	|	$Func_{boolean}$	*Functors*
$Func_{proc}$	::=	*serve(Term), build(Term), ...*	*User-defined complex actions or procedures' names*
$Func_{action}$	::=	**nil**	*Null action*
	|	*up* | *move(Term, Term)* | ...	*User-defined primitive actions' names*
$Func_{fluent}$	::=	*at(Term)* | *on(Term,* $Func_{fluent}$ *)* | ...	*User-defined fluents*
$Func_{boolean}$	::=	**and**($Func_{fluent}$, $Func_{boolean}$)	
	|	**or**($Func_{fluent}$, $Func_{boolean}$)	
	|	**not**($Func_{boolean}$)	
	|	$Func_{fluent}$	*Boolean functions*
	|	*Query*	*Tests on "rigid" information*
Term	::=	*Ind* | *Var*	*Terms can be individuals or variables*
Ind	::=	...	*Individuals identified by the user*
Var	::=	...	*Sorted Variables*

Table 4.1: The Syntax of OPENLOG.

4.3 The semantics of OPENLOG

4.3.1 Comments on the semantics of programming languages

In the history of Computer Science, three clear "strands" of methods to specify the semantics of programming languages can be identified [Win93]. These are 1) *operational semantics*, which "describes the meaning of a programming language by specifying how it executes on an abstract machine" (.ibid); 2) *denotational semantics* [Sto77], which "uses the more abstract mathematical concepts of complete partial orders, continuous functions and least fixed points" [Win93] to create a *Domain Theory* that defines and "grounds" the meaning of the language; and 3) *axiomatic semantics* that "tries to fix the meaning of a programming construct by giving proof rules for it within a program logic" (.ibid).

Denotational semantics have been criticized for being too difficult to apply to real programming language specification [Mos92]. A similar type of semantics based on abstract concepts has been advanced as a more pragmatic solution, this time using "actions" as the fundamental notion. It is called *action semantics* and has been pioneered by Mosses [Mos92].

To define the semantics of OPENLOG, we use a strategy that does not correspond directly to any of these "strands", although it has a close relationship with each of them, especially with the axiomatic semantics. We specify the semantics of the language by means of a logic program. That is, we define a transformation from programs in OPENLOG to normal logic programs (logic programs including negation), for which well-understood semantics already exists. Any semantics attributed to the logic program that "defines" OPENLOG, can then be regarded as a semantics for the programs written in this language.

One advantage of this novel approach to semantics is that one can build on the existing results for the semantics of logic programs. It may seem that relying on "the" semantics of logic programming implies that one will have to choose from at least 49[7] existing formalisations or perhaps to advance a new one. However, recent results, showing the convergence and coincidence of different semantics encourage the idea of using logic programs to define the semantics of other programming languages.

Research on the semantics of logic programs has had as a main focus one key aspect of any logic framework: the inference of negative information. The inclusion of negative literals into logic programs disables the usual *denotational* approach to semantics in logic programming. The reason for this is that with negative literals in the formalism, the abstract mathematical concepts of Scott-Strachey's semantics are not well defined and their "functions" are not well-behaved. However, the problem of negative information can be isolated from the problem of defining a semantics for a programming language, as our definition below shows.

Another advantage of our approach is that our "semantics" also specifies an interpreter for the language. Thus, the operational semantics of OPENLOG programs takes the the form of an interpreter for an abstract machine.

We now proceed to the definition of the logic program that specifies the semantics of OPENLOG.

4.3.2 A semantics and an interpreter for OPENLOG

The semantics of the language is stated[8] in table 4.2 by means of the predicate *done*[9]. The definition of *done* can also function as an interpreter for the language. Informally, $done(A, T_o, T_f)$ reads "An action of type A is started at T_o and completed at T_f". As we explained above,

[7] As counted by Reiter (reported by personal communication).

[8] PROLOG-like syntax is being used.

[9] The definitions of *rigid*, *nonrigid* and other predicates are also required but are not problematic.

Table 4.2 OPENLOG : Semantics and implementation

$done(Pr, T_o, T_f)$	$\leftarrow$ **proc** Pr **begin** C **end** $\wedge\ done(C, T_o, T_f)$	[**DN01**]
$done((\ C_1\ ;\ C_2), T_o, T_f)$	$\leftarrow\ done(C_1, T_o, T_1)\ \wedge\ T_1 \leq T_2$ $\wedge\ done(C_2, T_2, T_f)$	[**DN02**]
$done((\ C_1$ **par** $C_2), T_o, T_f)$	$\leftarrow\ done(C_1, T_o, T_1)\ \wedge\ done(C_2, T_o, T_f)$ $\wedge\ T_1\ \leq\ T_f$ $\vee\ done(C_1, T_o, T_f)\ \wedge\ done(C_2, T_o, T_1)$ $\wedge\ T_1\ <\ T_f$	[**DN03**]
$done((\ C_1 + C_2), T_o, T_f)$	$\leftarrow\ done(C_1, T_o, T_f)\ \wedge\ done(C_2, T_o, T_f)$	[**DN04**]
$done((\textbf{if}\ E\ \textbf{then}\ C_1), T_o, T_f)$	$\leftarrow\ holdsAt(E, T_o)\ \wedge\ done(C_1, T_o, T_f)$ $\vee\ \neg holdsAt(E, T_o)\ \wedge T_o = T_f$	[**DN05**]
$done((\textbf{if}\ E\ \textbf{then}\ C_1$ **else** $C_2), T_o, T_f)$	$\leftarrow\ holdsAt(E, T_o)\ \wedge\ done(C_1, T_o, T_f)$ $\vee\ \neg holdsAt(E, T_o)\ \wedge\ done(C_2, T_o, T_f)$	[**DN06**]
$done((\textbf{while}\ \exists L\ (E_b(L)\ \textbf{do}\ B(L))), T_o, T_f)$	$\leftarrow\ (\neg\exists L\ holdsAt(E_b(L), T_o)$ $\wedge\ T_o\ =\ T_f)$ $\vee\ \ (holdsAt(E_b(L'), T_o)$ $\wedge\ done(B(L'), T_o, T_1)$ $\wedge\ T_o < T_1$ $\wedge\ done((\textbf{while}\ \exists L\ (E_b(L)\ \textbf{do}\ B(L))\ ), T_1, T_f))$	[**DN07**]
$done((\textbf{begin}\ C\ \textbf{end}), T_o, T_f)$	$\leftarrow\ done(C, T_o, T_f)$	[**DN08**]
$done(\textbf{nil}, T_o, T_o)$		[**DN09**]
$holdsAt(\textbf{and}(X, Y), T)$	$\leftarrow\ holdsAt(X, T)\ \wedge\ holdsAt(Y, T)$	[**DN10**]
$holdsAt(\textbf{or}(X, Y), T)$	$\leftarrow\ holdsAt(X, T)\ \vee\ holdsAt(Y, T)$	[**DN11**]
$holdsAt(\textbf{not}(X), T)$	$\leftarrow\ \neg holdsAt(X, T)$	[**DN12**]
$holdsAt(X, T)$	$\leftarrow\ nonrigid(X)\ \wedge\ holds(X, T)$	[**DN13**]
$holdsAt(Q, T)$	$\leftarrow\ rigid(Q)\ \wedge\ Q$	[**DN14**]
$nonrigid(X)$	$\leftarrow\ isfluent(X)$	[**DN15**]
$rigid(X)$	$\leftarrow\ \neg isfluent(X)$	[**DN16**]

Table 4.2: The Semantics of OPENLOG.

due to the fact that the definition of *done* is a logic program, any semantics of normal logic programming can be used to give meaning to OPENLOG programs.

The only command-type that requires further clarification is *while*. L in **[DN07]** stands for the list of variables that appears in expression E_b and nowhere else in the procedure that contains the *while* instruction[10]. What the axiom states is that these variables (appearing only in the expression) must be considered existentially quantified. And any "unfolding" of the *while instruction* will make use of expressions $E_b(L')$ and $B(L')$ which are obtained by renaming the aforementioned variables in $E_b(L)$ and $B(L)$, respectively.

The definition of semantics in table 4.2 needs to be completed with a "base case" clause for the predicate *done* and the definition of *holds*. These two elements are also part of the semantics, but more important, they are the key elements of a *background theory* $\mathcal{B}$.

4.4 Background theories

Roughly, a *background theory* ($\mathcal{B}$) is a formal description of actions and properties and the relationships between action-types and property-types. It also provides the *historical* context in which actions (commands of the programming language) will be executed and goals will be achieved.

A background theory consists of two sub-theories: A set of *domain independent axioms* (DI-$\mathcal{B}$) (notably the base case of *done* and the definition of *holds*) stating how actions and properties interact. These domain independent axioms also describe how persistence of properties is cared for in the formalism.

The other component of the background theory is a set of *domain dependent axioms* (DD$\mathcal{B}$), describing the particular properties, actions and inter-relationships that characterize a domain of application (including the definitions of the predicates *initiates*, *terminates* and *isfluent*).

The semantics for OPENLOG and DD$\mathcal{B}$ can be isolated from the decision about what formalism to use to represent actions and to solve the frame problem (the problem of persistence of properties) in DI$\mathcal{B}$. The formulations presented in the following sections are alternatively based on the Situation Calculus[11] (section 4.5) and on Event Calculus [KS86] (section 4.6). Probably, the most important element in DI$\mathcal{B}$ is the definition of the *temporal projection predicate: holds*(P, T), which reads: *fluent* P *holds at* T [MH69].

The denotation of T depends on the underlying formalism in DI$\mathcal{B}$. One can say that T refers to the state of the universe (including, of course, the computing system). In this case events are state-transitions, and every event causes a state change and every state is caused by an event. The prime notion is then that of an **state**. Alternatively, one can say that T is a **time-point** and that time progresses independently of states. Events and states are indexed by time. Events initiate and terminate properties (fluents, as shown below) and when one refers to the *state at time* t_o, one is referring to all the fluents that hold at that time. Two states can still be identical if what holds in one also holds in the other, but each one is uniquely associated with a time point. In this case, the prime notion would be that of a **time point**.

In the context of the semantics of the programming language, a background theory determines a kind of *assertion language* [Win93] that could be used to make assertions about the state of computation before and after the execution of each command/instruction in a program.

Moreover, this assertion language could be used to *state the meaning* of each type of command as defined by the syntax of the programming language. This is done by associating the

[10] The symbol $E_b(L)$ is used, instead of E_b alone, to highlight the fact that the variables in L appear only in the expression E_b. The same applies for $E_b(L')$, $B(L)$ and $B(L')$.

[11] with certain sacrifice in expressiveness, however. The operators + and par would have to be excluded from the language as it is.

syntactic constructs with specific axioms and a set of rules to reason about the correctness of programs, using the axioms. This is the approach to the semantics of programming language known as *axiomatic semantics*. It was pioneered by Floyd [Flo67] and Hoare [Hoa69], who devised a set of rules to prove the correctness of programs.

As we said above, in *axiomatic semantics*, the designer of a programming language "captures" the meaning of the programs in the language by precisely stating how to prove that every instruction in a given program is correct. An instruction/command is correct if, and only if, its associated assertion actually holds. The assertion states that **if** the *preconditions* of the command (action) hold **then** its *postconditions* also hold. This kind of assertion is normally written (in an extension of the assertion language known as the *correctness assertion language*) as ([Win93], [Gor88]):

$$\{A\}\ P\ \{B\} \tag{4.1}$$

where A describes the preconditions and B the postconditions for a program P to be *partially correct*. This means that if P is executed in a configuration satisfying A *and it terminates*[12], then the state (of the computing system) after the execution must satisfy B. A and B are sentences in the original assertion language which, in our context, would be the language of the background theory.

Hoare axiomatic approach has been criticized by many including Mosses [Mos92] who says that its pragmatic aspects are poor and that "the semantics of a statement S [..] essentially consists of all pairs (P, Q) of conditions such that $\{P\ \}\ S\ \{Q\}$ is provable - *a somewhat inaccessible entity*" (.ibid, the italics are ours). It is understandable that the pragmatic aspects of Hoare logic are poor. It is a logic. It is trying to capture the meaning of programming expressions without being biased towards a particular computer architecture or implementation. And the entities that provide the semantics of an statement are inaccessible so long as there is no effective procedure to compute the provability relation of the logical language.

Another criticism to the axiomatic approach came from C. Moss. In his Phd thesis [Mos81], Moss presented a set of axioms similar to the one presented in this chapter. His intention was, as is our here, to define the semantics of a programming language (a subset of ALGOL in his case) in terms of logic programs. After praising the axiomatic approach for being superior for proving properties of programs and developing correct programs, Moss concluded that it "fails to give a complete answer" because: 1) The approach only supports YES/NO queries about what the execution of a program does and 2) "To specify the meaning of a program having a loop, it is necessary for the programmer to supply an inductive *invariant* assertion [..]"(.ibid).

We disagree with criticism 1) because, as we will show below (in the proof of proposition [ELEVA]) an axiomatisation (such as the one above in table 4.2), can be used to obtain "traces" of the execution of the program (that we call plans) on which a more complex analysis can be based. For that, of course, we need a new kind of proof procedure. As we discussed in chapter 3, an abductive proof procedure can produce more informative answers than YES or NO. Whether this type of procedure can produce the "complete answer" indicated by Moss requires more investigation. But the axiomatization can indeed produce a more informative answer about what the execution of a program does.

We also have to disagree, to some extent, with criticism 2). One certainly has to provide some sort of invariant assertion for the loops. But this invariant does not have to be specific for every instance of a loop. It can be stated, for example, as part of the semantics of the *while* instruction, to be used for every possible instance of this instruction. We have done that in table 4.2. Axiom [DN07] defines the invariant $\exists(E_b(L)\ \mathbf{do}\ B(L))$ which is used to state that the looping will persists while there is an instance of $E_b(L)$ for which $B(L)$ should be attempted.

[12] To claim "total correctness" the program must be guaranteed to terminate.

We have addressed these criticisms in an attempt to support the argument that logical (axiomatic) semantics can produce useful semantics for programming languages. We complete our axiomatisation in the following sections showing alternative languages to describe background theories. The resulting background theories are more than systematic assertion languages for programming constructs. They can be regarded as general *theories of action and change* that describe how a given universe evolves.

4.5 Background theories in the Situation Calculus

The Situation Calculus (SC) [MH69] has been the preferred formalism in some sub-fields of AI (planning, common-sense and non-monotonic reasoning) for quite a long time. As explained in the introduction, one reason for this could be the naturalness with which a problem can be described as a relation between two "situations". Work on the SC led to the discovery of the frame problem [MH69] and many efforts have been made since to improve the flexibility of the language (see [GLR90]) and its associated ontology. This ontology assumes the existence of *properties that change as time passes*, known as **fluents**. There are, therefore, terms in the language denoting fluents. The term P in $holds(P,T)$ denotes a fluent. Also, the language has terms to denote "situations" which were apparently originally regarded as snapshots of the universe ([MH69] section 2), but that have recently been re-interpreted as "histories" (See, for instance [Rei96]). T in $holds(P,T)$ could be said to refer to a situation in either of these senses.

The other important terms in the language denote *actions* that *cause* fluents to change. The results of these changes are described by logical sentences involving *holds*, the temporal projection predicate.

4.5.1 The temporal projection predicate in SC

With the temporal projection predicate one can establish which properties hold and which do not hold throughout the history of a problem. A history is a record of changes of the fluents in that part of the universe relevant to the problem. How to represent such historical data and how to reason with it are the two main facets of the Frame Problem.

The following axiom SC1, known as the axiom of change, provides a solution in the context of the SC.

$$holds(P, do(A,S)) \leftarrow [\neg terminates(A,S,P) \wedge holds(P,S)] \vee initiates(A,S,P) \quad \textbf{[SC1]}$$

Declaratively it reads: A property P holds in the situation ($do(A,S)$) that results after the execution of action A in situation S **if** property P is not terminated by action A in situation S where it already holds **or** action A initiates property P in S.

SC1 is very similar to the combination of axioms presented by Kowalski and Sadri in [KS94]. In SC1, however, there is no predicate referring to the occurrence of an action (*happens* or *do*). The reason for this will be explained below, although it can be seen that *do* does appear as a term. Also, notice that SC1 is "an elegant" combination of the **effect axioms** (axioms describing the effects of the actions) and the **frame axioms** in traditional axiomatisations of the Situation Calculus (as argued in [RN95], pg. 206).

4.5.2 Action generation in SC

The temporal projection predicate as defined by SC1, is already sufficiently expressive to support classical planning in one-agent applications. Given a description of an initial state (everything that holds at time t_0), appropriate definitions for the predicates *initiates* and *terminates*, and a conjunction of *goal* literals (of the form $holds(p, T_i)$), one can compute sequences of actions such as $T_i = do(a_n, do(a_{n-1}, \ldots, do(a_0, t_0)), ..)$.

Thus, SC1 can be used *to generate* the set of actions required to achieve the goals. But, the same deductive procedure applied above could be used to *test* whether a "given" (obtained by some other means) literal of the form $holds(p, do(a_n, do(a_{n-1}, \ldots, do(a_o, t_o)), ..)$ is implied by SC1, its supporting predicates and the initial situation.

This double-nature of the deductive procedure is a feature of logic. It is related to the fact that in logic there is no distinction between proofs and computations. In SC, nevertheless, one can re-establish the distinction by observing that when $holds(P, S)$ is invoked with a variable S, the program will attempt to generate an instantiation of S. On the other hand, when S is a ground term at invocation time, the program will simply "test" whether P holds at that given S. And if S is somehow partially instantiated (e.g. $S = do(a_1, S')$, where S_1 is a variable), the program will try to fill the gaps.

The axiom for *holds*, however, is not the only way we have to generate action sequences. As we can describe goals with the predicate *done* (for tasks that must be done within a given time-window, e.g. $done(open, t_1, t_2)$), we can use the axiom defining this predicate to generate those sequences. We need, of course, to complete the definition of *done* but this is done with axiom [DNSCO] below.

Our aim (the reasons for which are explained below and in chapter 6) is to restrict the use of *holds* to be only a "tester" in the sense explained above. This can be achieved in SC by restricting the term T in $holds(P, T)$ to be a ground term (i.e. not a variable) whenever the predicate *holds* is invoked.

This "testing-only" restriction on *holds* in SC is very similar to the extensions made on the selection rule of some proof procedures, to cater for negative literals and avoid *floundering* [Hog90][13]. It should, therefore, be easy to incorporate the restriction into proof procedures that prevent *floundering* such a SLDNF [14]

The "testing-only" restriction is more difficult to enforce on the Abductive Event Calculus (AEC) where, even with a grounded S in the invocation, $holds(P, S)$ will generate new actions.

AEC and its limitations will be discussed in the following section. Meanwhile, as we said above, the definition of *done* must be completed. This is achieved with the axiom [DNSC0]. This axiom states a "base case" for the definition of *done* modelled along the lines of SC:

$$done(A, S, do(A, S)) \leftarrow primitive(A) \qquad \textbf{[DNSC0]}$$

The axiom reads: An action A can be executed at situation S, taking the system to situation $do(A, S)$ **if** it is a primitive (low-level, directly executable) action.

It is important to observe that the planning task is, by means of *done*, put under the control of the programmer. It will be he/she who will define how to unfold a (*done*) goal into the set of actions that must be performed to achieve it (a sequence of *do* terms in this case). For this,

[13] That rule in SLDNF says: do not select for resolution a negative literal that contains variables.

[14] Some PROLOG systems leave the prevention of *floundering* in the hands of the programmer. This could also be done here, provided that the programmer has means to check that every *holds* literal does not contain variables when it is selected for processing.

the programmer will write an OPENLOG program to describe the specific relation between the goal and the actions that can achieve it.

The programmer is, therefore, responsible for testing that the preconditions of each primitive action hold at the appropriate time. This somehow conflicts with the efforts made to provide an open programming framework that adjusts itself to a changing environment characterized by "opportunities" arising in an unpredictable fashion[15].

One could relieve the programmer of this responsibility by including a possibly redundant condition in DNSC0, as shown in DNSC0'. The condition *preconds* ensures that all preconditions of actions A hold at situation S:

$$done(A, S, do(A, S)) \quad \leftarrow primitive(A) \ \wedge \ preconds(A, S) \quad [DNSC0']$$

preconds can be redundant if the programmer has already include the testing of all the preconditions of that action as part of the programs. For instance, assuming that $clear(A)$ is the only precondition of $move(B, A)$ and that one is using [DNSCO'], the following code will test $clear(A)$ twice:

```
...
if clear(A) then move(B,A)
...
```

It will do it first after unfolding the expression in the **if** using [DN05] in table 4.2 and later while testing the preconditions of $move(B, A)$ as the axiom [DNSCO'] dictates.

Axioms SC1, DNSC0 and those in DD$\mathcal{B}$ (described above) constitute a background theory that can be used for systematic reasoning about actions and change. There is, however, a more expressive alternative language to formalize background theories: the Event Calculus, which is introduced in the following section.

4.6 Background theories in the Event Calculus

4.6.1 The temporal projection predicate in EC

The paper in which the Event Calculus (EC) was presented ([KS86]) offers, not only a set of inference rules, but also an ontology based on event and properties and the notions of initiation and termination of properties. The intuition behind EC is: A property (in the world) holds **if** an event has happened to initiate it **and**, after the event, nothing has happened to terminate it. We use the following axioms to formalize this intuition[16]:

[15] Opportunity is used here to refer to those fortunate circumstances when the environment helps the agent to achieve its goals. For the goal "go through that door and fetch a pen", the door being open and the pen being on the table immediately after, are not properties to be rigidly envisaged by the programmer and procured by the agent, but opportunities to be exploited by the system.

[16] Notice the condition $T_1 < T$ in [[EC1] and [EC2]. This condition implies that this version of the Event Calculus cannot cater for *destructive* interference between two simultaneous actions. That is, when two actions a_1 and a_2 finish at the same time (i.e. $do(a_1, \_, t_1)$ and $do(a_2, \_, t_1)$ and a_1 initiates some fluent p ($initiates(a_1, t_1, p)$) while a_2 terminates it ($terminates(a_2, t_1, p)$), then one will still be able to deduce $holds(p, t_2)$ for some t_2 such that $t1 < t_2$. a_2 will not block the post-conditions of a_1. The reasons for the weaker version of EC are discussed below, after introducing OEAC.

$$holds(P,T) \leftarrow do(A,T',T_1) \wedge initiates(A,T_1,P) \wedge T_1 < T \wedge \neg clipped(T_1,P,T) \quad \textbf{[EC1]}$$

$$clipped(T_1,P,T_2) \leftarrow do(A,T',T) \wedge terminates(A,T,P) \wedge T_1 < T \wedge T \leq T_2 \quad \textbf{[EC2]}$$

These axioms are different from most formulations of the EC (in particular [KS94]) in that the well-known predicate $happens(Event,Time)$ is replaced by the predicate $do(Action,Starting\_Time,Finishing$

The declarative reading of these axioms is precisely the one given in the paragraph above. There are, however, a couple of issues that must be addressed before one can take that as a formal declarative reading of these fundamental formulae.

4.6.1.1 The role of reification

The first of them (one that has been skipped over in previous sections) is that of *reification*. Most of the power of EC, and of SC extended with the notions of termination and initiation, rests on the fact that properties are considered *objects of the world*. Therefore, existential and universal quantification over them is permitted without having to go beyond the realm of First Order Logic. They can be represented by terms in the language[18].

4.6.1.2 On the representation of time

The second issue that needs mention is that of the representation of time. EC is neutral with respect to important aspects of the semantics of time. Terms can be used to represent *intervals* or *time points*, although in this work only time-points are used. Intervals are *implicitly represented* by their extreme time-points when it is necessary[19]. This might seem an erroneous decision given the popularity of interval-based representations ([AK90],[Sri91]). However, this project is driven by the belief that there is no significant advantage (weighted against the disadvantages, of course) in such interval-based representations. From the point of view of knowledge representation, intervals can be regarded as possibly infinite sets of points. Even if the time line is *dense*, EC is expressive enough to describe properties that hold over intervals of time. This is also valid for SC as this expressiveness is a characteristic of First Order Logic, on which both EC and SC can be formalised. An example of the type of axioms we have in mind is the following:

Example 4.6.1 To formally describe the light being *on* over an interval starting at time t_0 (inclusive) and ending "just before" t_f, one could write:

$$\forall T \ (holds(on,T) \leftarrow t_o \leq T \wedge T < t_f) \tag{4.2}$$

[17] The intention is to have the name of the agent also represented by a term in the predicate: $do(Agent, Action, Starting\_Time, Finishing\_Time)$. For the sake of simplicity, however, the term for agents is omitted here.

[18] This "representational strategy", highly successful in practice (see [All84] and [Sha97]), has "raised some eyebrows" among philosophically-oriented researchers (see [Gal91, Gal95])

[19] There is no reference to intervals as terms in the formulation. However, the axiomatization is based on time-points that initiate and terminate intervals over which certain properties hold.

There have been arguments in favour of terms representing intervals as the more appropriate choice for theory of actions [All91], [AF94]. However, as intervals can be described in terms of points, we have chosen to have points as terms and to express properties of intervals using points as we did in the example above.

EC is also neutral with respect to the structure of time. Time can be dense or discrete. Both possibilities are representable, in principle, in EC (see [Sha97] for some models with continous time in the Event Calculus).

Some researchers ([Rei96]) believe that EC requires a total linear-time ordering (given two points, the formalization should be able to say whether one is after the other or they are identical). The truth is that EC supports reasoning on **partial linear time orderings** in which the relationships between certain pairs of points are unknown.

In an agent, information about time-points ordering normally arrives gradually and can come from 1) communications: the agent is told by other agents how to order the points, 2) deduction: the agent decides the ordering, given other related time-points, and 3) experience: the agent perceives that one point is after or before the other. Thus, the formalization must be open to updates coming from any of these sources. EC seems to be better than SC in fulfilling the requirement of being open and easy to update, as we argue below.

Nevertheless, it is true that EC cannot embed a branching model of time as SC does. However, this does not imply that our agent, reasoning on a theory based on EC, will not be able to consider alternative futures (or pasts). The agent architecture (presented in Chapter 2) takes care of branching in the time line, (i. e. non-determinism due to a multiplicity of alternative courses of action) independently from EC and the background theory. Alternative pasts and futures are represented by the different extensions of the history $\mathcal{H}$ represented by the nodes in the frontier of goals of the agent. An agent like ours would be able to reason about different possibilities for "state of affairs" in its future (and in its past, see [Sha96] for an example of an agent that uses abduction to "explain" its observations), thus overcoming the limitations of a linear time model.

4.6.1.3 The first fundamental difference between EC and SC

The axiom SC1 in the Situation Calculus can be used *constructibly* to *extract*, from the answer to goals of the form *holds(P,T)*, terms representing plans. This was first done by Green [Gre69] in his first order logic axiomatization for problem solving.

In the Event Calculus, one would have to either 1) use *if and only if* versions of the axioms (that is, EC1 and EC2 would have to be strengthened) to *deduce*, from the goals *holds(P,T)*, certain sets of *do*-atoms; or 2) *abduce* those *do*-atoms from the (*if halves* of) the axioms (alone).

The first significant difference between EC and SC is that, as shown above, the *do*'s in SC are terms, while in EC one ends with a set of do-atoms as a *residue*.

This is more than a variation of syntax. It means that while in SC one can select actions (as terms) without leaving the object language, in EC this is not so straightforward.

In particular, if one is doing *abduction*, one will have to distinguish between literals that can be abduced and literals that cannot be abduced (e.g. actions can be abduced, other literals cannot).

It will be the proof procedure, not any syntactic construct of the object language, that collects the appropriate (*do*) atoms in the "residue".

Before returning to these comparisons between SC and EC, let us explain how one can collect primitive actions in EC.

4.6.2 Action generation in EC

To generate actions for planning, we employ the equivalent of axiom [DNSC0] for EC, which is introduced below. The intention is, once again, to place all the generation mechanism "underneath" *done*, the predicate that interprets programs in OPENLOG. In this way, it is the programmer who will use domain specific information to decide when to test and when to generate steps in the process of planning.

4.6.2.1 Completing the background theory in EC

It is known ([Esh88b], [Sha89], [MBD95]) that to make an *abductive theorem prover* [Ton95] behave as a planner, one has to define the set of abducibles, say Ab. In the present context one can make $Ab = \{do, <, =\}$[20]. The domain-independent background theory can then be completed with the following definition (base case of *done*)[21]:

$$done(A, T_o, T_f) \quad \leftarrow \quad primitive(A) \ \wedge T_o \leq T_f \ \wedge \ do(A, T_o, T_f) \quad \textbf{[DNEC0]}$$

By using an abductive proof-procedure (as **iff**PP [Fun96]) with these definitions, the result of successively *unfolding* a *done* goal will be a set of *do*'s (the steps of the plans to achieve the goal) plus a minimal set of "$\{<, =\}$" required to correctly order the *do*'s.

4.6.2.2 The role of abduction

Abduction provides EC with capabilities that can match the constructive mechanism of the Situation Calculus. By declaring *do* atoms as *abducibles* one has a simple criterion, built into the proof procedure, to distinguish the elements of the "residue".

The process of abduction can be understood as deduction on the "only if" halves of the definitions of the predicates involved. The **iff** proof procedure [FK96], as explained in chapter 3, is a realization of this possibility. The key operational advantage of the abductive proof procedure is that instead of simply proving or disproving an statement (i.e. given a query, arriving at confirmation or refutation of the sentence in the query), the procedure can generate hypotheses supporting the statement (i.e. given a query, the prover will produce a *residue*: a set of literals that in case of "being true" would make the query valid).

In the planning context, one can designate action-atoms as residual atoms, given a planning goal as query. Of course, that designation of atoms as *abducibles* is still attached to the particular object-level theory on which the reasoner operates. So, for instance, EC1 and EC2, regarding the predicate *do* as an abducible, constitute a version of the **Abductive Event Calculus**, other versions of which have appeared in [Eva89] and [Esh88a].

The *Abductive Event Calculus* (AEC) is as expressive as SC for the planning application in the mono-agent context. Given a goal $hold(P, T)$, with EC one will arrive at a residual set of atoms of the form $do(a_i, t_j)$[22] that will "explain" (and when executed, possibly achieve) that goal.

The greater flexibility of EC (with respect to SC) is already evident: Unlike the *do*'s in $do(a_n, do(a_{n-1}, \ldots, \ do(a_0, t_0)), ..)$, the atoms obtained by abduction are not *restricted* by a built-in temporal ordering. So, actions can be added, deleted and modified without having to rebuild the whole sequence.

[20] Recall from chapter 3 that, although = (equality) requires special treatment, it can still be regarded as an abducible.

[21] The definition of *primitive* must also be provided by the designer. It should correspond to the list of low-level, indivisible actions that the agent can perform.

[22] together with some $t_j < t_k$ (time-point ordering) atoms.

Notice again that this is not only a matter of having a more convenient syntax. Actions in SC depend on their immediate predecessors to define their situation of occurrence. In AEC (in EC in general), immediate and non-immediate predecessors are equally important to define the context in which an action is executed. Only those actions related to the occurring one (because they affect the same fluents) will be considered.

4.6.2.3 The problem of *over-generation* of abducibles

In previous sections, we explain how to restrict SC so that *holds* is used only for "testing" and not for action generation. The solution is essentially to avoid the reduction of *holds* literals containing variables.

Unfortunately, this solution does not work for AEC. In the abductive solution (explained below), reduction of a *holds* literal with AEC will inevitably "add" a new element to the "bag of abducibles" (the residue) regardless of whether there are variables involved.

Example 4.6.2 Suppose we have the OPENLOG code:

```
...
if raining then carry(umbrella)
...
```

That section of code will be mapped by the interpreter (in table 4.2) to:

$$holds(raining, T) \wedge \ done(carry(umbrella), T, T_f) \tag{4.3}$$

One wants the system to "test" whether $holds(raining, T)$ before planning to carry the umbrella. Unfortunately, any sensible background theory for this program will include information such as:

$$initiates(alter\_atmospheric\_pressure, T, raining) \tag{4.4}$$

and this, together with the fact that every *do* obtained while reducing [EC1] is abduced, implies that the system will always end up with something like:

$$do(alter\_atmospheric\_pressure, \_, T_1) \wedge T_1 < T \wedge done(carry(umbrella), T, T_f) \tag{4.5}$$

Thus, the agent will always plan to carry the umbrella, irrespective of whether the "event" $do(alter\_atmospheric\_pressure, \_, T_1)$ has been observed. Notice that this action may not even be among the agent capabilities.

We call this the problem of *over-generation* of abducibles. It is a problem because the system (the proof procedure) cannot "test" whether a property holds. Instead, it will always force in an assumption (a *do*-atom in our case) to make the property hold. There is no distinction between *what is the case* (and therefore can be tested) and *what the system assumes to be the case* (because it is required in a plan to achieve some goal).

This problem is clarified in the following sections. Meanwhile, let us say that we have adopted a controversial solution: *context-dependent abduction*, a mechanism that we present and justify (in terms of deduction and interleaving of planning and testing) in chapter 6.

4.7 The Event Calculus versus The Situation Calculus.

4.7.1 On distinguishing between "testing" and "generation"

A first advantage of the SC solution with respect to EC's is that there can be a clear and easy separation between the "testing" of properties ([SC1]) and the "generation" of actions

([DNSC0]). Both "contexts of operation" (testing and generation) can be unambiguously related to the syntax of the language (*holds* for testing, the base case of *done* for generation). So, whenever the programmer writes in OPENLOG:

```
...
look(door) ;
if open(door) then close(door) ;
...
```

this code will generate a plan in which the agent *looks at the door* and then tests whether *the door is open*, closing it if it is so. The plan will not contemplate *opening the door*, even if the agent *believes* (it is in its knowledge base) that action initiates the door being open. Observe that this means that in SC there is a simple way of "closing" the history of the problem. One must simply assume that everything that happens is explicitly represented as a term.

In contrast, to achieve that "separation" is not easy in EC. The main problem with the Abductive Event Calculus is the "over-generation of abducibles" mentioned above. Rigidly declaring a predicate (or set of predicates) as abducible *biases* the system towards the generation of new abducible atoms whenever there is no information to prevent it. In particular, axioms EC1 and EC2, intended only as testers of properties over time, now become potential "generators" of abducible atoms. EC becomes the *Abductive Event Calculus* (AEC).

So, although it can be a sound policy to reason with (e.g. the abductive **iff** proof procedure, which has been proved to be a sound reasoning mechanism [Fun96]), *unconstrained abduction in planning* might generate not only too many actions, but even actions that are unfeasible because they are not under the control of the agent[23].

Thus, the abductive strategy needs to be enriched or complemented with other control rules to limit its application. A first answer to this problem has already been built into the **iff** proof procedure. This proof procedure "disables" abduction in the body of implications and, by extension, inside negated literals in the goals (For instance, when one unfolds a positive *hold* literal with EC1, EC2 will not be used to generate a new abducible because it has been invoked through the negated literal in EC1).

The "disconnection" or inhibition of abduction in the body of implications is perfectly justified on semantic grounds (See [Fun96] and [FK96] or chapter 3). This strategy minimizes the abducible atoms in the answer to a query. However, the rule has an exception: The *equality* (=) predicate, considered an abducible in all other respects, is, in effect, "abduced" in the body of implications, by case analysis on equalities.

Nevertheless, the disconnection of abduction in the body of implications can be regarded as a sort of *context-dependent abduction*. The context being set by the body of the implication. As can be seen in chapter 3, there are special provisions in the proof procedure for separating literal "inside" and "outside" implications (in CN and in UC respectively), mainly to guarantee that they are treated differently with respect to abduction.

We explore other convenient forms of context-dependent abduction in chapter 6, as we said before.

4.7.2 On dealing with parallelism

A problem with SC is that it prevents the use of the operators **par** and + in OPENLOG "as it is".

[23] For instance, given the piece of code **if raining then carry(umbrella)**, a planner using AEC as background theory may add **increase_cloud_concentration** to the plan, simply because *it knows* that action causes rain, as we explained above

The reason for this can be seen, for instance, in clause [DN04] in table 4.2. From a goal of the form $done(c_1\ par\ c_2, s_0, T_f)$, one obtains $done(c_1, s_0, T_f) \wedge\ done(c_2, s_0, T_f)$ in which the variable T_f appears twice, one for each command. But, due to the way SC generates *do*-terms (using [DNSC0]), T_f will collect the sequence of actions corresponding to only one of the commands. The literal corresponding to the other command will lead to failure, because it will be used to "test" that command, rather than planning for it. One could solve this problem by using special terms to refer to "parallel situations". Something like $done(c_1\ par\ c_2, s_0, doplus(c_1, c_2, T_f))$ to record the fact that c_1 and c_2 should be executed simultaneously. This, however, would lead to a complete redesign of the representation to cater for correct reasoning with those special terms. In particular, one would no longer be able to say that the "time-arguments" refers to simple time-points. One may even need a complex executive, if one passes "*doplus*" for execution, because c_1 and c_2 above can be complex sequences of commands, that need to be synchronized at runtime. Hence, several non-trivial changes will be triggered by those extensions to [DN04] in table 4.2. Similar problems would arise with the operator **par**.

None of these problems affect EC because the ordering of actions is not "built in" the structure of the terms. The ordering is formalized separately by means of the predicate $<$. To that effect, the predicate $<$ is declared as an abducible (it is included in the set Ab mentioned above).

Of course, by making the ordering independent of the structure of the terms, one can describe parallel actions without having to use new and less meaningful terms. The predicate $=$ can be combined with $<$ ($\leq$) to express the fact that actions occur before, after *or at the same time* with respect to other actions.

4.7.3 On the treatment of observations

Parallelism and prevention of "over-generation", as discussed above, are very important aspects in the design of an agent. We believe, however, that there is a more crucial aspect in the design of an open and reactive agent architecture: **the treatment of observations**.

4.7.3.1 Introduction to this problem

What we call the *treatment of observations* is the task of updating the agent with information that comes from the sensors. We showed, in chapter 2, how the agent's architecture supports the insertion of new data from the environment into the agent. There was also, in the specification of GLORIA (also in chapter 2), the unusual decision of attaching inputs to the goals. We will discuss this decision later in this chapter. In this section, we concentrate on the (object-level) details of the treatment of observations and why we prefer EC, rather than SC, to support this task.

The purpose of sensing is to provide the agent with information about its environment, so that it can plan appropriate actions to achieve a given set of goals. For the purposes of reasoning, that information can be compiled into *observations*.

In principle, observations (also called inputs) can be anything that can be represented in the agent's object language. However, in the planning context and as a first approximation (see [Sha96] for a more general approach), one can restrict observations to belong to one of the following two classes:

1. *observations of properties*: what is the case at certain situations or points in time; and
2. *observations of events*: what has happened over certain periods of time (This is the case when events have duration. One can also have instantaneous events).

This restriction has the advantage that one can establish unambiguous relationships between physical sensory signals and the agent's internal representations. So, observations can be represented by atoms in a theory.

The treatment of observations is problematic because, as the environment changes, new inputs may conflict with plans derived from previous information. The capacities to withdraw previous conclusions (which support certain plans) and to revise the plans are, therefore, required. These requirements are the core of important problems in AI and computer science, such as non-monotonic reasoning, knowledge assimilation [Kow79b] and belief revision.

It should be noticed, however, that assimilating observations for planning differs from the general problem of knowledge assimilation and beliefs revision in an important respect. In case of contradiction, the agent is compelled to reject its planning assumptions instead of the observations or the beliefs in its knowledge base. Whether this constitutes an effective distinction between these problems requires further investigation.

4.7.3.2 Observations in SC

In SC, one can easily incorporate observations of properties as *holds atoms* into the existing knowledge, for instance:

Example 4.7.1 In the Blocks World, the atoms:

$$\begin{array}{ll} holds(on(a, table), s_0) & [OB01] \\ holds(on(b, c), s_0) & [OB02] \\ holds(on(a, b), do(move(a, b), s_0)) & [OB03] \end{array}$$

could be added to [SC1] as part of the definition of the temporal projection predicate *holds* provided, of course, that they are consistent with the existing definition and the rest of the theory.

[OB01] indicates that it has been observed that a was *on* the table at s_0, [OB02], that b was *on* c at s_0 and [OB02] that a was *on* b at the situation that resulted from moving a onto b at s_0. The "new" definition of *holds* would be:

$$\begin{array}{ll} holds(P, do(A, S)) \leftarrow [\neg terminates(A, S, P) \ \wedge \ holds(P, S)] & \\ \qquad \vee \ initiates(A, S, P) & \textbf{[SC1]} \end{array}$$

$$\begin{array}{ll} holds(on(a, table), s_0) & [OB01] \\ holds(on(b, c), s_0) & [OB02] \\ holds(on(a, b), do(move(a, b), s_0)) & [OB03] \end{array}$$

Notice that, if there is no inconsistency of the sort mentioned in the example, the new definition is ready to be used for further reasoning. The recursive definition of *holds* in SC1 permits the propagation of the information in the observations to other situations.

Also, when all the observations are grounded on the initial situation (s_0), the testing of consistency can be simplified[24]. This could be part of the reason why the Situation Calculus is so popular as a formalising media, as we argued in the introductory chapter.

[24] To add observations of the form $hold(P, s0)$ to a theory that only contains observations of that form, one could reduce the test of consistency to a test of this integrity constraint:

$$\neg(holds(P, s_0) \wedge \ holds(P', s_0) \ \wedge \ incompatible(P, P') \ ) \tag{4.6}$$

with an appropriate definition of $incompatible$.

$$
\begin{array}{lll}
holds(P,T) & \leftarrow do(A,T',T_1) \wedge initiates(A,T_1,P) & \\
& \quad \wedge\, T_1 < T \wedge \neg clipped(T_1,P,T) & \textbf{[EC11]} \\
holds(P,T) & \leftarrow obs(P,T_1) \wedge T_1 \leq T \wedge \neg clipped(T_1,P,T) & \textbf{[EC12]} \\
clipped(T_1,P,T_2) & \leftarrow do(A,T',T) \wedge terminates(A,T,P) & \\
& \quad \wedge\, T_1 < T \wedge T \leq T_2 & \textbf{[EC21]} \\
clipped(T_1,P,T_2) & \leftarrow obs(P',T) \wedge incompatible(P,P') & \\
& \quad \wedge\, T_1 < T \wedge T \leq T_2 & \textbf{[EC22]}
\end{array}
$$

Figure 4.1: The Observational and Abductive Event Calculus (OAEC)

However, when it comes to the second class of observations, SC causes problems. Actions in SC are represented by terms, not by atoms. So, to add new actions as observations to the theory one would have to "wrap" them up with some predicate. The predicate *holds* can do this, as shown by [OB03] in the example above, but either 1) one needs to accompany the action with the observation of some property (as $on(a,b)$ above) or 2) one has to use existentially quantified variables such as $\exists P,S\, holds(P,do(action,S))$. Alternative 2) conflicts with our project to have clauses in a logic program defining *holds*. We would need to skolemize [Hog90] such expressions, causing even more complications for the proof procedure (as commented in [Fun96]).

And if one introduces a new predicate (say, $do\_sc(Action)$) as in EC, one would be duplicating the representations of actions (actions as terms and also as predicates). This is already a load on the representation, without considering that one still needs to introduce other elements in the representation to support reasoning with the new predicate.

4.7.3.3 Observations in (the Observational and Abductive) EC

There is a sort of duality between SC and EC with respect to the representation of observations of properties and observations of actions. SC favours the representation of observations of properties. In EC, representing observations of actions is the easier problem to address.

In EC, actions can normally be represented as ground atoms of the form $do(action,t_1,t_2)$. So, without existentially quantified variables, there is no problem storing those actions in logic programs. (We show below that, even with existentially quantified variables the abductive EC can deal with observations of actions and events).

With observations of properties, EC does have problems. Observations represented by atoms of the form $holds(p,t)$ could not be used to assume the persistency of fluents like p. That is, there is no mechanism to indicate that a fluent p persists after the time-point t when it is observed. The axioms for persistency of EC work only with fluents initiated by events and actions.

Moreover, it is not clear how those observations interact with other observations and assumptions about events and actions. For instance, does the observation $holds(on(light\_bulb),t_2)$ terminate the fluent $off(light\_bulb)$ initiated by $do(switch\_off(light\_bulb),t_0,t_1)$ *an instant after* t_1?.

Fortunately, we can extend EC to solve the problem of representing observations of properties, while preserving many of the useful features we have been enumerating.

To cater for inputs of new information, we extend the Event Calculus with the new predicate $obs(P,T)$. The predicate reads: *at time T, it has been observed that P is the case*, where T is a time-point and P designates a propositional fluent[25]. To include *obs*, axioms EC1 and EC2 are substituted by EC11, EC12, EC21 and EC22 as shown in figure 4.1.

This extended version of the Event Calculus, designated as the Observational and Abductive Event Calculus (OAEC), has a set of characteristics worth listing:

1. Observations of properties and actions (or events) have similar effects on persistency. The persistency of fluents can now be clipped by observations of *incompatible* fluents. A fluent can persist since its initiation by an action, but also since it was observed.

2. Although *backward persistency of properties* is equally representable, EC11 only caters for *forward persistence*. Properties (fluents) are assumed to hold if they are observed at some point and nothing clips their persistence after that point.

3. EC22 relies on a new predicate: *incompatible* which states that two given properties cannot co-exist (e.g. $incompatible(here(me), there(me))$).

4. More important, this version of the Event Calculus is too weak to detect synergistic interference between simultaneous actions or incompatibility between properties at a certain point in time.

   So if, for instance, the agent has $obs(on,1)$ and $obs(off,1)$ as inputs and $incompatible(on,off)$ in its knowledge base, then $holds(on,3)$ and $holds(off,3)$ will be equally deducible with OAEC.

   Similarly, $do(switch\_on,0,1)$, which initiates *on* and terminates *off immediately after 1*, will not interfere with (i.e. will not block the consequences of) $do(switch\_off,0,1)$ which terminates *on* and initiates *off immediately after 1*, as well.

   One could prevent this weakness by strengthening the inequalities in the axioms of OAEC. But this would yield an axiomatization stronger than required (For instance, in the first example above, one would not be able to infer $holds(on,3)$, even if $obs(on,0)$ is also among the observations).

   Thus, OAEC **must be accompanied** by another mechanism to check consistency and non-interference, such as the integrity constraint in equation 4.6 in the footnote of page 95, which could, of course, be manipulated by the iff proof procedure.

In OEAC we have a solution that encompasses AEC and supports a natural representation of observations. The lack of agreement with intuition, mentioned before, comes from the fact that *obs* atoms (as *do*'s) are *abducibles*. Abducible atoms are not part of the knowledge base that contains the definition of the unfoldable predicates. Instead, *abducibles* are seen as permanently attached to the goals. So, the history of a problem ($\mathcal{H}$ in chapter 1), constituted by *do* and *obs* atoms, should be seen as part of the goals (in $\mathcal{G}$), rather than co-joint to the database of definitions (in $\mathcal{K}$).

This awkward movement is motivated by the semantics of the proof procedure we are employing (see chapter 3). However, it seems to coincide with the use of separated "oracles" in systems such as *transactional logic* [BK96], that offers a similar kind of functionalities. Interestingly, it also coincides with the separation of observations (inputs) from the rest of the knowledge base when it comes to minimization. This can be seen in systems and frameworks as dissimilar as [San94], [Sch94] and [Sha97].

[25] As with *do*, *obs* could also be extended to record the observing agent ($obs(Agent,P,T)$). In this thesis, it is always $obs(self,P,T)$. We omit the *Agent* argument for the sake of simplicity.

We resume this discussion in the chapter 6, where more operational elements will be available.

4.7.4 On dealing with new goals

SC is well suited for exhaustively performed reasoning on theories where all the inputs are of the form $hold(p, s_0)$ (s_0 refers to the initial situation). However, when inputs can occur at different time points or situations and, therefore, can *activate new goals at anytime*, the reasoning process has to go through a non-trivial recovery, as the following example shows:

Example 4.7.2 Consider the OPENLOG program below in figure 4.2, page 101. Suppose that the elevator is using this program and a background theory based on SC, to plan its actions and serve the floors.

Also suppose that the elevator is initially at floor 1 and the button is *on* at floor 3. This means (as will be explained below in this chapter) that the goal of serving floor 3 is activated and the controller starts planning how to achieve it. This is the sequence of activities in which the agent engages:

- The situation s_0 is designated as the initial situation and the goal below is activated.
- **(reasoning on)** $done(serve(3), s_0, T_{f1})$
  $\downarrow$
- **(reasoning on)** $done(serve(3), do(up(3), do(up(2), s_0)), T_{f1})$
- Reasoning is interrupted at this point
- **(acting/observing)**
- After acting (executing $do(up(2), s_0)$, assuming that one has a method to extract steps from the partial plan) and observing its surroundings, the agent learns that $h(on(2), do(up(2), s_0))$ (i.e.the button has been turned on at floor 2).

  Let us ignore the problem of assimilating that observation for the time being. The agent still has the problem of dealing with the new goal that should be activated (this time to serve floor 2).

  Observe that the original plan is there after execution. The reason for this is that SC has to have a grounded initial situation s_0 (for several reasons, among them the need to "close" the history and perform testing only, instead of generation, as we explained in the previous section).

  Thus, after the new goal is activated the agent has the following set of goals.

- **(reasoning on)**
  $done(serve(2), do(up(2), s_0), T_{f2}) \ \wedge done(serve(3), do(up(3), do(up(2), s_0)), T_{f1})$
  $\downarrow$
- At this point the agent has to establish the relation between the newly activated goal and the old partially executed plan. This time it is easy. The alternatives are:

  $done(serve(2), do(up(2), s_0), T_{f2}) \wedge done(serve(3), T_{f2}, T_{f1})$ and
  $done(serve(3), do(up(3), do(up(2), s_0)), T_{f1}) \wedge done(serve(2), T_{f1}, T_{f2})$.

  How the agent's proof procedure will generate these options is not hard to envisage: it will have to consider all the possible *linearizations* [Fun96] of the set of actions that are compatible with the existing plan.

This will force a complete ordering of tasks at this early stage. Storing this ordering (all the alternative plans) may be costly in terms of storage and computing time (as we show in the following section) and unnecessary if one can achieve the same behaviour with no explicit ordering. (See the discussion about planning in chapter 6).

4.7.5 On memory required to store partial plans

Storing plans (list of actions a their orderings) is more expensive (in terms of memory consumption) for the SC-based representation than for the EC-solution *when information about the ordering of actions is limited.*

To see this, assume that one has Z different types of actions that could be part of a plan. Also assume that N such actions do occurr[26]. Then the space of memory required to store those N actions and the information indicating how those actions are ordered is given by the following formulations[27]:

Proposition 2 *The memory required to store a plan with N actions and Z action-types in SC vary from a minimal*

$$SCStorage_{fullinfo} = N * aprox(log_2(Z+1)) \tag{4.8}$$

when the ordering of the actions is completely known, to a maximum of:

$$SCStorage_{noinfo} = (N * N! + N! - 1) * aprox(log_2(Z+1)) \tag{4.9}$$

when the ordering of the actions is completely unknown.

Proof. See Appendix.

Proposition 3 *The memory required to store a plan with N actions and Z action-types in EC vary from a minimal*

$$ECStorage_{noinfo} = N * (aprox(log_2(Z) + aprox(log_2(N))) \tag{4.10}$$

when the ordering of the actions is completely unknown, to a maximum of:

$$\begin{aligned} ECStorage_{fullinfo} = & N * (aprox(log_2(Z) + aprox(log_2(N))) \\ & + N * \frac{N-1}{2} * (aprox(log_2(N))) \end{aligned} \tag{4.11}$$

when the ordering of the actions is completely known.

[26] Recall that an action type corresponds to each different identifier A that we can put in $do(A,T)$. When we say that N actions occur, we mean that there must be N literals (in EC) or terms (in SC) of the form $do(A,T)$ to be stored **plus** the ordering between them (i.e. $<$ terms in EC). For instance, in EC one could have

$$do(a_1, t_o) \wedge do(a_2, t_1) \wedge t_o < t_1 \tag{4.7}$$

indicating two actions (the do predicates), two time-points (t_o, t_1), one item for the ordering relation $(t_o < t_1)$ and two actions types (a_1, a_2). See the appendix for examples in SC.

[27] The function $aprox(X)$ gives the nearest greater integer above X. e.g. if $X = 1.3$, $aprox(X) = 2$.

Proof. See Appendix.

The previous results support the following proposition which incline the balance in favour of EC as the representational strategy for partial planning (planning in which there is no explicit and complete information about the ordering of actions).

Proposition 4 *For plans with incomplete information about the ordering of the actions, memory consumption in EC is lower than in SC.*

Proof. See Appendix.

4.7.6 Summary of comparisons

1. The striking advantage of the new Event Calculus solution (AEC)[28] is that the formalism is free from over-restrictions on the ordering of actions. EC uses the predicate $<$ to express the ordering of actions suggested by the knowledge in a OPENLOG program and its background theory. But this is not the strict total ordering imposed by the SC-solution (which is built in terms of the form $do(a_1, do(a_2, \ldots, s_0, \ldots))$. In EC, one has a minimal ordering of actions, dictated by domain specific rationality about how to achieve a particular goal in that domain. There is a degree of *least commitment* in this representational strategy.

2. In practice this means that AEC supports the use of **par** and $+$ in OPENLOG without any modification to the definition of *done*. It also means that plans are **open** to alterations due to assimilation of inputs and activation of goals anytime during processing. The *world changes while the agent reasons* and the built-in chain of actions in SC does not lend itself to the *on-line assimilation* of those changes.

All these problems with SC are related to its inherent limitations to represent concurrency. In our case, this problem is exacerbated by the fact that extensions of SC to cater for concurrency ([Pin94], [GLR90]) do not seem to make any easier the task of interleaving inputting and reasoning. Having observed the tendencies among some SC supporters (e.g. [Rei96]) to move towards event-based representations, we decided to explore the aforementioned extensions of EC that, despite being counter-intuitive at first, seem to solve all the problems discussed in this section.

We end this chapter on the practical arena, by showing the platform OPENLOG/background theory is already sufficiently expressive to tackle the benchmark example: the Elevator controller.

4.8 Programming the Elevator Controller with OPENLOG

In the introductory chapter 1, we stated the intention of proving proposition 1 and in the process of doing so, show how an agent could be programmed to behave as an elevator controller. The following two subsections use the language (OPENLOG) and a background theory (based on the Event Calculus as shown above), to program the agent. We also re-state proposition 1 in more formal terms (the proof is shown in the Appendix).

```
proc serve( N )
  begin
    if currentfloor( C ) then
      if C=N then
        begin
          turnoff( N ) par open ; close
        end
      else
        if C<N then
           begin
             addone( C, Nx ); up( Nx ); serve( N )
           end
        else
           begin
             subone( C, Nx ); down( Nx ); serve( N )
           end
  end

proc control
  begin
    while on(N) do
      begin
        serve(N)
      end ;
    park
  end

proc park
  begin
    if currentfloor(C) then
      if C=0 then
        open
      else
        begin
          down(0); open
        end
  end
```

Figure 4.2: The elevator controller in OPENLOG.

$$
\begin{array}{ll}
initiates(up(N), T, currentfloor(N)) & \\
\quad \leftarrow\ preconds(up(N), T) & [INI-01] \\
initiates(down(N), T, currentfloor(N)) & \\
\quad \leftarrow\ preconds(down(N), T) & [INI-02] \\
initiates(turnon(N), T, on(N)) & [INI-03] \\
terminates(up(N), T, currentfloor(M)) & \\
\quad \leftarrow\ preconds(up(N), T) & \\
\quad \wedge\ \neg(N = M) & [TER-01] \\
terminates(down(N), T, currentfloor(M)) & \\
\quad \leftarrow\ preconds(down(N), T) & \\
\quad \wedge\ \neg(N = M) & [TER-02] \\
terminates(turnoff(N), T, on(N)) & \\
\quad \leftarrow\ preconds(turnoff(N), T) & [TER-03] \\
 & \\
preconds(up(N), T) & \\
\quad \leftarrow\ holds(currentfloor(M), T) & \\
\quad \wedge\ M < N & [PRE-01] \\
preconds(down(N), T) & \\
\quad \leftarrow\ holds(currentfloor(M), T) & \\
\quad \wedge\ M > N & [PRE-02] \\
preconds(open, T) & [PRE-03] \\
preconds(close, T) & [PRE-04] \\
preconds(turnoff(N), T) & \\
\quad \leftarrow\ holds(on(N), T) & \\
\quad \wedge\ holds(currentfloor(N), T) & [PRE-05]
\end{array}
$$

Figure 4.3: Background theory for the elevator controller

$$
\begin{array}{ll}
do(self, up(4), t_0, t_1) & [DO-01] \\
do(x, turnon(5), t_2, t_3) & [DO-02] \\
do(y, turnon(3), t_1, t_3) & [DO-03]
\end{array}
$$

Figure 4.4: The history that the elevator knows about

$$
\begin{array}{ll}
initiates(up(N), T, going\_up) & \\
\quad \leftarrow \neg terminates(up(N), T, going\_up) & [INI-04] \\
initiates(down(N), T, going\_down) & \\
\quad \leftarrow \neg terminates(down(N), T, going\_down) & [INI-05] \\
terminates(up(N), T, going\_up) & \\
\quad \leftarrow holds(currentfloor(L), T) & \\
\quad \wedge \neg(holds(on(N), T) \wedge N > L) & [TER-04] \\
terminates(down(N), T, going\_down) & \\
\quad \leftarrow holds(currentfloor(L), T) & \\
\quad \wedge \neg(holds(on(N), T) \wedge N < L) & [TER-05]
\end{array}
$$

Figure 4.5: The background theory for policy 3

4.8.1 The elevator controller for policy 1

The OPENLOG program (ELEVATOR) in figure 4.2 is equivalent to the GOLOG program presented in [LRL+95] (pg. 10)[29]. The (domain-dependent) background theory (ELE_DD$\mathcal{B}$) supporting this program is shown in figure 4.3. The elevator agent also has a record of its own history (ELE_H) since it was started [30], as it is displayed in figure 4.4.

The combination of ELEVATOR, ELE_DD$\mathcal{B}$, EC1, EC2 and DONE[31] provide the elevator-agent with the same functionalities that the GOLOG program, plus some additional ones. In the context of the history ELE_H, this agent will do the same things the agent controlled with the GOLOG program will, given the initial situation that Levesque *et al* describe in [LRL+95][32].

We can now re-state proposition 1 as follows:

Proposition 5 *Let ELE-PLAN be:*

$$
\begin{array}{c}
\{do(self, up(5), t_4, t_5) \wedge do(self, turnoff(5), t_6, t_7) \wedge do(self, open, t_6, t_8) \wedge \\
do(self, close, t_9, t_{10}) \wedge do(self, down(4), t_{11}, t_{12}) \wedge do(self, down(3), t_{12}, t_{13}) \wedge \\
do(self, open, t_{14}, t_{15}) \wedge do(self, turnoff(3), t_{14}, t_{16}) \wedge do(self, close, t_{17}, t_{18}) \wedge \\
done(self, park, t_{18}, t_{100})\}
\end{array}
$$

Let $INEQ = \{t_0 < \ldots < t_{100}\}$. *Let ELE_T be the conjunction of ELE_H, ELEVATOR, ELE_DD$\mathcal{B}$, EC1, EC2, INEQ and DONE, then:*

$$
ELE\_T \cup ELE\_PLAN \vdash_{\mathbf{iff}} done(control, t_4, t_{100}) \qquad \text{[ELEVA]}
$$

Proof: See Appendix (A.4).

4.8.2 The elevator controller for policy 3

[28] Event Calculus with Abduction of do, $=$ and $<$.

[29] Note that $addone(X, Y) \equiv assign(Y, X+1)$ and $subone(X, Y) \equiv assign(Y, X-1)$. It is assumed that there is a built-in mechanism to perform these mathematical operations.

[30] do has been extended to record the identity of the agent. This will require similar modifications in other predicates. Again, for simplicity, these modifications are ignored.

[31] the definition of $done$ plus the base case

[32] In their notation: $current\_floor(S_0) = 4$, $on(5, S_0)$ and $on(3, S_0)$.

```
proc serve( N )
  begin
    if currentfloor( C ) then
      if C=N then
        begin
          turnoff( N ) par open ; close
        end
      else
        if C<N then
          if not going_down then
            begin
              addone( C, Nx ); up( Nx ); serve( N )
            end
        else
          if not going_up then
            begin
              subone( C, Nx ); down( Nx ); serve( N )
            end
  end
```

Figure 4.6: The elevator controller with policy 3.

This policy can be straightforwardly implemented by extending ELE_DD$\mathcal{B}$ with two new fluents: **going_up** and **going_down** and using them inside the program. The new background theory (ELE_DD$\mathcal{B}$') must include the axioms in figure 4.5. And the procedure **serve(N)** should be rewritten as in figure 4.6.

4.9 The representation at work: plans that become invalid because the world changes

In this section, we use the examples above to illustrate the dynamic character of the architecture. The agent's reasoner (described in chapter 3) will process the programs and background theories presented in this chapter. This processing will yield *plans*: sequences of actions one of which the agent will try to execute to achieve the goal that may have been activated (as shown in chapters 2 and 6).

Imagine that in the pursuit of the goal[33]:

$$(\ 0 < T_1 \wedge serve(2, T_1, T_2) \ ) \tag{4.12}$$

the agent derives the sub-goal[34]

$$(0 < T_1 \wedge do(up(2), T_1, T_3) \ \wedge \ T_3 \leq T_4 \ \wedge \ do(open, T_4, T_5) \ \wedge$$

[33] For simplicity, we omit the quantifiers. The variables in these formulae are existentially quantified.

[34] Again for simplicity, we do not show all the literals and implications involved. To see how the reasoner obtains the "clipped" implication, consider the following: The condition $holds(on(2), T_4)$ is among the preconditions of $turnoff$ (entry [PRE-05], table in figure 4.3, page 102). Trying to prove this condition, the planner will produce a derivation such as:

1.- $holds(on(2), T_4)$
  ↓ unfolding with [EC12] ([EC11] omitted for simplicity)
2.- $obs(P, T) \ \wedge \ T \leq T_4 \wedge \ \neg clipped(T, on(2), T_4)$

$$T_5 \leq T_6 \ \wedge \ do(turnoff(2), T_4, T_6) \ \wedge \ T_6 \leq T_7 \wedge \ do(close, T_7, T_2) \wedge$$
$$\ldots \wedge \ (\textbf{false} \leftarrow \ clipped(0, on(2), T_4) \ ) \ \wedge \ \ldots \qquad (4.13)$$

using the information: $obs(currentfloor(1), 0)$ and $obs(on(2), 0)$.

At this point the planner suspends its processing because it runs out of resources. The plan represented by expression 4.13 above is then passed to the *executive*, which successfully executes $do(up(2), 1, 2)$. Therefore, after assimilating the feedback the agent has a plan in which $T_1 = 1 \ \wedge \ T_3 = 2$.

However, as part of that feedback the agent also perceives the input:

$$do(someone, turnoff(2), 1, 2). \qquad (4.14)$$

That is, somebody else switched off the buttom at floor 2, while the elevator was arriving at that floor.

On re-entering, the reasoner, which always checks the integrity constraints first (*demo_impl* is the first routine to be called), unfolds the implication in expression 4.13 above to[35]:

$$(\textbf{false} \leftarrow \ do(X, A, T', T) \ ) \ \wedge \ terminates(A, T, on(2)) \ \wedge \ 0 < T \ \wedge \ T \leq T_4 \ ) \qquad (4.15)$$

in which the only existentially quantified variable is T_4.

Propagation of $do(someone, turnoff(2), 1, 2)$ and $2 \leq T_4$ (this one obtained from $T_3 \leq T_4$ and the assimilated $T_3 = 2$) will cause the *promotion* of **false** invalidating this plan.

Observe in the example that the fact that the elevator executes the first action of the plan is crucial. Only after this execution the agent has the information about T_4 that it requires to falsify the plan ($2 \leq T_4$). In general, this type of information is available after some action in the plan has been executed.

Thus, provided that the reasoner (*demo*) has sufficient resources to "exhaustively" process a goal whose preconditions have vanished and provided that at least an action of the plan is attempted, the plan will be abandoned.

A more general solution could be obtained by providing the reasoner with means to handle inequalities and time constraints involving the current time (*now*), as indicated, for instance, by the time-counter of the *cycle* predicate. This would allow the agent to deduce that it is too late to follow a plan (i.e. the plan is obsolete), without having to execute one action of this plan first, as in the example above.

This solution could be combined with a mechanism for evaluating plans according to agent's preferences, using information such as estimates of the duration of each action and aggregates based on these estimates. The agent should prefer those plans whose duration do not violate the constraints on the time-points. We discuss a mechanism for encoding preferences into an agent in the chapter 5.

It is worth noticing that a related but different problem with obsolete plans occurs when a goal, that has been activated by some external condition in the environment (by using an

---

↓ by propagation of $obs(on(2), 0)$ and rewriting of $\neg$

3.- $0 \leq T_4 \ \wedge \ (\textbf{false} \leftarrow \ clipped(0, on(2), T_4))$
↓ unfolding with [EC21] ([EC22] omitted for simplicity)

4.- $0 \leq T_4 \wedge (\textbf{false} \leftarrow \ do(Agent, Act, T, T') \ \wedge \ terminates(A, T', P)$
$\wedge \ 0 < T' \ \wedge \ T' \leq T_4 \ )$

This is, of course, only one possible derivation starting from $holds(on(2), T_4)$. Observe that 1) implications are kept in case new *obs*'s arrive in future cycles; 2) The planner only reaches the last stage (4) above when it has enough resources to do so. Otherwise it will stop the derivation at some previous point. For instance, in this section we assume that *demo* suspends processing when it reaches the frontier corresponding to position 3 above.

[35] Notice that we have reintroduced into the predicate *do* the argument identifying the agent.

integrity constraint), becomes redundant because its "activating" condition has ceased to hold (perhaps because the goal has been satisfied by another agent or process). This problem is discussed in chapter 5 (section 5.2).

4.10 Discussion

OPENLOG is a logic programming language that can be used to write procedural code which can be combined with a declarative specification of a problem domain (a background theory) to guide an agent at problem-solving and planning in that domain.

To define the language, logical characterization has been given to the traditional programming structures (**if then else**, **while**, **;**, ...) in such a way that any program written with those structures can be translated into a set of logical sentences.

This mapping from procedural code to logical sentences is not only sought for the sake of clarity. The logic chosen to provide semantics for the procedural structures is also used to specify a theory of actions that models dynamic universes.

This theory of actions can be based on Kowalski and Sergot's Event Calculus [KS86], a logical formalism with an ontology based on events and properties that can be initiated and terminated by events. The Event Calculus provides a solution to the Frame Problem and also permits the efficient representation of concurrent activities and continuous domains. This has permitted the straightforward extension of the capabilities of standard PASCAL to allow for the description of parallel actions in OPENLOG programs.

Thus, the designer/programmer is offered a specification-implementation that can be used to model complex universes and also to write high-level algorithms to guide the activities of agents acting in a dynamic environment.

As in other logic programming languages, programs in OPENLOG are processed by a theorem prover that transforms goals into a set of alternative plans which a simple automata can follow.

Unlike in other approaches, however, programs in OPENLOG are intended to be interpreted[36] rather than compiled[37]. The reason for this is crucial. The process of planning (the theorem prover transforming goals into plans) is interleaved with the execution of those plans and the inputting and assimilation of observations. This guarantees *architectural reactivity*; the system as a whole will process inputs as soon as it can, increasing its chances of an opportune response (normally by an minor adjustment to its plans as will be illustrated in chapter 6). The first practical consequence of this is that the system will generate and use *partial plans* which it will refine progressively as its knowledge of the environment increases. This is a crucial difference between OPENLOG and the similar logic-based programming language GOLOG[LRL+95]. We have explored the similarities and differences between GOLOG and a previous version of OPENLOG in [DQ96].

OPENLOG also supports the other form of reactivity: *knowledge-based reactivity*. OPENLOG is a language for describing strategies for problem solving. The desired strategies are normally those that can be decomposed into one immediate primitive action and a remaining set of actions to be processed later. The planning process *moves forward* from the current state toward a final state.

This form of planning may seem atypical in the current context because theorem provers (and the planner here is a theorem prover) are normally backward-reasoning mechanisms. An

[36] As in JAVA [Mic95] and other commercial products, where code is pre-compiled to an intermediate form to be read by an interpreter/executive.

[37] As in Situated Agents [RK95] and GOLOG [LRL+95]

```
proc goto( Site )
  begin
    if at(Site) then nil
    else
      if closer(Site, S) and cleared(S) then
        begin
          step-on(S) ;
          goto(Site)
        end
      else
        if cleared(S) then
          begin
            step-on(S) ;
            goto(Site)
          end
   end
```

Figure 4.7: The pathfinder in OPENLOG.

interesting aspect of this representation is that planning is performed by searching the time line in a forward direction. This is called *progression*.

The representational strategy that supports this form of planning is not new. It is at the core of a well known device to specify grammars and program their parsers: Definite Clause Grammar or DCG [PW80]. OPENLOG programs are like DCG*s* in that they are higher level macros that can be completely and unambiguously translated into logic programs. Unlike DCG however, OPENLOG provides for negative literals.

There is another critical difference between OPENLOG and DCG. In DCGs, the "state of the computation" (which in that case contains the sentence being parsed) is carried along through arguments as is common in stream logic programming. This has the inconvenience of requiring the explicit representation of all objects in the application domain and is, therefore, cumbersome and limiting (We tested the approach in the prototypical implementation of pathfinder reactive automatas that do forward planning, reported in [DQ94]). Background theories are a flexible and powerful alternative to this approach.

The representational strategy is then very important for the purposes of reactivity. In OPENLOG the programmer is encouraged (although not restricted) to write procedures to lead the planner from the topmost goals to the low-level, directly-executable actions, in the shortest deduction time. Part of the problem of "intelligent reactivity" is, of course, that the *best* strategy always depends on the particular circumstances in which the agent is at a given point. However, the designer can always try to envisage such circumstances and to devise a general strategy to approach the problem with the least inefficiency. A "good" program for reactive *pathfinding*, for instance, is the one shown in figure 4.7[38]:

The procedure in figure 4.7 is the typical example of a piece of code organized so that the agent that uses it can be reactive (*knowledge-based reactivity*). The pathfinder is asked to find a route to its destination and to follow it. With the code in fig. 4.7, the planner does not need to build the whole plan to achieve the goal $goto(Site)$, before starting to move. On the contrary,

[38]Read $at(Site)$ as "the agent is currently at $Site$; $closer(Site, S)$ as "by stepping of S from the current location (one step away), the agent will be closer to $Site$"; $cleared(S)$ as "the space S is clear"; and $step-on(S)$ as "the agent moves one step from the current position to S. For more details see the discussion in [DQ94].

it will first decide in which direction its destination is and it may even execute a first step in that direction, before carrying on planning. The advantage is that the agent is ready to start moving sooner (reacts earlier) and, if it does move, it risks less error later in its plans as it will decide each step on *fresher* and more precise information.

An agent architecture like GLORIA can also be enriched by other functionalities of a logical framework. An agent can be set up to be highly autonomous by providing it with appropriate integrity constraints for activation of goals. A first exercise in that direction was the language defined by [ICGRAMM] in chapter 2. That notation, however, is slightly demanding for a programmer and the use of quantifiers can be confusing if one has to write long pieces of code.

In the following chapter, we re-introduce the ideas in [ICGRAMM] as part of a better defined programming language that we call ACTILOG. We also show how to put heuristic information into the agent to improve its ability to react timely and sensibly.

Chapter 5

Agent Reactivity and Preferences

5.1 An alternative to OPENLOG: The ACTILOG language

ACTILOG is a language to write **condition** $\rightarrow$ **action** activation rules. In the introductory chapter 1 we suggested that the process of *activation of goals* could be understood as the derivation of unconditional goals from integrity contraints. We saw in chapter 2 how implications (conditional goals) could be used to state integrity contraints for an agent. These integrity contraints described conditions under which the agent's goals must be reduced to plans that can be executed. For instance, a rule such as **if** A **then** B, will indicate to the agent that whenever it can prove that A is the case, it then should pursue goal B. B is normally the description of a task that must be reduced to a set of low-level, primitive actions that the agent can execute.

ACTILOG is similar to other *production rule* languages (as OPS5 [Bro85]). It can be used to write **condition-action rules** to characterize the behaviour of a reactive agent.

A first difference with respect to previous work is that ACTILOG, as OPENLOG, relies on a general purpose representation of actions and events (i.e. a logic of actions) in the form of background theories. Temporal and common-sense reasoning about initiation and termination of properties is, as we have seen, possible within this framework.

A second important difference (with respect to OPS5, in particular) is that ACTILOG is an *object-level* language[1]. It does not include syntactic constructs like **goal G** or **plan P**. These characterizations are provided by the architecture of the agent (in chapter 3).

ACTILOG is intended as a language to write declarative sentences stating the relations between observations and subsequent actions to be performed in response to those observations. These sentences are regarded as *integrity contraints* for the behaviour of the agent, in close analogy to integrity constraints for information stored in a database.

All the control devices required to interpret and verify integrity contraints are provided by the proof procedure that characterizes the reasoning mechanism of the agent. The intention is to separate concerns about how to describe conditions for activation from concerns about the efficiency of the platform.

However, the enriched syntax of ACTILOG (with respect to simple implications with atomic heads) supports the arrangement of the activating conditions so as to minimize redundant pro-

[1] As it is the case with OPENLOG.

cessing. The head of an implication can be almost any logical sentence (including implications) and thus it is possible to write, not only sentences of the form: $(A \leftarrow B) \leftarrow C$, but also sentences such as $((A \leftarrow B) \wedge (C \leftarrow D)) \leftarrow E)$, where E is a condition shared by both nested implications. This captures some of the functionalities of the RETE algorithm which has been used to improve the efficiency of the OPS5 platform (.ibid).

The following two sections describe ACTILOG in detail. Afterwards, We compare the language with OPENLOG and discuss the advantages of each.

5.1.1 Syntax of ACTILOG

The syntax of ACTILOG is presented in table 5.1 in a variant of BNF form. The conventions to read the table are the same as before[2]. As in OPENLOG, ACTILOG's syntax is open so that the programmer can include fluent and action names into the language. Actually, all the lower level syntactic categories, including boolean fluents, are borrowed from table 4.1.

The top-most syntactic category is $UNIT$. A "unit" in ACTILOG gathers a set of activation rules (defined by ACT_Rule) related to a particular task. Below (in fig. 5.1), we give an example of ACTILOG encoding by translating the sentences in [ICSERVE] in chapter 2 for the elevator controller.

Another important category in the syntax is $Quants$. It stands for the sub-expression in an ACT_rule that specifies which variables are existentially and universally quantified.

Variables for which quantification is not indicated are assumed as universally quantified and their scope of quantification is the whole activation unit. This means that the scope of the variable so *implicitly* quantified will include the scope of quantification of the other variables. This aspect must be emphasized because it implies that existentially quantified variables will *depend on* those implicitly quantified variables for *skolemization*, as shown in example 5.1.1:.

Example 5.1.1 Consider the ACTILOG rule:

```
exists T1 if on(N) at T and T lt T1 then serve(N) at T1
```

It should be read as: $\forall N\ \forall T\ \exists T1(serve(N, T1) \leftarrow on(N, T) \wedge T < T1)$.

So, in clausal form one would write: $serve(N, f(N, T)) \leftarrow on(N, T) \wedge T < f(N, T)$, where $f(N, T)$ is a skolem function.

Thus, the syntax of ACTILOG takes it beyond the realm of Horn clauses extended with negation. One can now have existentially quantified variables in the *head* of the clause. The implications of this are discussed in the following section.

The other syntactic categories are better understood by the translation of the rules into integrity constraints involving the predicates $holds(P, T)$ and $done(A, T_o, T_f)$. This is the purpose of tables 5.2, 5.3 and 5.4 in the following section.

Also, notice that ACTILOG allows for *composite task names*, using the operators "**;**" and "**par**" (and we could also add "**+**"). The idea is to borrow part of the definition of *done* in table 4.2 to deal with these. However, for the sake of simplicity we omit these operators in the semantics of ACTILOG. Observe, however, that one still needs the "base-case" of the definition of *done* ($[DNEC0]$ or $[DNSC0]$ in chapter 4.

[2] in table 4.1 in the previous chapter. Notably, C* represents zero or more occurrence of the category symbol C

Table 5.1 ACTILOG Language: Syntax			
Unit	::=	*Set* **to** *TaskName*	*Activation Unit*
Set	::=	*Act_Rule* (**and** *Set*)*	*Activation Set*
Act_Rule	::=	*Quants* **if** *Body* **then** *Head*	*Basic Activation Rule*
Quants	::=	*One_Quant**	*Quantifiers*
One_Quant	::=	$\exists$ *Var*	*One Quantifier*
	|	$\forall$ *Var*	
Body	::=	*Condition* (**and** *Body*)*	*Body of an IC*
Head	::=	*Disjunct* (**or** *Head*)*	*Head of an IC*
Disjunct	::=	*Set*	
	|	*Task*	
	|	**false**	
Condition	::=	$Func_{fluent}$ **at** *Term*	*Conditions*
	|	*Task*	
	|	**not** *Condition*	
	|	*Query*	*Tests on "rigid" information*
Task	::=	*TaskName Schedule*	*Task descriptions*
Schedule	::=	*Schedule* **and** *Schedule*	*Schedules*
	|	**at** *Term* | **before** *Term*	
	|	**after** *Term* | **starting at** *Term*	
	|	**finishing at** *Term*	
	|	**starting before** *Term*	
	|	**finishing before** *Term*	
	|	**starting after** *Term*	
	|	**finishing after** *Term*	
TaskName	::=	$Func_{action}$ | $Func_{proc}$	*Action names*
	|	*TaskName* (; *TaskName*)*	
	|	*TaskName* (**par** *TaskName*)*	
$Func_{action}$	::=	...	*As in OPENLOG*
$Func_{proc}$	::=	...	*As in OPENLOG*
$Func_{fluent}$	::=	...	*As in OPENLOG*
$Func_{boolean}$	::=	...	*As in OPENLOG*
Term	::=	*Ind* | *Var*	*As in OPENLOG*
Ind	::=	...	*As in OPENLOG*
Var	::=	...	*As in OPENLOG*

Table 5.1: Syntax of ACTILOG

5.1.2 Semantics of ACTILOG

It must be evident at this stage that, first OPENLOG and now ACTILOG, are no more than "syntactic sugar" for logic and traditional logic programming in the case of OPENLOG. The exercise of defining these languages is important, however, because it helps to clarify what logical concepts are involved in programming an agent.

Thus, as with OPENLOG, to understand the meaning of any ACTILOG unit, one must restore it to its underlying logical form. Unlike OPENLOG programs however, an ACTILOG unit cannot be transformed into a normal logic program, without losing expressiveness. This is due to the fact that existential quantification is highly restricted in logic programs. We must use a richer form of logic that admits explicit quantification of variables and a more complex sentence structure.

Nevertheless, this is not a problem in our system because, as discussed in chapter 3, **iff**PP can accommodate a more general structure for implications (conditional goals). A few new functionalities must be added to the specification of **iff**PP:

1. The proof procedure must admit nested implications with the syntax described by grammar [ICGRAMM] in chapter 2 (repeated here for easy reference):

$$
\begin{array}{l}
Conditional \ ::= \ Head \leftarrow Body \\
Head \ ::= \ Disjunct^{\vee} \\
Disjunct \ ::= \ Conditional^{\wedge} \\
\quad \mid (Conjunct^{\wedge}) \wedge (Conditional^{\wedge}) \\
\quad \mid Conjunct^{\wedge} \\
Body \ ::= \ Conjunct^{\wedge} \\
Conjunct \ ::= \ Literal \mid \textbf{false} \mid \textbf{true} \\
Literal \ ::= \ Atom \mid \neg Atom \qquad\qquad [\textbf{ICGRAMM}]
\end{array}
$$

2. The proof procedure must be able to deal with existential quantification. The **iff** proof procedure (**iff**PP) can do this. We state the requirement in this chapter because it is essential for the notion of activation of goals, as the examples below show.

It is worth noticing that the inference rules remain the same except for **splitting of implications** and **case analysis**, which must now include a new set of conditions for their application.

Recall that the *splitting of implication* in **iff**PP is not applied if there are universally quantified variables in the head of the implication. The reason for this, which also applies to the rule of *case analysis* is related to skolemization and is better explained by example 5.1.2, a follow-up to example 5.1.1:

Example 5.1.2 Suppose that we split:

$$\forall N \ \forall T \ \exists T1((serve(N, T1) \wedge T < T1) \leftarrow on(N, T)) \tag{5.1}$$

We will end up with:

$$\forall N \ \forall T \ \exists T1((serve(N, T1) \wedge T < T1) \vee (\textbf{false} \leftarrow on(N, T))) \tag{5.2}$$

The reason not to split the sentence in this example is that the first disjunct in the resulting sentence ($serve(N, T1)$) cannot be incorporated into the unconditional goals (as it should be), because it involves the universally quantified variable N. If one insists on doing so, the proof procedure will treat N as existentially quantified.

```
if currentfloor(M) at T and on(N) at T then (
        if M eq N then open par turnoff(N); close after T
    and if M lt N then addone(M,Nx); up(Nx) after T
    and if N lt M then subone(M,Nx); down(Nx) after T )
```

Figure 5.1: ACTILOG Rules for the elevator controller

However, it remains the problem that the system is losing the *dependency between existential and universal quantification*. One can see this by looking back at the clausal form of the sentence in the example 5.1.2: $serve(N, f(N,T)) \leftarrow on(N,T) \wedge T < f(N,T)$, which after splitting leaves $serve(N, f(N,T))$ as a separate disjunct. The value of the second argument of *serve* is *determined*, not only by N but also by T.

Of course, nothing has been lost if one keeps the dependency by appealing to the skolem function ($f(N,T)$). However, this would imply significant modifications to the proof procedure. Use of skolemization has been attempted before (see Denecker and De Schreye's SLDNFA [DDS92] for a system similar to **iff**PP, but that uses skolemization) and it has proved to be cumbersome and inefficient.

However, one can reach a proper compromise with the following strategy: The proof procedure will preserve the dependencies between variables in the implications and will be **banned** from splitting (or doing case analysis) on any implication the head of which contains variables with *active dependencies*.

The concept of **active dependency** is simple. The dependency between $T1$ and N and T above is active if N and T, in that implication, have not been assigned known constant values. For instance, when, by propagation of $on(3,1)$, the implication above becomes $\exists T1(serve(3,T1) \leftarrow 1 < T1)$ then this can safely be handled by splitting because $T1$ is now as defined as it can be by skolemization ($T1 = f(3,1)$).

Observe that, for this strategy, the only extension required in **iff**PP is a list of "dependencies" between variables in the implications. A list which could be built by straightforward parsing of the quantifiers in the original integrity constraints. To make the process easier, we restrict the quantifiers in the ACTILOG rules to appear as shown in table 5.1.

All this explained, we can now show how to transform ACTILOG units into sets of integrity contraints for agent programming. The procedure is described by a normal (meta-)logic program in tables 5.2, 5.3 and 5.4. To simplify the presentation the syntax of the logic programs is slightly relaxed. "{}" represents both empty categories in ACTILOG and empty formulae. The predicate *append*/3 has the usual interpretation.

5.1.3 OPENLOG versus ACTILOG

If we do not allow for activation of goals (i.e. we do not want to use ACTILOG, only OPENLOG), the agent would have to have one, top-most main goal from which all the possible activities of the agent are derived. This is the solution with GOLOG (See [LRL+95]) that we tried to emulate in chapter 4 for comparison.

But ACTILOG rules can make the agent more *open* to its environment, (as we show below). Thus, the programmer will normally have to use ACTILOG and OPENLOG to program the agent. Furthermore, one could have ACTILOG extended as suggested in table 5.1 to include the operators ";", "*par*" and "+". This would make it as expressive as OPENLOG for the representation of sequential and parallel actions.

Table 5.2 ACTILOG translation into Integrity Constraints

$rewrite\_activa\_ic(Set$ **to** $TaskName, IC)$
$\quad \leftarrow transform(Set, IC)$ **[RW – ACTI]**

$transform(QVars\ FRule$ **and** $RestRules,$
$\quad NQVars(NewFRule \wedge NRestRules))$
$\quad \leftarrow transform(\{\}\ FRule, QVarsFR\ (NewFRule))$
$\quad \wedge transform(\{\}\ RestRules, QVarsRest\ (NRestRules))$
$\quad \wedge transform\_quantifiers(QVars, QVars')$
$\quad append(QVarsFR, QVarsRest, QVarTemp)$
$\quad append(QVars', QVarTemp, NQVars)$ **[TRSET]**

$transform(QVars$ **if** $Body$ **then** $Head,$
$\quad NQVars(NewHead \leftarrow NewBody))$
$\quad \leftarrow transform(\{\}\ Body, \{\}\ NewBody)$
$\quad \wedge transform(QVars\ Head, NQVars\ (NewHead))$ **[TRRULE]**

$transform(\{\}\ Condition$ **and** $RestConds,$
$\quad \{\}(holds(P, T) \wedge NRestCond))$
$\quad \leftarrow Condition = P$ **at** T
$\quad \wedge is\_fluent(P)$
$\quad transform(\{\}\ RestCond, \{\}\ (NRestCond))$ **[TRCOND – FL]**

Table 5.2: Translating ACTILOG rules into Integrity Constraints (Part 1)

Table 5.3 ACTILOG translation into Integrity Constraints

$transform(\{\}\ Condition$ **and** $RestConds,$
$\quad \{\}(done(Name, T_1, T_2) \wedge LogSched \wedge NewRC))$
$\quad \leftarrow\ Condition = Name\ Schedule$
$\quad \wedge\ actionname(Name)$
$\quad \wedge\ transform\_schedule(T_1, T_2, Schedule, LogSched)$
$\quad \wedge\ transform(\{\}\ RestConds, \{\}\ (NewRC))$ **[TRCOND – ACT]**

$transform(\{\}$ **not** $Condition, \{\}\neg(NewCond))$
$\quad \leftarrow\ transform(\{\}\ Condition, \{\}\ (NewCond))$ **[TRCOND – NOT]**

$transform(QVars\ Disjunct$ **or** $RestDisj,$
$\quad NQVars(NewDisj \vee NewRD))$
$\quad \leftarrow\ transform(\{\}\ Disjunct, Vars1\ (NewDis))$
$\quad \wedge\ transform(\{\}\ RestDisj, ReVars\ (NewRD))$
$\quad \wedge\ transform\_quantifiers(QVars, LogQVars)$
$\quad append(Vars1, ReVars, QVarTemp)$
$\quad append(LogQVars, QVarTemp, NQVars)$ **[TRHEAD – OR]**

$transform(QVars\ Task,$
$\quad NQVars(LogSched \wedge done(TaskName, T_1, T_2)))$
$\quad \leftarrow\ Task = TaskName\ Schedule$
$\quad \wedge\ transform\_schedule(T_1, T_2, Schedule, LogSched)$
$\quad \wedge\ transform\_quantifiers(QVars, QVars')$
$\quad append(QVars', \{\exists T_1\ \exists T_2\}, NQVars)$ **[TRHEAD – ACT]**

Table 5.3: Translating ACTILOG rules into Integrity Constraints (Part 2)

Table 5.4 ACTILOG translation into Integrity Constraints	
$transform\_quantifiers(\{\},\{\})$	**[TRQU1]**
$transform\_quantifiers(\textbf{exists}\ V\ RestQV, \exists V\ RestQV')$ $\leftarrow\ var(V) \wedge\ transform\_quantifiers(RestQV, RestQV')$	**[TRQU2]**
$transform\_schedule(T_o, T_f, \textbf{at}\ T, T\ \textbf{le}\ T_o \wedge T\ \textbf{lt}\ T_f)$	**[TRSCH1]**
$transform\_schedule(T_o, T_f, \textbf{before}\ T, T_o\ \textbf{lt}\ T \wedge T_f\ \textbf{le}\ T)$	**[TRSCH2]**
$transform\_schedule(T_o, T_f, \textbf{after}\ T, T\ \textbf{le}\ T_o \wedge T\ \textbf{lt}\ T_f)$	**[TRSCH3]**
$transform\_schedule(T_o, T_f, \textbf{starting at}\ T, T_o\ \textbf{eq}\ T \wedge T_f\ \textbf{lt}\ T)$	**[TRSCH4]**
$transform\_schedule(T_o, T_f, \textbf{finishing at}\ T, T_o\ \textbf{lt}\ T \wedge T_f\ \textbf{eq}\ T)$	**[TRSCH5]**
$transform\_schedule(T_o, T_f, \textbf{starting before}\ T, T_o\ \textbf{lt}\ T)$	**[TRSCH6]**
$transform\_schedule(T_o, T_f, \textbf{finishing before}\ T, T_f\ \textbf{lt}\ T)$	**[TRSCH7]**
$transform\_schedule(T_o, T_f, \textbf{starting after}\ T, T\ \textbf{lt}\ T_o)$	**[TRSCH8]**
$transform\_schedule(T_o, T_f, \textbf{finishing after}\ T, T\ \textbf{le}\ T_f)$	**[TRSCH9]**

Table 5.4: Translating ACTILOG rules into Integrity Constraints (Part 3)

As one could expect after seeing the example in chapter 2 ([ICSERVE]), the ACTILOG rule in figure 5.1 provides a solution for the elevator controller as complete as those shown in chapter 4.

Observe that an ACTILOG "unit" will have neither recursive call, nor **while** statements. The iterative behaviour is generated by the architecture of the agent, i.e. by the *cycl*ing in which the whole system is engaged (as explained in chapter 2).

An ACTILOG unit is more *open* to the environment than a OPENLOG procedure because *cycle* will check the environment on each iteration and new information will be constantly arriving. There is less interaction with the environment when one has a **while** statement in a OPENLOG procedure which is being unfolded during a call to *demo*. By using **while**, one is introducing an iterative process in addition to (and without the benefits of interaction with the environment of) the iterative process generated by *cycle*. It is like having a loop within a loop, with the inconvenience that the "included-loop" (*demo* processing the **while** statement) is not forced to check the environment on every iteration, as *cycle* is.

Notice that, this is the case even if **while** statements can be interrupted to assimilate inputs. To achieve the same number of "tests" on the environment (calls to *observe* or *try* in chapter 3) per unit of time, one would have to to restrict a *demo* program processing a **while** to suspend processing after each iteration. This requires a careful tuning of the resource argument R of *demo* or a modification of the structure of this predicate, to make it specific to the requirements of the **while** construct.

In ACTILOG, *cycle* defines the only iterative mechanism. No "loops within loops" can affect the interaction with the environment.

In addition, ACTILOG units can support "planning ahead". Actions will be promoted from the head of implications to the bag of abducibles (the residue Δ in chapter 3) and after that they will be "firing" implications and triggering subsequent actions.

There still is one more advantage in ACTILOG due to the fact the we are using the **iff** abductive proof procedure. Plans generated from ACTILOG rules, in contrast to those obtained

from OPENLOG procedures, *can be made to contain a minimal set of abduced steps*. The checking of preconditions can be done in the body of the implications, where abduction is not allowed by the proof procedure. This form of precondition testing blurs the distinction between triggering conditions and proper preconditions of actions. However, by using ACTILOG only, we will not have to inhibit the abductive process, to cater for "over-generation of abducibles", the problem explained above (also discussed in chapter 6).

Thus, OPENLOG and ACTILOG, in the context of abductive logic programs, *could be* alternative solutions for the same problem (i.e. both could be used to generate the same behaviour in the agent) if OPENLOG is accompanied by a mechanism to inhibit abduction. We return to this discussion in chapter 6.

All these advantages suggest that ACTILOG is a more general programming framework than OPENLOG. There are however, points in favour of using OPENLOG as the programming language (or even better, a combination of OPENLOG and ACTILOG, as we suggested above. We followed this approach in the prototype discussed in chapter 6).

The first advantage comes from Software Engineering. For complex tasks and domains, the set of integrity contraints can be very large and difficult to arrange as one "unit". In those circumstances, a more "modular" approach, for instance with procedures in OPENLOG, could be more advisable.

The second advantage in the OPENLOG solution is related to the first but is more subtle. In OPENLOG procedures, the ultimate goal being pursued can always be inferred from the code of the procedures. For instance, in the elevator example, once $on(3,1)$ triggers the goal $1 < T_1 \wedge serve(3,1,T_1)$, $serve(3,T_2,T_1) \wedge 1 < T_2$ can always be inferred from the literals in the frontier. These literals are part of the agent's goals while the agent is trying to achieve "serving the third floor by T_1".

Having information about which higher goal the agent is aiming to (and how much is still to be done to achieve it) in a partial plan is then easier in OPENLOG.

This kind of information can be particularly useful when the system is using heuristics to guide its search process and when it is trying to decide on the importance or urgency of its goals.

But even this can be done, to some extend, in ACTILOG, although by appealing to an extra-logical resource. In the first category in table 5.1, a $Unit$ is characterized by a Set and a $TaskName$ ($Unit ::= Set$ **to** $TaskName$), where $TaskName$ indicates the ultimate goal at which the integrity contraints in Set are aiming.

This is an extra-logical device because $TaskName$ is lost in the translation of ACTILOG rules into integrity contraints that define their semantics. However, if one maintains this "label" attached to the ACTILOG unit, one could identify the tasks that have been triggered and reason about their state of planning and execution.

Of course, this is not the only way of knowing about pending tasks. One could also use "state encoding", as described by Allen [All91]: within the language, one would introduce the fluent $serving(N,T)$, initiated by the observation $on(N,T)$, and this would be enough for the agent to know which the on-going tasks are.

One last remark about ACTILOG and the activation of goals. Observe that, from the perspective of a reactive agent, *there may be no need* to remember which higher goal the agent is planning and acting for. For instance, in the case of the elevator controller, the agent does not need to remember $serve(3,T)$, activated by $on(3,T')$ for some $T' < T$.

If the signal stays on "outside in the environment", the agent will be able to realize that the task is still pending if it fails to reach its higher goal ($serve(3,T)$ in this case) with the first (re-)actions. It is as if the agent is using the "world as its own model" [Bro91a] and so, representations (memory) of inputs and goals (such as records of the signals and the triggered

tasks) will not be necessary. In a "cooperating" environment like that, an agent needs fewer deliberative resources in order to be efficient and effective. We have exploited this possibility in the implementation discussed chapter 6. The following section discussed the logic of activation of goals with one example to illustrate how the reactive nature of integrity constraints can be combined with planning.

5.2 Activation of goals for planning

The purpose of activating a goal is to have the agent plan actions to achieve it. As we discussed above, sometimes the environment is such that the agent does not need to plan. In those cases, reactivity becomes more important in producing sensible behaviour, and then simple integrity constraint or ACTILOG rules are sufficient to generate that behaviour.

However, the "reactive" use of integrity constraints to activate goals could be a source of inadequate or improper behaviour. This could be the case, for instance, if the agent continues executing a plan that it has devised to achieve an "activated" goal, even though the "activating" conditions have ceased to hold.

To illustrate this, let us use the context of the example discussed in section 4.9, in chapter 4. Imagine that the goal:

$$\exists T_1\ \exists T2\ (0 < T_1\ \wedge\ serve(2, T_1, T_2)\ ) \tag{5.3}$$

has been activated from the implication:

$$\exists T_1\ \exists T2\ (T < T_1\ \wedge\ serve(N, T_1, T_2) \leftarrow\ obs(on(N), T)\ ) \tag{5.4}$$

by the input: $obs(on(2), 0)$

Also imagine that, as in chapter 4, half-way through the execution of the corresponding plan, the signal at floor 2 is turned off. The agent observes this, because it has interrupted its reasoning to try the first action of the plan, and the information about the new status of the signal arrives as "feedback".

It would be incorrect[3] for the elevator to keep executing this plan as its motivating condition (that the signal was "on" and the floor ought to be served) has vanished.

The problem is that the elevator (executing an OPENLOG "serve" procedure as in chapter 4 and with the integrity constraint 5.4 above) has no means of deducing that the plan is now unnecessary and must be abandoned, until it actually tries the $turnoff$ action (which will fail because the signal is not "on").

We can solve this problem in several ways with our agent architecture. We discuss one general[4] and one specific solution below.

A general solution is to modify the axiom [DNEC0] to include an explicit test of all the preconditions of all the primitive actions, like this:

$$\begin{array}{ll} done(A, T_o, T_f) & \leftarrow\ primitive(A)\ \wedge\ preconds(A, T_o) \\ & \wedge T_o \leq T_f\ \wedge\ do(A, T_o, T_f) \qquad\qquad [\mathbf{DNEC0'}] \end{array}$$

The axiom [DNECO'] would allow for the "clipped" constraint (discussed in chapter 4, section 4.9) to be produced and used by the planner to falsify the plan. If the agent completes that plan up to the point where the preconditions of $turnoff$ are reasoned about, it will "realize" (before trying to execute it) that the action $turnoff(2)$ is going to fail (precisely because the

[3] with respect to an idealised model of perfect rationality with no resource constraints for reasoning.

[4] General solution for those cases when the "motivating" condition (e.g. $on(2)$ above) is also the precondition of some action in the plan (as in the case of $turnoff(2)$ above).

elevator assumes that the signal will not be "on" at that floor). This is the reason to drop the plan.

Notice that as in section 4.9, we are assuming here that either some action of the plan has been executed or the planner has access to some mechanism to handle inequalities and time-constraints involving the current time. As we said in that section, this inequality-handling mechanism could be combined with a mechanism to evaluate preferences, which is the subject of the following sections. One could also maintain an explicit record of the goal that has been activated and its activating condition as "contextual" information. This type of information and the use of labels attached to the plans is discussed below in chapter 6, section 6.3.4.

That "general" solution to the problem of activating conditions that ceased to hold (leaving "triggered" plans *without justification for their execution*) could be expected to be inefficient. This is because the planner needs to "complete" the plan up to the point where the constraints on the preconditions of the actions are made explicit (e.g. the constraint $\mathbf{false} \leftarrow clipped(0, on(2), T_4)$ must be derived by the planner, before it can be used to test whether the precondition persists).

One could improve the efficiency of the planner by providing a more precise and informative integrity constraint to activate the "serve" goal. This would be a specific solution because it uses knowledge specific to the problem. For instance, after introducing a new abducible predicate[5] *serving*, the constraints:

$$\begin{aligned} &\exists T_1\ \exists T_2\ (\ (T < T_1\ \wedge\ serving(N, T, T_2)\ \wedge\ serve(N, T_1, T_2)\ ) \\ &\qquad\qquad \leftarrow\ obs(on(N), T)\ ) \\ &\wedge\ (\mathbf{false} \leftarrow\ (serving(N, T_3, T_4)\ \wedge\ do(S, turnoff(N), T_0, T_f) \\ &\qquad\qquad \wedge\ S \neq self\ \wedge\ T_3 \leq T_0\ \wedge\ T_f \leq T_4\ )\ ) \end{aligned}$$

will have any plan to achieve the goal $serve(N, T_1, T_2)$ falsified, if an event that switches the signal off (presumably other agent doing it) is observed before the plan is executed by *this* agent[6]

Thus, integrity constraints do support some basic, rational behaviour in a multi-agent, dynamic environment. Whether they can be extended to cater for more complex cases of coordination and cooperative behaviour requires further investigation.

5.3 How to incorporate preferences into an agent

5.3.1 From control strategies to time management

This chapter and the previous are about languages to encode domain specific knowledge into an agent. In the chapter 4 and the first section of this chapter the attention concentrates on languages to describe domains of expertise and strategies for problem solving in those domains. These knowledge-description tools and the rules of inferences of the proof procedure (as described in chapter 3) are essential components of an agent with reasoning capabilities. However, they may not be sufficient to yield efficient and effective behaviour.

[5] This means that the set Ab, introduced in chapter 3, will contain $\{do, =, <, obs, serving\}$. The introduction of *serving* could be regarded as an instance of "state-encoding" as discussed by Allen in [All91] and also mentioned in the previous section.

[6] Here we also assume that there is a mechanism to deduce that the turning off of the signal does occur after the instant when the goal is activated ($T_3 \leq T_0$) and before the plan is completed ($T_f \leq T_4$). This is the aforementioned inequality-handling mechanism. Proper inequality processing is already provided for cases when some action in the plan has been executed, as shown above. The algorithms for treatment of inequality in the current architecture are discussed in chapter 6 (see figure 6.2).

It is known that the effectiveness and efficiency of an inference rule is highly dependent on the strategy to apply it. In logic programming, the selection of clauses and literals for resolution is guided by a set of predefined **control rules** that determine the way in which the space of possible derivations is searched. Two types of rules are normally used: A *search rule* determines the clause that will be employed to resolve a literal and a *computation rule* determines the literal, within a conjunction, that must be selected for resolution. A *text-order search rule* and *left-to-right* computation rule determine the depth-first search engine of standard PROLOG interpreters [Hog90]. If one wants to alter this uninformed, brute-force search strategy, one can substitute those control rules by a *control strategy* that incorporates *heuristics* and domain-specific information for a more effective search, as in the A^* algorithm and its variants ([HNR68], [DP87]).

Search and heuristic search are well-known territory in Artificial Intelligence. There is a large set of well founded solutions available. However, what is wanted here is a control mechanism that works for any problem, but that can be made more effective and efficient for certain contexts and domains. What we want is to provide knowledge to the agent, so that it can reach a prompt but sound decision when it has to react in stressing circumstances. What we need is not only a flexible search engine and a set of heuristics for each application domain, but also flexible languages to program this heuristic knowledge into the agent.

So, it is perhaps more promising to approach the problem of agent control strategies from the perspective of decision makers that can manage their tasks and time in a effective and efficient manner in very dynamic contexts.

We come to this problem with an important resource. The combination of object and meta-language allows access to domain-specific, context dependent information and, within the same language, to the inference rules that characterize the reasoning mechanism of the agent. Thus, the language allows for description of a more complex reasoning mechanism where the control on the inference rule is tailored to the requirements the application (e.g. planning).

In books on time management it is common to see references to *importance* and *urgency* as criteria for organizing plans and schedules. If a task is urgent, it must be planned before any other and executed as soon as possible. If a task is important, it must be ensured that it will be executed. By rating tasks by both attributes, an agent will be able to decide what to do next. By considering all the tasks that are important, and "ordering" them by urgency, the agent has a general strategy for time management and optimal planning.

The notions of *importance* and *urgency* of tasks can be accommodated within the agent architecture presented in this thesis, in the following way:

- Any important task must have some "triggering conditions" (e.g. if a car is moving toward you while you cross the road, you know it is important to get out of its way). A set of integrity constraints, written as ACTILOG rules, can be used to capture the relationships between those triggering conditions and the important tasks. For instance, for the elevator it is important to *serve* any floor in which the calling button has been pressed on.

  Thus, writing integrity constraints (or ACTILOG rules) is the first step to program the "important" goals into an agent. The second step can be done in two different ways.

  The programmer could manually reduce a task to a set of primitive actions that achieve it, and then accommodate these actions into integrity contraints. In this case the pure ACTILOG language (describe in table 5.1 would be sufficient as representational medium.

  Alternatively, the programmer can define OPENLOG procedures to be used with the predicate *done* representing tasks that have been activated. In this case, the activated goals (*done* atoms) which will be reduced to primitive actions by unfolding.

- As ACTILOG rules can trigger actions leading to different goals, the agent must have some way of *choosing the most urgent ones*. For instance, if observations $on(3,1)$ and $on(4,2)$ have activated the goals $serve(3,T_1)$ and $serve(4,T_2)$, the agent must be able to decide that the former is more urgent than the latter (apparently, i.e. it depends on the serving policy).

  So, *within the same plan* (the conjunction of goals in a node) the system must perform some kind of ordering or priority setting, based on the urgency of each task.

- There is a variant of "importance" that is not fully captured by ACTILOG rules. Sometimes, a certain course of action is said to be *more important than* some other course of action. The reason to "prefer" a certain course of actions, as opposed to others, may not be that it contains certain activated actions (in the case of the plans in our architecture, all the nodes/plans will contain all the activated goals), but that *it is more likely to achieve its top-most goals*.

  An agent will not only select actions within a plan according to some urgency criterion, but it will also choose some plan (within the frontier of goals that contains all the alternative courses of action for the agent), according to some quantitative or qualitative measurement of the **usefulness**, utility or likelihood of that plan to achieve the goals.

So, dispensing with the notion of "absolute importance", already captured by ACTILOG rules (or integrity constraints), we still have to extend our system to be able to program *preferences* between actions in a plan (the urgency criteria) and between plans (the usefulness criteria)[7]

5.3.2 Towards a qualitative formalization of preferences

Simon's bounded rationality [Sim55], was an attempt to go beyond the limitations of previous models of rationality in traditional economy theory. He wanted a new theory that did not make the assumption that the "economic man [.. has] a well organized and stable system of preferences, and a skill in computation that enables him to calculate, for the alternative courses of action that are available to him, which of these will permit him to reach the highest attainable point of his preference scale" (.ibid). Simon thought that those, by then classical, concepts of rationality made severe demands upon the choosing agent. In particular, such an agent should be able to attach definite "pay-offs" to each possible outcome of its actions. This implies that the nature of the outcome should be precisely defined (no uncertainty) and that pay-offs must be completely ordered. Thus, the agent is assumed to be *omniscient* with respect to its preferences.

One of the main concerns of the work reported in this thesis is to overcome the "omniscient agent" problem, albeit in a different sense. We want to capture the notion of an agent that *cannot reason about all the consequences of its beliefs because of its limited resources to compute them*. It is encouraging to find out that our models can incorporate all the notions in Simon's rationality and allow for qualitative criteria and subjective preferences. The mapping is as follows:

1. Simon's "economic or administrative" man is the agent modelled as a cycling process that interleaves observation, thinking and acting.

2. This agent has a set of "behaviour alternatives" represented here by the nodes in the frontier of derivations (chapter 3).

[7] This process of choosing between goals and between plans is similar to *conflict resolution* in production-rule systems.

Table 5.5 Resource – bounded List Ordering	
$order(List, List, 0)$	$[ORD-00]$
$order([], [], 1)$	$[ORD-01]$
$order([Item], [Item], 1)$	$[ORD-02]$
$order([FirstItem, SecondItem \mid Rest],$	
$\quad OrderedList, R)$	
$\quad \leftarrow precedes(FirstItem, SecondItem)$	
$\quad \wedge\ R_1 + R_2 + 1 = R$	
$\quad \wedge\ order([SecondItem \mid Rest], TempOrd, R_1)$	
$\quad \wedge\ order([FirstItem \mid TempOrd], OrderedList, R_2)$	$[ORD-03]$
$order([FirstItem, SecondItem \mid Rest],$	
$\quad OrderedList, R)$	
$\quad \leftarrow precedes(SecondItem, FirstItem)$	
$\quad \wedge\ R_1 + R_2 + 1 = R$	
$\quad \wedge\ order([FirstItem \mid Rest], TempOrd, R_1)$	
$\quad \wedge\ order([Second \mid TempOrd], OrderedList, R_2)$	$[ORD-04]$
$precedes(H_1, H_2) \leftarrow\ utility(H_2) \leq utility(H_1)$	$[PRE]$

Table 5.5: Resource-bounded List Ordering

3. The set of "future states of affairs", is implicitly represented as the logical consequences of the knowledge in the background theories and of the behaviour alternatives adopted by the agent (and other agents in the multi-agent setting).

4. The "preference order over future states of affairs" in Simon's theory is substituted here by a preference order over "behaviour alternatives", because these determine those future state of affairs. Note that this implies that 1) there is some computation involved in ordering alternative plans so that the "most preferred" are chosen for *further refinement and execution*, 2) the preference order is *internal* to every agent and 3) preferences may or may not be based on some computed pay-offs for each alternative.

One of the attractions of our model is that one can make the computation to "order the plans" resource-bounded, as was done with the *demo* predicate. The logic program in table 5.5 specifies a resource-bounded list ordering algorithm. Note that, with limited resources (such as limited time to compute) this algorithm will consider only a few elements of the list, leaving the rest untouched. As in *demo*, an argument of the predicate is devoted to "count" the resources spent on the ordering process. One can make explicit the limits on these resources (with something like $R < n$ as we did for *demo* in chapter 2).

Observe that the specification in table 5.5 captures the ordering principle but the actual ordering strategy depends on the way the resources R_1 and R_2 (for the recursive calls) are assigned. With $R_1 = (R-1)\ div\ 2$ and $R_2 = R\ div\ 2$, where *div* is the integer-division operator, and for large value of R, this logic program will behave like the well-known "bubble" algorithm for ordering lists.

Apart from the strict resource bounding provided by the third argument of *order* in table 5.5, there is another aspect that limits the capability of the agent to fully ordering its preferences. Observe that the ordering depends on an appropriate definition of the predicate *precedes* in

Table 5.6 Resource – bounded usefulness ordering

$use\_order(Goals, Goals, 0)$ [$USORD00$]

$use\_order(\mathbf{false}, \mathbf{false}, R)$ [$USORD01$]

$use\_order(\{Plan\}, \{Plan\}, R)$ [$USORD02$]

$use\_order(\{FirstPlan \vee SecondPlan \vee Rest\},$
$\quad OrderedGoals, R_1 + R_2 + 1)$
$\quad \leftarrow (\ prefers(FirstPlan, SecondPlan)$
$\quad\quad \vee\ indifferent(FirstPlan, SecondPlan)\ )$
$\quad \wedge\ use\_order(\{SecondPlan \vee Rest\}, TempOrd, R_1)$
$\quad \wedge\ use\_order(\{FirstPlan \vee TempOrd\}, OrderedGoals, R_2)$ [$USORD03$]

$use\_order(\{FirstPlan \vee SecondPlan \vee Rest\},$
$\quad OrderedGoals, R_1 + R_2 + 1)$
$\quad \leftarrow\ prefers(SecondPlan, FirstPlan)$
$\quad \wedge\ use\_order(\{FirstPlan \vee Rest\}, TempOrd, R_1)$
$\quad \wedge\ use\_order(\{SecondPlan \vee TempOrd\}, OrderedGoals, R_2)$ [$USORD04$]

$prefers(Plan_1, Plan_2) \leftarrow\ Plan_1 \equiv (\Delta_1, UC_1, CN_1, HF_1, M)$
$\quad \wedge\ Plan_2 \equiv (\Delta_2, UC_2, CN_2, HF_2, M)$
$\quad \wedge\ demoNonAb(KB \cup \Delta_1, H_1, \mathbf{true})$
$\quad \wedge\ demoNonAb(KB \cup \Delta_2, H_2, \mathbf{true})$
$\quad \wedge\ H_1 \neq H_2$
$\quad \wedge\ demoNonAb(KB \cup \Delta_1, H_2, \mathbf{false})$
$\quad \wedge\ demoNonAb(KB \cup \Delta_2, H_1, \mathbf{false})$
$\quad absolutely\_prefers(H_1, H_2)$ [$PREF01$]

$indifferent(Plan_1, Plan_2) \leftarrow\ \neg prefers(Plan_1, Plan_2)$
$\quad \wedge\ \neg prefers(Plan_2, Plan_1)$

Table 5.6: Resource-bounded, context-dependent preferences between plans

this case. In table 5.5, we suggest to use the traditional method in Decision Analysis ([Rai70], [Jon75]): One estimates the utility of each item, as an scalar value, and use it to decide on the precedence between them (greater values precede smaller ones). A hidden assumption is that *utility* is a total function, mapping every item to a value.

However, one could relax this last assumption and use a "partial" definition of *precedes*. This is exactly what we do in tables 5.6 and 5.7 where the logic program above is adapted to describe the mechanisms of plan and action ordering based on the agent's programmed preferences.

Thus, the problem of how to incorporate preferences into an agent is reduced to how to define the predicates *prefers* (or *absolutely_prefers*) in table 5.6[8] and *more_urgent* in table 5.7. What we do in the remaining sections of this chapter is to propose "surface syntaxes" for the logic programs that will provide those definitions. As we have done with OPENLOG and ACTILOG, this "syntactic sugar" is presented as two more independent programming languages.

Finally, observe that without any information about preferences the system will preserve

[8] *demoNonAb* is explained below.

Table 5.7 Resource – bounded urgency ordering	
$urg\_order(Goals, Goals, 0)$	$[URORD00]$
$urg\_order(\mathbf{true}, \mathbf{true}, R)$	$[URORD01]$
$urg\_order((Action), (Action), R)$	$[URORD02]$
$urg\_order((FirstAction \wedge SecondAction \wedge Rest),$	
$\quad OrderedPlan, R_1 + R_2 + 1)$	
$\quad \leftarrow$ ($more\_urgent(FirstAction, SecondAction)$	
$\quad \vee indifferent(FirstAction, SecondAction)$)	
$\quad \wedge urg\_order((SecondAction \wedge Rest), TempOrd, R_1)$	
$\quad \wedge urg\_order((FirstAction \wedge TempOrd), OrderedPlan, R_2)$	$[URORD03]$
$urg\_order((FirstAction \wedge SecondAction \wedge Rest\},$	
$\quad OrderedPlan, R_1 + R_2 + 1)$	
$\quad \leftarrow more\_urgent(SecondAction, FirstAction)$	
$\quad \wedge urg\_order((FirstAction \wedge Rest), TempOrd, R_1)$	
$\quad \wedge urg\_order((SecondAction \wedge TempOrd), OrderedPlan, R_2)$	$[URORD04]$
$indifferent(Action_1, Action_2)$	
$\quad \leftarrow \neg more\_urgent(Action_1, Action_2)$	
$\quad \wedge \neg more\_urgent(Action_2, Action_1)$	

Table 5.7: Resource-bounded, context-dependent preferences between actions

the ordering of actions suggested by the text of OPENLOG procedures or ACTILOG rules. This is a kind of text-order, default priority.

5.4 PRIOLOG: the logical language of priorities

PRIOLOG must be used to encode knowledge about the relative urgency of actions. This knowledge will be used by the procedure *urg_order*, invoked by *demo* (chapter 3), to order literals within a node for selection and further processing. The syntax of the language is given in table 5.8.

Figure 5.2 shows a set of PRIOLOG rules[9] that could be used to obtain different behaviours in the elevator controller. Figure 5.3 shows the set of PRIOLOG rules used by the elevator to implement policy 3 (chapter 4). A compiled version of these rules was used in an implementation discussed in chapter 6.

These rules can be read as clauses in a normal logic program. The intention is to translate them into the definition of the meta-predicate *more_urgent*, called by *urg_ord*.

$$more\_urgent(KB, T, \Delta, D1, D2) \tag{5.5}$$

Notice that the meta-predicate is provided with "contextual" information from the agent including the definition of *holds* (in KB), other actions and observations in the plan from which the comparing actions are taken (in Δ) and the current time (T, accessible through the term **now** i.e. the parser will have to replace the constant **now** with the variable T designating the *current time* according to the *cycle* predicate)).

[9] We assume that the interpreter admits comments as in PROLOG: anything to the right of a "%" is a comment.

Table 5.8 PRIOLOG Language: Syntax			
Urg_Set	::=	*Urg_Rule* (**and** *Urg_Set*)*	*Urgency Set*
Urg_Rule	::=	**consider** $TaskName_1$	
		before $TaskName_2$ **if** *Body*	*Priority Rule*
Body	::=	*Condition* (**and** *Body*)*	*Body of an IC*
	|	**true**	
Condition	::=	$Func_{boolean}$ **at** *Term*	*Conditions*
	|	*TaskName* **starts at** *Term*	
	|	*TaskName* **finishes at** *Term*	
	|	*TaskName* **starts before** *TaskName*	
	|	*TaskName* **finishes before** *TaskName*	
	|	*TaskName* **'s earliest start is** *Term*	
	|	*TaskName* **'s latest finish is** *Term*	
	|	*Query*	*Tests on "rigid" information*
	|	**not** *Condition*	
TaskName	::=	$Func_{action}$	*Action names*
	|	$Func_{proc}$	
	|	*TaskName* (; *TaskName*)*	
	|	*TaskName* (**par** *TaskName*)*	
$Func_{action}$	::=	...	*As in OPENLOG*
$Func_{proc}$	::=	...	*As in OPENLOG*
$Func_{fluent}$	::=	...	*As in OPENLOG*
$Func_{boolean}$	::=	...	*As in OPENLOG*
Term	::=	*Ind* | *Var*	*As in OPENLOG*
Ind	::=	...	*As in OPENLOG*
Var	::=	...	*As in OPENLOG*

Table 5.8: Syntax of PRIOLOG

```
% Chronological ordering..
consider D1 before D2 if
    D2 starts at T2 and
    D1 finishes at T1' and T1' lt T2'

% First to call, first to be served..
consider D1 before D2 if
    D1's earliest start T1 and
    D2's earliest start is T2 and
    T1 le T2

% Pure Shortest path..
consider serve(N1) before serve(N2) if
    currentfloor( L ) at now and
    abs( N1 - L ) < abs( N2 - L )

% Minimal waiting time..
consider serve(N1) before serve(N2) if
    serve(N1) 's earliest start is T1 and
    serve(N2) 's earliest start is T2 and
    abs( T1 - now ) > abs( T2 - now )

% finish everything before you stop..
consider D1 before park if true
```

Figure 5.2: Examples of PRIOLOG rules

```
consider turnoff(N) before D2 if
    currentfloor(N) at now

consider serve(N) before serve(M) if
  going_up at now and
  N lt M and
  currentfloor(H) at now and H lt M and H lt N

consider serve(N) before serve(M) if
  going_up at now and
  M lt N and
  currentfloor(H) at now and M lt H and H lt N

consider serve(N) before serve(M) if
  going_down at now and
  N lt M and
  currentfloor(H) at now and N lt H and H lt M

consider serve(N) before serve(M) if
  going_down at now and
  M lt N and
  currentfloor(H) at now and N lt H and M lt H
```

Figure 5.3: PRIOLOG rules used by the elevator

```
consider serve(N1) before serve(N2) if
     currentfloor( L ) at now and
     abs( N1 - L ) < abs( N2 - L ) and
     serve(N2)'s earliest start is T2es and
     estimated_duration( serve(N1), Duration ) and
     maximum_waiting_time_per_passenger( MWTPP ) and
     now + Duration < T2es + MWTPP
```

Figure 5.4: Policy 4 for the elevator controller

There is also some additional processing of data provided by the interpreter of this language. Conditions of the form D **'s earliest start is** T, for instance, refer to the "activation time" of task D or more simply, to the lowest bound of the actual starting time of the task.

Example 5.4.1 In $t_o < T_1 \wedge T_1 < T_2 \wedge done(serve(3), T_2, T_f)$, t_o is the earliest starting time for task $serve(3)$. The corresponding treatment can also be provided for the operator "**'s latest finish is**".

The PRIOLOG language is sufficiently expressive that we can now offer a more interesting solution for the elevator controller in figure 5.4.

5.4.1 The elevator controller for policy 4

The policy that the clause in figure 5.4 formalizes is: *consider serving floor* $N1$ *before floor* $N2$ **if** *you are at* L *now and* L *is closer to* $N1$ *than to* $N2$ **and** *by going to* $N1$ *you are not likely to exceed the "maximum-waiting-time limit" for those clients at floor* $N2$.

This PRIOLOG clause, an OPENLOG code defining the $serve(N)$ procedure and an ACTILOG unit indicating the conditions for activation of $serve(N)$ goals, constitute a complete specification of the elevator controller (for policy 4)[10]

5.5 USELOG: programming the usefulness criterion

The language USELOG is also "syntactic sugar" for logic clauses. It encodes heuristic knowledge about the relative usefulness of plans to achieve goals. *demo* has access to this knowledge through the logic program *use_order*.

The issue of preference between plans is slightly more complicated than preference between actions. As we said above, agents manifest preferences between "future states of affairs" or, following von Wright[vW63], a preference statement is a statement about situations.

Comparing two situations by exhaustively analysing what holds in one versus what holds in the other is likely to be a task of enormous complexity. Some reduction of complexity can be achieved by a more restrictive interpretation of preference statements. It was von Wright (.ibid) who suggested that when an agent "prefers oranges to apples" then it will prefer the situation where it has an orange *and no apple* to the situation where it has an apple but *no orange*. Huang and Masuch call this the *conjunction expansion principle* [HMP92] and suggest that is a good idea to restrict attention to preference statements that obey the principle.

[10] This is if one takes the specification of *cycle* and *demo* for granted.

```
prefers P1 to P2 if
     P1 implies F1 and P2 implies F1 and
     probability of F1 given P1 is V1 and
     probability of F1 given P2 is V2 and
     V2 lt V1

prefers P1 to P2 if
     energy_consumption(P1, E1) and
     energy_consumption(P2, E2) and E1 le E2
```

Figure 5.5: Example of USELOG rules.

The *conjunction expansion principle* is formalized in our framework by the entry [PREF01] in table 5.6. The idea is to have *a quick mechanism* embodied by the program $demoNonAb$[11] in table 5.6, (which also implements **implies**[12]).

This program, which could be a deductive-only version of *demo* (or *demo* plus inhibition of abduction, as we explain in chapter 6), could be used to obtain properties H_1 and H_2 that *hold after the execution of plans* $Plan_1$ *and* $Plan_2$*, respectively*, but that do not hold after the other plan in each case[13]. One can then ask for the absolute preference between P_1 and P_2. And, one can also have the case when neither plan is preferred to the other.

However useful, the *conjunction expansion principle* is just one criterion to guide the choice between situation or plans. One should leave open to the programmer of the agent the possibility of encoding other kinds of criterion, so long as it does not lead to inconsistency of preferences. A more general criterion will probably be based on a mixture of qualitative knowledge and estimations of probabilities, if they are available. This is the purpose of the syntactic constructs: **implies**, **does not implies**, **probability of** and **utility of** in the USELOG language, as shown in table 5.9.

With USELOG one can write preference clauses like those in figure 5.5.

The first clause assumes that one has a mechanism, perhaps some Bayesian formalization, to compute probabilities. The second uses a program to estimate energy consumption generated by the execution of each plan. These diverse types of information can be incorporated into the agent in order to improve its effectiveness at problem solving.

5.6 Discussion

This chapter and the previous one have presented a family of languages to program an agent. The characteristic common to all these languages is that their sentences have an unambiguous translation into subsets of first order logic. In the case of OPENLOG, the translation has a more restrictive output, yielding *normal logic programs*. In the case of ACTILOG the translation is into a form that supports sentences formalizing integrity contraints, that can be used to guide the process of activation of goals in the agent. PRIOLOG and USELOG translate into logic

[11] Notice that, for the sake of simplicity, we have omitted the resource argument in $demoNonAb$. This argument, however, is essential to guarantee that the mechanism is indeed *quick* and terminates within some predefined window of time.

[12] That is, to test whether P **implies** F (as in table 5.6 and in the example in figure 5.5 below), the system will test $demoNonAb(KB \cup \mathbf{P}, \mathbf{F}, \mathbf{true})$.

[13] In the notation in table 5.6, $demoNonAb(K', H', \mathbf{true})$ means that H' is implied by K' and $demoNonAb(K', H', \mathbf{false})$ means that H' is not implied by K'.

Table 5.9 USELOG Language: Syntax			
USEFUL_Set	::=	*USEFUL_Rule* (**and** *USEFUL_Set*)*	*Utilities Set*
USEFUL_Rule	::=	**prefer** *Plan* **to** *Plan*	
		if *Body*	*Usefullness Rule*
Body	::=	*Condition* (**and** *Body*)*	*Body of an IC*
	|	**true**	
Plan	::=	...	*A conjunction of literals*
Fact	::=	$Func_{boolean}$ **at** *Term*	*Facts*
Condition	::=	*Plan* **implies** *Fact*	
	|	*Plan* **does not imply** *Fact*	
	|	**probability of** *Fact* **is** *Term*	
	|	**probability of** *Fact* **given** *Plan* **is** *Term*	
	|	**utility of** *Fact* **is** *Term*	
	|	*Query*	*Tests on "rigid" information*
	|	**not** *Condition*	
$Func_{action}$	::=	...	*As in OPENLOG*
$Func_{proc}$	::=	...	*As in OPENLOG*
$Func_{fluent}$	::=	...	*As in OPENLOG*
$Func_{boolean}$	::=	...	*As in OPENLOG*
Term	::=	*Ind* | *Var*	*As in OPENLOG*
Ind	::=	...	*As in OPENLOG*
Var	::=	...	*As in OPENLOG*

Table 5.9: Syntax of USELOG

programs and provide a way to blend domain or problem specific knowledge with the general purpose, reasoning mechanism of the agent.

The predicates *use_order* and *urg_order* support a resource-bounded system of preference for agents that can be programmed in the languages PRIOLOG and USELOG, and that can be integrated into the structure of the embedded proof procedure (*demo*) as shown in chapter 3.

These developments, applied to the architecture in the previous chapter, will support a fine tunning of agent performance for specific applications. However, there are still several important remarks to make about general-purpose, reactive planning. These considerations and the specification of an algorithm for reactive planning are the subjects of the following chapter 6.

Chapter 6

The Agent's Planning Mechanism

This chapter describes an adaptation of the **iff** proof procedure (presented in chapter 3), to be used as the planner of our agent. We will also discuss a prototype of GLORIA that simulates the elevator controller.

Before the adaptations to **iff**PP, a brief review of planning literature.

6.1 A brief history of automatic planning

6.1.1 STRIPS (1971)

STRIPS [FN71] was not the first planner but it is probably the most widely known. It is based on a classical *state-based* representation of a problem (as described in chapter 1) in which actions are state-transitions. Action types are specified by *operators* with pre and post-conditions. An operator establishes which (post) conditions must be **added** to and which (pre)-conditions must be **deleted** from the description of the current state to yield a new state. Goals are specified by a set of conditions that must hold in the final state.

Working on a complete description of the initial state, STRIPS searches for a state in which all the goals' conditions and actions' preconditions are satisfied. This search is performed by simulating the execution of actions.

With its use of *add-delete* lists of conditions, STRIPS incorporates a form of *default* reasoning. Properties that are not explicitly affected by an action (and therefore are not mentioned in the add-delete lists of its related operator) are left untouched when an state description is transformed into a new state description. As long as the effects of actions do not depend on their contexts, this form of default reasoning is enough to deal with the frame problem.

Search in STRIPS is guided by *means-end analysis*, a technique inherited from GPS[NSS60]: only those operators that contribute to the achievement of the goal are considered in the search for a solution. Although *means-end analysis* can be used to reason forwards (from initial state to the goal state), STRIPS only reasons backwards (from goals to initial state).

STRIPS has been used in many applications, including the embedded planner of SHAKEY, the robot at SRI[RN95]. However, the fact that it generates fully instantiated plans, with completely specified and fully ordered actions, with no context-dependent effects and with no notion of duration, makes this approach too limited for general planning.

6.1.2 ABSTRIPS (1974)

ABSTRIPS [Sac74] main contribution was the introduction of *hierarchical planning*. In ABSTRIPS, pre-conditions of actions are assigned a value indicating their *criticality*. The planner looks first at those conditions with the highest criticality. With a careful (*ad hoc*) setting of criticalities, the designer can overcome the potential combinatorial explosion that occurs in systems which perform *means-end* analysis only.

Hierarchical planning can be applied to action specification. To obtain a plan, for instance, to **build_a_structure**, one can use the definition:

```
build_a_structure if get_parts(P) and ensemble_parts(P).
```

Thus, given the goal **build_a_structure**, a planner would reduce it to **get_parts(P)** and **ensemble_parts(P)**. It could then use the output of **get_parts(P)** (an instantiation for P) to *focus* the search performed for **ensemble_parts(P)**. This is better than the exhaustive matching performed by *means-end* analysis on a grounded equivalent representation.

Apart from the hierarchical organization of goals, ABSTRIPS is equivalent to STRIPS and therefore it inherits all the aforementioned limitations.

6.1.3 WARPLAN (1974)

WARPLAN [War74] was the first planner completely written in PROLOG. It served to show the capabilities of the language to do high level, very compact programming: WARPLAN consisted of approximately 100 lines of code.

A version of WARPLAN (WARPLAN-C [War76]) could do "conditional planning": plans could contain conditional expressions to be tested and decided on at execution time. Also, the report in which WARPLAN was presented [War74], was the first to mention (on page 16) the idea of *partial planning* (also known as *nonlinear planning*) that became the main breakthrough in planning technology a few years later.

6.1.4 NOAH (1975)

NOAH [Sac75] was designed by the creator of ABSTRIPS and introduced a very influential notion: *least commitment* plan generation. In NOAH, a plan is, initially, a partially ordered set of actions. By *posting ordering constraints* [1] only when they are strictly required by the problem description or by action interferences, the chances of finding a feasible plan (without having to backtrack) increase.

NOAH inherits the idea of abstract hierarchies from ABSTRIPS. However, NOAH is not capable of backtracking (that is, the system does not store decision points so that it can come back to them when some branch of exploration fails). And because actions are indexed to consecutive global states, actions must be fully ordered in a complete plan.

6.1.5 NONLIN (1976)

NONLIN ([Tat76],[Tat77]) extended NOAH by incorporating search capabilities (and therefore backtracking) into the planner. The other important characteristic of NONLIN is that it does not search on domain states (states with descriptions of properties of the world), but on *plan states* (states containing descriptions of actions, ordering contraints and goals). When one represents an abstract hierarchy of actions, as we did above for *build_a_structure*, this is the type of search space that is generated.

[1] This is the expression used in the planning jargon to indicate that one is introducing constraints on a plan description. "Imposing constraints" is a clearer expression, but it is not used.

6.1.6 MOLGEN (1981)

MOLGEN [Ste81] applied the principle of *least commitment* to the choice of objects to be manipulated by actions. In logical terms, this requires from the system the capability to reason about objects whose existence is known, but whose identity is not. The abductive proof procedure described in chapter 3 is capable of this type of reasoning.

Another capability of the **iff**PP, which is very important in planners since NOAH and is also used in MOLGEN, is that of *posting constraints* mentioned above. The process by which the **iff**PP *abduces* certain atoms, inequalities in particular, can be seen as a mechanism for increasingly refining a set of constraints on a plan.

6.1.7 DEVISER (1983)

DEVISER [Ver83] is a partial planner similar to NONLIN. DEVISER allows numeric constraints on actions execution times. Each action is associated with a start time window and a duration. By using numbers (time intervals and points) instead of states as index, DEVISER supports some limited forms of concurrency. However, an important feature of DEVISER is that it "allows plans to be generated which take account of scheduled changes [..] occurring *after* the start of plan execution time" [Lin93]. Thus, this system contains a basic element for the interleaving of planning and execution. Our system shares that feature with DEVISER.

6.1.8 Interval Logic Planner (1983)

James Allen pioneered the study of logic-based planners. His "general theory of action and time"[All84] served as the logical framework for a Planning algorithm described in [AK90] and [All83] which Pelavin calls the Interval Logic Planner[Pel91]. The main element of that logical framework is a temporal logic based on intervals. Thirteen relations (*before*, *equal*, *meets*, *overlaps*, *during*, *start* and *finishes*) constitute the set of possible relations that can hold between intervals (Later he and P. Hayes proved that interval relations can be defined in terms of *meets* only, but that "there are important efficiency gains from using the larger set of primitives"[AH87]).

The heart of the Interval Logic Planner is an algorithm that computes the transitive closure of the primitive relations mentioned above. It does this by "posting" and propagating[2] constraints through a network that stores all the information about how the intervals are related. However, verification of (global) consistency in such a network is intractable (as Allen admits in [All83], pg. 836). Lingard [Lin93] and Pelavin [Pel91] also point out that the Interval Logic Planner is not able to detect "destructive synergistic interference" between actions in a plan.

6.1.9 TWEAK (1987)

TWEAK [Cha90] is a formally defined partial planner which operates by posting (imposing) constraints on action ordering and object descriptions.

To design TWEAK, Chapman first established a *modal truth criterion* which, just like the definition of *holds* (the temporal projection predicate described in chapter 4), can be used to establish whether some proposition is (in his case, necessarily or possibly) true in some situation. Then, using the truth criterion as a scheme, he devised a "nondeterministic achievement procedure" to produce an ordering of the actions that achieve some given goals. Chapman noticed the importance of equality treatment. He stated a relaxed form of equivalence between

[2] This is essentially the same as computing the transitive closure of the orderings between intervals. The word "propagating" is also part of the planning jargon.

terms that he called "co-designation" ($\approx$) and that allows expressions such as `(on x y)` $\approx$ `(on v z)`, to be considered as constraints on the variables involved (x, y, v, z in this case).

The following is a rendering of Chapman's modal truth criterion in a language similar to those introduced in chapter 4 for easy comparison:

$$\begin{aligned} holds(P,S) \leftarrow\ & holds(P,T) \ \wedge\ T < S \\ & \wedge \neg clip(T,P,S) \\ & \vee\ do(A,T) \ \wedge\ initiate(A,P) \ \wedge T < S \\ & \wedge \neg clip(T,P,S) \qquad\qquad [NHOLDS] \end{aligned}$$

$$\begin{aligned} clip(T,P,S) \leftarrow\ & do(C,T') \ \wedge\ terminates(C,P) \\ & \wedge\ T \leq T' \ \wedge\ T' < S \ \wedge \neg declip(T',P,S) \qquad [NCLIP] \end{aligned}$$

$$\begin{aligned} declip(T',P,S) \leftarrow\ & do(W,T'') \ \wedge\ initiates(W,P) \\ & \wedge\ T' \leq T'' \ \wedge\ T'' < S \qquad\qquad [NDECLIP] \end{aligned}$$

Read $do(A,T)$ as: do step A in situation T; $initiates(A,P)$ as: step A asserts P; and $terminates(C,P)$ as: A possibly asserts a property R that possibly co-designates with the negation of P.

[NCLIP] is, in Chapman's terminology, the axiom of the *clobberers* (step C is "clobbering" proposition P). [NDECLIP] is the axiom of the *white knights* (W in this case, is the white knight that re-installs P).

Because of his use of "co-designation", Chapman's specification requires [NDECLIP]. As can be seen, the axiom is logically redundant and so the description can be made even closer to those axioms for OAEC in chapter 4.

Recently, Missiaen *et al* [MBD95] built a similar rendering of Chapman's truth criterion in the language of the Event Calculus. They use Chapman's truth criterion in the engine of the planning system CHICA (.ibid). Missiaen *et al* also use an axiom similar to [NDECLIP], which they justify for efficiency reasons.

Chapman proved that TWEAK was correct and complete [Cha90]. To do so, he had to restrict the representation used by the planner. Notice that *initiate* and *terminate* above do not have a time index as their analogue in OAEC in chapter 4. This means that the action representation does not allow "the effects of actions to depend on the situation in which they are applied"(.ibid). It does not allow for indirect side-effects either. Chapman even shows an example of synergistic interference that cannot be accounted for by his system. Chapman carefully studied the limitations of his system and he went on to prove that if one tries to ignore the restrictions, one would be confronting these two theorems (which he proved):

Theorem 6.1.1 Intractability Theorem (Taken from [Cha90]). The problem of determining whether a proposition is necessarily true in a nonlinear plan whose action representation is sufficiently strong to represent conditional actions, dependency of effects on input situations, or derived side-effects is NP-hard.

Theorem 6.1.2 Second Undecidability Theorem. (Taken from [Cha90]). Planning is undecidable even with a finite initial situation if the action representation is extended to represent actions whose effects are a function of their situation.

According to Chapman, these theorems suggest that "writing planners for extended action representations is a *quixotic* enterprise". And, when trying, one may either: 1)"hope for the best", and look for some trick to improve efficiency, 2)"relax the correctness requirement" and

produce plans that may not work or 3) "relax the generality requirement" by allowing, for instance, domain-specific criteria to guide the planner.

Following our strategy of avoiding concerns about efficiency and implementation details, we have implicitly chosen alternative 3). We count on the possibility of providing domain-specific information (with rules as those shown in chapter 5) about importance, urgency and preferences, to improve the ability of the planner to reach a solution. In a paper after TWEAK's Chapman and Agre ([AC90]) acknowledge that "Complexity theory is, unfortunately, not and ideal tool for proving negative results. [..] heuristic solutions might work well enough in practice".

Chapman's pessimistic results led him and others to explore radically different approaches to planning. This resulted in the exploration of *reactive planning* and reactive platforms, some of which we discuss in the following section.

6.1.10 O-PLAN (1985)

The **O-PLAN** system[CT91] is a complex planning architecture. It inherits and extends the capabilities of previous planners (such as NONLIN and DEVISER) by allowing, for instance, numerical constraints on actions' and goals' durations and on consumable resources.

The designers of O-PLAN seem to have chosen the third alternative of those mentioned above (i.e. "relax the generality requirement") to confront the intractability problem. One of the most attractive features of O-PLAN is the use of a *blackboard control architecture* that incorporates domain specific knowledge to allow the system to "focus" its search for plans.

Thus, automatic planning seems to be headed towards the development of domain specific, highly customized systems that could be quickly adapted to new requirements and use heuristics. In this context, the possibility of quick prototyping that characterizes logic programming in particular, can be crucial.

6.2 Reactive Planning (1986-1989-1991)

6.2.1 What is reactive planning

Reactive planning is a mode of planning that requires from the planner a swift decision about what to do next, considering mainly (and sometimes only) what it perceives about its current situation. The fundamental idea seems to be to try to establish a direct (programmed or hard-wired) connection between an agent's perceptors and its effectors.

The reactive approach is sometimes called "situated" because it is the current situation of the agent that determines what it will do next.

The reactive approach is so different from traditional planning that one could wonder if it is planning at all. The reason that it can still be called planning is that the agent can still be seen as "deciding" what it will do next, and the decision can be anything from "do nothing" to "do these actions in parallel". The basic insight is that *a careful arrangement of perceptor-effector connections is normally sufficient for the agent to have "meaningful" behaviour*, even though it may not be optimal or even correct.

Notice that we are now talking about an "embodied agent": an agent with a body, with perceptors and effectors and a set of "indexes" to its current situation (i.e. *now*, *here* and *self*). There does not seem to be any need for a planning module to reason about actions, separated from the *executive* or even from the *perception* unit. All of them are fused in one whole perception-reaction unit.

Four well-known projects have pioneered this approach to agent construction (more than an approach to planning itself): **Brook's subsumption architectures** [Bro86], proposing the

rejection of symbolic representation as we explained in the introductory chapter 1, **PENGI** [AC90]: Agre and Chapman's attempt to go beyond the limitations of traditional planning discussed in Chapman's paper [Cha90], **Situated Agents** [RK95], a rich platform to systematically generate "reactive agents" and Maes' **Agent Network Architectures** [Mae91], another attempt to build flexible and reactive entities. All four groups avoid the use of logic as a representation language. Brooks completely rejects representations, Chapman and Agre are ambiguous in that respect whereas Rosenschein, Kaelbling and Maes still concede that logic could be used as an specification language and that their systems could be made sense of in a logical description. Maes subscribes to the view that "reactive agents can have goals"[Mae91]. Her sets of *competence modules* to implement an agent, despite having the basic structure of neural networks, are described in declarative terms.

There have been other important efforts to integrate a planner with a module for reactive behaviour, sometimes under the names of *interleaved planning* and *heuristic planning* (See [AC90] for an overview) and *universal planning*. Georgeff and Lansky [GL90] proposed PRS (*Procedural Reasoning System*), a system that combines a database (for beliefs about the world), a set of current goals (or *desires*), a set of procedures (which they call *Knowledge Areas*), explaining what to do to achieve the goals, and an interpreter for manipulating all these components. The system is inspired by the Belief-Desire-Intention paradigm [Bra87] and was used to control FLAKEY, a robot in a space station scenario. Other systems include IPEM [AIS88], which integrated partial planning with execution and Poole's logic programs for Robot Control [Poo95]. All these systems rely on some kind of *condition* $\rightarrow$ *action* rules by which inputs are related to outputs.

6.2.2 Criticism of Reactive Planning

Reactive planning has been analysed by Ginsberg in [Gin89] (who calls it universal planning). He proves that "even if the compile-time costs of the analysis are ignored, the size of the table must, in general, grow exponentially with the complexity of the domain". By "the table", he means the arrangement of input-condition and outputs-actions that the systems must somehow store to be able to relate every identified situation to an action. If there are n independent, sensory inputs to be dealt with and a actuators (outputs), then "there are $(2^a)^{2^n}$ distinct universal plans" (.ibid, proposition 1.2). Thus, it is simply impractical, Ginsberg argues, for "an agent to precompute its response to every situation in which it might find itself" (.ibid).

One must notice that an agent need not be a purely reactive entity. An agent with explicit goals (or intentions, a closely related concept), can use them, not only to decide what to do next by computing a plan from an action theory, but also to focus its subsequent sensory and reasoning processes. Among the triggered actions the agent can have sensory actions that set particular sensors with particular control parameters.

One must also consider the effect on reactivity of a rich but compact knowledge representation. A theory of action is not just a useful tool for analysis. It can be used to generate a representation that implicitly captures the same information a "table" of condition-actions rules would have to make explicit.

Thus, logic is likely to be more critical to the generation of efficient and effective behaviour than what the defenders of reactivity may have thought, even without considering complex activities, such as language manipulation, communication and social interaction.

The review above is intented to highlight the critical aspects of automatic planning. In the following section, we concentrate on the logic programs that defined the planning system of GLORIA-like agents.

6.3 The planning programs

The specification of GLORIA's planner is identical to the description of the **iff**PP given in chapter 3, except for two major modifications that are presented in the following subsections. Both extensions are motivated by practical considerations. However, they bring about fundamental changes in the way the logic of the whole system must be understood.

6.3.1 A reason to inhibit abduction in OPENLOG programs

In chapter 3 we explained how the **iff**PP deals with negation. Negative literals in a node are written as implications with **false** as a head. Those literals in the body of these implications are then treated almost as they would if they appeared as positive literals in regular goals. The only difference is that *abducibles* appearing in the body of the implications *are not abduced.* Abduction in the body of an implication should be **inhibited**. Abducibles in that position should only be processed by the propagation rule.

We also explained in chapter 3 that this "context dependent inhibition of abduction" is actually required by the semantics of abductive logic programs for the sake of minimality. In this section, we take this idea further and propose inhibiting abduction in other contexts than in the context of negation.

We inhibit abduction "below" (in the proof tree of) the temporal projection predicate **holds**. Just like the inhibition "below" a negative literal's proof. This may seem like an *ad hoc* solution. It is not. It is true that this extension specializes the proof procedure with respect to a particular predicate (*holds*). But this is not a domain specific solution. It is to be used in any domain and in any problem description.

Now, why do we do it? Almost for the same reason that Fung and Kowalski prevent abduction below negations: to obtain a minimal set of abducibles. Unlike with negations, not inhibiting below *holds* may generate correct solutions for the abductive procedure (in the sense that it may produce a set of actions that achieves a goal.). But it may also generate *too many* and *too big* solutions. Solutions which are incorrect for a planner[3] and even worse, totally unuseful for a reactive planning (where correctness could not be an issue)[4].

The fundamental reason to inhibit abduction below *holds* is that doing so allows us to solve the problem of *over-generation* of abducibles (discussed in chapter 4). The gain is that the system can now distinguish between abduced (hypothetical) information (such as its own plans) and input information (data that comes from the environment). Notice that the important feature is that the system distinguishes between the two sources of information but it treats them identically when it comes to reason about the future with the projection predicate. This implies, for instance, that the agent equally believes that *the wall is white* at time t when it sees its colour before t and does not expect any changes to happen, **and** when it decides to paint the wall by itself before t (even though it has not painted it yet i.e. it is just an intention).

The operational details can be clarified by the examples in the following section. Before that, however, let us explain that a very important consequence of inhibiting abduction is that *the behaviour dictated by an OPENLOG program is identical to the behaviour dictated by an ACTILOG program.*

[3] In general, it could produce actions that cannot be performed because they are out of the agent capabilities. See the examples in chapter 4 in the discussion about over-generation of abducibles.

[4] One may be puzzled by solutions that are correct for the proof procedure but not for the planner. The answer to the puzzle is that, when using the proof. proc. as a planner, one requires a more precise interpretation for abducibles: "the things that the agent wants to do" as specified by its programs or integrity contraints. In the general case, they are "the things that may happen" as predicted by the theory of actions. See the discussion about over-generation of abducibles (chapter 4)

6.3.2 Making OPENLOG equivalent to ACTILOG

The examples in this section are a confirmation of our initial conjecture that procedural knowledge could be added to an agent as OPENLOG procedures or as ACTILOG rules. From now on, only efficiency matters will have to be considered when one chooses between the languages.

As explained in [KS97], this also means that one can write logical procedures with embedded condition-action rules. An **if .. then** construct in OPENLOG can now be understood as a production rule. To illustrate this, consider the following examples[5]:

Example 6.3.1 Consider the abductive logic program:

```
h(P,T) :-
  do(A,T2), T2 lt T, init( A, T2, P ), persists( T2, P, T ).

init( a1, _, p ).

Tq le T2 :- T1 eq T2.
Tq le T2 :- T1 lt T2.

abd(do).
abd(persist).
```

in which *do* and *persists* are declared as abducibles and no predicate has been marked for inhibition of abduction.

When the logic program above is queried with:

$$
\begin{aligned}
G \equiv\ & do(a1,1) \wedge (do(a2,T) \leftarrow\ h(p,T)) \\
& \wedge\ (\exists T2\ (T\ lt\ T2\ \wedge\ persists(T,P,T2)\ \leftarrow\ do(A,T)\ \wedge\ init(A,T,P))) \\
& \wedge\ (\mathbf{false} \leftarrow persists(T1,P,T3)\ \wedge\ do(A,T)\ \wedge\ term(A,T,P) \\
& \qquad \wedge\ T1\ le\ T\ \wedge\ T\ lt\ T3\ )\ )
\end{aligned}
$$

the answer generated by the prover is (Recall that *le* and *lt* are $\leq$ and $<$, for the prover and that variables only in implications are implicitly universally quantified):

```
Node 1

 % Delta[1] = {  do(a1, 1),  1 lt G17484,  persist(1, p, G17484),
 do(a2, G17484),  };
 % UC[1] = {  };
 % CN[1] = { ... }
 % HF[1] = { [] }
```

where $T2 = G17484$ (For simplicity, we omit the content of CN).

Example 6.3.2 Now, consider the program:

```
h(P,T) :-
  do(A,T2), T2 lt T, init( A, T2, P ), persists( T2, P, T ).
```

[5] The notation is as in chapter 3: `Delta[i]` refers to the set of abducibles in node i, `UC[i]`, to the unconditional goals still to be processed, `CN[i]` to the implications in the node, and `HF[i]` is the history of factoring of the node.

```
init( a1, _, p ).

done(T) :- h(p,T), do(a2,T).

Tq le T2 :- T1 eq T2.
Tq le T2 :- T1 lt T2.

abd(do).
abd(persist).

for_testing_only(h(_,_)).
```

in which again *do* and *persists* are declared as abducibles and the system has been set up (with $for\_testing\_only(h(\_,\_))$ to inhibit abduction below $h(P,T)$, the answer to the query:

$$\begin{aligned} G \equiv \quad & do(a1,1) \wedge done(Ts) \\ & \wedge (\exists\ T2\ (T\ lt\ T2\ \wedge\ persists(T,P,T2)\ \leftarrow\ do(A,T)\ \wedge\ init(A,T,P))) \\ & \wedge (\mathbf{false} \leftarrow\ persists(T1,P,T3)\ \wedge\ do(A,T)\ \wedge\ term(A,T,P) \\ & \qquad \wedge\ T1\ le\ T\ \wedge\ T\ lt\ T3\ ) \end{aligned}$$

is:

```
Node 1

 % Delta[1] = {  do(a1, 1),  1 lt G8340,  persist(1, p, G8340),
 do(a2, G8340),  };
 % UC[1] = {  };
 % CN[1] = { .. } ;
 % HF[1] = { [] }

Node 2

 % Delta[2] = {  do(a1, 1),  1 lt G8340,  persist(1, p, G8340), };
 % UC[2] = {
        t :: (1 lt G20984, init(a1, 1, p), persist(1, p, G20984),
              true), p :: (do(a2, G20984), true),  };
 % CN[2] = { .. } ;
 % HF[2] = { [G20984 eq G8340] }

Node 3

 % Delta[3] = {  do(a1, 1),  1 lt G8340,  persist(1, p, G8340),  };
 % UC[3] = {
        t :: (do(G20988, G20992), G20992 lt G20984,
              init(G20988, G20992, p), persist(G20992, p, G20984),
              true), p :: (do(a2, G20984), true),  };
 % CN[3] = { .. } ;
 % HF[3] = { [G20988 eq a1,G20992 eq 1] }
```

In examples 6.3.1 and 6.3.2, the proof procedure produces essentially *the same answer*[6] to different queries. In example 6.3.1, the "dependency" of $do(a2, T)$ on $h(p, T)$ is expressed as an integrity constraint. In example 6.3.2, the same dependency is made part of the structure of the predicate *done*. In both cases (and using the planning interpretation) the execution of the action $a2$ at time T is subject to $h(p, T)$.

Notice that the definition of *done* in the abductive logic program above could be obtained by partially evaluating [Hog90] the following OPENLOG program (using the definition of *done* in table 4.2, chapter 4):

```
proc goal begin
  if p then a2
end
```

whereas the integrity contraint would be written in ACTILOG as:

```
if p at T then a2 at T
```

6.3.3 Inhibition of abduction and reactivity

In this section we illustrate with an example the relationship between inhibition of abduction and reactive planning. In this mode of planning, the basic strategy is to check the state of the environment and to take action as soon as possible. The action should, of course, be within the capabilities of the agent. To illustrate these ideas, consider the scenario in figure 6.1:

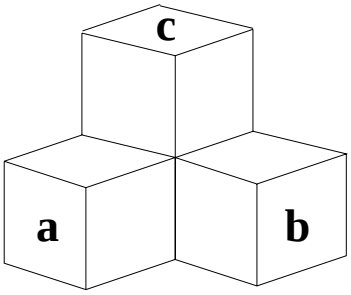
An agent is presented with the challenge of climbing a mountain of blocks. The agent can climbing one block at a time provided, of course, that the block is there. The planning problem is then to decide which blocks to climb onto and in which order. An OPENLOG procedure to guide this planning could be:

```
proc climb
  begin
    if infront(A) and currentlevel( C )
       and A is_higher_than C then
         begin
           step_on( A ) ; climb
         end
  end
```

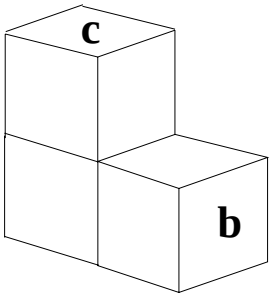
So, given the scenario in figure 6.1 a), the agent will try to generate the alternative plans $do(step\_on(a), 1) \wedge do(step\_on(c), 2)$ and $do(step\_on(b), 1) \wedge do(step\_on(c), 2)$. Whereas in figure 6.1 b) the only possible plan is $do(step\_on(b), 1) \wedge do(step\_on(c), 2)$, because the block a is not there (or because the agent cannot see it).

Notice that the agent may know about actions that initiate $infront(a)$ (such as, say, $put\_block\_in\_front(a)$). It could therefore schedule one of these actions before attempting the climbing, even if the agent is not physically capable of executing it. This is exactly what the bf iffPP will do without inhibition of abduction. With the inhibited version, on the other hand, we can express the fact that at that stage, the agent is just interested in testing whether $infront(A)$ actually holds. If the programmer decides that the agent must build the mountain to climb, then she/he will have to write for the "climber-builder" agent a program such as this:

[6] Up to factoring, the three nodes in example 6.3.2 are equivalent. In nodes 2 and 3 factoring cannot be applied again on the next abducible ($<$ and do respectively). Neither can one abduce those abducibles because the context is to "inhibit" abduction (the "t ::" at the head of the conjunction) as explained in the following section.



a)



b)

Figure 6.1: A World Block scenario for reactive planning

```
proc climb
  begin
    if infront(A) and currentlevel( C )
       and A is_higher_than C then
         begin
           step_on( A ) ; climb
         end
    else
       if available(A) and not infront(A) then
          put_block_in_front(A) ; climb
  end
```

In this second program, when the agent has not block in front (so that the first test fails) and there is some block available in the neighbourdhood, then the agent will indeed schedule (abduce) the action $put\_block\_in\_front(A)$ for execution (i.e. assuming that action is a primitive action).

It is worth noting that, with inhibited abduction, the agent can be seen as interleaving the "testing" of properties with the "planning" of actions. This testing is program-driven, i.e. the programs and the goals establish when the system will be testing and when will be planning. Moreover, notice that the "testing" is not restricted to the current state of the world. Previously planned actions can be used to establish that some property holds at a certain point in a plan. So, for instance, the climbing agent above will be able to deduce, if it has time to think about it, that after $do(a, 1)$, $infront(c)$ will hold.

6.3.4 How is the inhibition of abduction achieved?

Inhibition is achieved by maintaining an explicit identifier for the context of each unprocessed literal in the node. Every conjunct in UC and CN has a **context label** stating whether abduction is allowed or not. A structure $p :: C$ (where p is for **plan**ning) indicates that abduction is allowed on literals in C or obtained (by the inference rules) from C. A structure $t :: C$ (where t comes from **test**ing) indicates the abduction is inhibited and therefore the system must refrain from abducing any literal in C or derived from C by the inference rules (that is, these literals are just for "testing")[7].

Notice that implications in CN also require these context labels. This is because goals "activated" by the processing of the implications, may eventually generate abducibles.

Although the use of *context labels* may seem to be just a convenient procedural device, there is evidence that they are required to capture human-like reasoning capabilities. Gabbay, for instance, has based a comprehensive logical system (See [Gab93] and the literature in *labelled deductive systems*), which can be used to derived many well known logical systems (including nonmonotonic logics), on the manipulation of labels very similar to the *context labels* above.

It is worth noting, however, that "contexts" need not be restricted to "labels". Extra-logical information (as we explained in chapter 5) can improve the planner's effectiveness and can be made available as "contextual" information. The *pedigree of the goals*[8], which McCarthy suggested [McC95] would be important for an intelligent agent, could be part of the contexts. In particular, the elevator controller using only ACTILOG rules (as in chapter 5), could use "context labels" to indicate the overall purpose (implicit goal) of each ACTILOG unit (set of integrity contraints) and to permit resolution of conflicts between activated goals.

[7] The examples in the previous section only show structure $t :: C$. This means that abduction is not performed on literals in C.

[8] A record that permits to know where each goal comes from.

All these possibilities suggest that the use of context labels could be beneficial for the implementation of effective agents. In this thesis we restrict ourselves to the type of labels mentioned above. These labels establish a distinction between planning and testing.

We explain below, as part of the description of the algorithm, how the inference rules preserve contexts within a single derivation, except for the new rule that set them.

6.3.5 The Planning algorithms

Tables 6.1, 6.2 and 6.3 contain the new version of tables 3.2 and 3.5 (in chapter 3) with the adaptations for planning[9].

The first extension in both tables is the use of a new data structure for UC and CN, to accommodate context labels. So, UC and CN are now conjunctions of structures $Context :: C$. In UC, C is a conjunction of literals as before. In CN, C is an implication. The :: is, clearly, the operator that relates contexts to formulae. Among the procedures in those tables, the only one that changes its structure is *demo_abd*.

The new clauses are:

- **[DMAB-TES]** sets the context for an atom G whose predicate has been declared as *for_testing_only*. The clause simple creates a new structure $(test :: G)$ (or $(t :: G)$ for simplicity). Any structure obtained from this will copy its context. Observe that the planner can switch goals from the **plan**ning context to the **test**ing context, but not the other way around.

- **[DMAB-WHT]** This clause processes those abducibles that cannot be factored (because they were factored before) and cannot be abduced either (because their context is "testing"). We are being ambiguous about *what to do* with nodes containing this combination. In principle, any such node should be dropped because when factoring fails, this means that *the testing has failed* and this node does not contain a feasible plan. However, notice that factoring fails with those abducibles *currently* in Δ. As the content of Δ is constantly changing in an open architecture (by assimilation of inputs), this implies that factoring could succeed in the future and, therefore, the node should be preserved from further attempts with the factoring rules. On the other hand, keeping these nodes in the frontier introduces loops (factoring will be tried over and over again). The best one can do is to schedule those attempts at factoring after other nodes (alternative plans) have been explored. In GLORIA, this could be achieved by reordering the frontier and putting the failing node at the back of the frontier. This makes sense in terms of behaviour as well. If a test fails, one should not try it again immediately afterwards, especially when one has alternative courses of action.

In all the other rules of inference, the context of a node is simply copied into any node obtained from it.

The extensions described in this section account for the mechanism to inhibit abduction. However, as we said above, this is not the only adaptation required by **iff**PP to become an efficient planner. The following section describes the mechanism to deal with time-point orderings in such a way that the planner computes orderings *just when it is necessary*.

6.3.6 Dealing with time and time orderings

As we explained at the end of chapter 3, **inequalities** ($<$ atoms) are treated specially by the version of **iff**PP described in this thesis. This extends the original **iff** specification which only

[9] We use identical notation. Recall the use of $\wedge_i$, where i indicates the level of "nesting" of the $\wedge$ operator.

DEMO_abd′ : The abductive procedure for planning	
$demo\_abd(KB, InGoals, OutGoals, R)$	
$\leftarrow\ InGoals \equiv\ FirstNode \vee AltGoals$	
$\wedge_1\ FirstNode = (\Delta, (Cont :: (G \wedge Rest)) \wedge RUC, CN, HF, M)$	
$\wedge_1\ (\ (\ G = \neg G'$	
$\wedge_2\ NewCN = (Cont :: (\mathbf{false} \leftarrow G'), \{\}) \wedge CN$	
$\wedge_2\ NewNode = (\Delta, Rest, NewCN, HF, M)$	
$\wedge_2\ NextGoals = (NewNode \vee AltGoals)$	
$\wedge_2\ demo\_impl(KB, NextGoals, OutGoals, R-1)$	**[DMAB – NEG]**
$\vee_2\ (\ G \neq \neg G'$	
$\wedge_2\ (\ (\ unfoldable(G)$	
$\wedge_3\ definition(KB, G, D)$	
$\wedge_3\ NewNode = (\Delta, (Cont :: (D \wedge Rest))$	
$\wedge RUC, CN, HF, M)$	
$\wedge_3\ NextGoals \equiv NewNode \vee AltGoals$	
$\wedge_3\ useful\_ord(NextGoals, OrdGoals, R_{use})$	
$\wedge_3\ R_{use} < k_{use}$	
$\wedge_3\ NextGoals = OrdGoals\ )$	**[DMAB – UNF]**
$\vee_3$	
$(\ (\ equality(G) \vee_4 inequality(G)\ )$	
$\wedge_3\ \Delta' = G \wedge \Delta$	
$\wedge_3\ NewNode = (\Delta', (Cont :: Rest) \wedge RUC, CN, HF, M)$	
$\wedge_3\ NextGoals = (NewNode \vee AltGoals)\ )$	**[DMAB – EQU]**
$\vee_3$	
$(\ for\_testing\_only(G)\ \wedge_3\ Cont \neq test$	
$\wedge_3\ NewNode = (\Delta', (test :: G) \wedge$	
$(Cont :: Rest) \wedge RUC, CN, HF, M)$	
$\wedge_3\ NextGoals = (NewNode \vee AltGoals)\ )$	**[DMAB – TES]**
...	

Table 6.1: The abductive procedure adapted for reactive planning (Part 1)

DEMO_abd′ : The abductive procedure for planning	
...	
$\vee_3$	
($abducible(G)$	
$\wedge_3$ $factorable(\Delta, G, HF)$	
$\wedge_3$ $factoring(InGoals, NextGoals)$)	**[DMAB – FAC]**
$\vee_3$	
($abducible(G)$	
$\wedge_3$ $\neg factorable(\Delta, G, HF)$ $\wedge_3$ $Cont = plan$	
$\wedge_3$ $\Delta' = G \wedge \Delta$	
$\wedge_3$ $NewNode = (\Delta', (Cont :: Rest), CN, HF)$	
$\wedge_3$ $NextGoals = (NewNode \vee AltGoals)$))	**[DMAB – ABD]**
$\vee_3$	
($abducible(G)$	
$\wedge_3$ $\neg factorable(\Delta, G, HF)$ $\wedge_3$ $Cont = test$	
$\wedge_3$ $NewNode = (\Delta, (Cont :: Rest), CN, HF)$	
$\wedge_3$ $what\_to\_do(NewNode, AltGoals, NextGoals)$))	**[DMAB – WHT]**
$\wedge_2$ ($demo(KB, NextGoals, OutGoals, R-1)$)))	
$\vee_1$ ($\neg rule\_apply\_to\_uc(FirstNode)$	
$\wedge_2$ $demo\_impl(KB, InGoals, OutGoals, R)$)	**[DMAB – NRA]**

Table 6.2: The abductive procedure adapted for reactive planning (Part 2)

treats = specially. By, "specially" we mean that, although all these predicates are regarded as *abducibles* by the semantic framework, the proof procedure does not give them the same treatment that it does to other abducibles (such as *do*, in our representations).

Nevertheless, we still have to put knowledge about $<$ somewhere in the system. The transitivity axiom ($X < Y \leftarrow X < Z \wedge Z < Y$) and the anti-symmetry axiom (**false** $\leftarrow X < Y \wedge Y < X$) are essential for a planner. As $<$ has been declared an abducible, the first alternative is to add these axioms as integrity contraints. However, there is another alternative that allows a careful "fine-tunning" of what should be deduced about $<$, given what the agent knows and is " assuming" in its plans. The alternative is: *to write meta-rules to process partially instantiated inequalities.* Thus, the system will, on the one hand reason about the ordering of time-points whose value is unknown, and on the other, avoid commitments to orderings that are not explicitly required by the plans.

The solution is in the spirit of writing PRIORLOG and USELOG rules to focus the work of the prover. It is also similar to regarding the $<$ predicate as a "built-in" predicate that need no definition (as explored in [Wet97]).

6.3.6.1 Computing: $X < Y$ in Δ

The key element to resolve partially instantiated inequalities is a procedure to decide whether *it can be proved that* $X < Y$ (where X or Y or both are variables), using *what the agent already knows about* X and Y. What the agent knows about variables X and Y, appearing in $<$ atoms, can come only from other assumptions (abducibles) in the node. The program in figure 6.2 refers to Δ (the set of abducibles) to decide whether some inequality can be assumed to hold. This program can be explained as follows:

DEMO_ONE_IMPL : adding context	
$demo\_one\_impl(KB, ExQVars, \Delta, InImp, OutImps)$	
$\leftarrow$ ($noimp(InImps)$ $\wedge_1$ $empty(OutImps)$)	**[DMON – BAS1]**
$\vee_1$ ($\neg rule\_apply\_imp(InImp)$	**[DMON – BAS2]**
$\vee_1$ ($InImp \equiv (Cont :: (H \leftarrow (G \wedge Rest), HP))$	
$\wedge_1$ ((($equality(G)$ $\vee_3$ $inequality(G)$)	
$\wedge_2$ (($process\_equalities(ExQVars, InImp, NewImp)$	
$\wedge_3$ $demo\_one\_impl(ExQVars, \Delta, NewImp, OutImps)$)	
$\vee_3$ ($\neg process\_equalities(ExQVars, InImp, NewImp)$	
$\wedge_3$ $OutImps = (NewImp)$)))	**[DMON – EQU]**
$\vee_2$ ($G = \neg G'$	
$\wedge_2$ $H' = (H \vee G')$	
$\wedge_2$ $NewImp = (Cont :: (H' \leftarrow Rest, HP))$	
$\wedge_2$ $demo\_one\_impl(ExQVars, \Delta, NewImp, OutImps)$)	**[DMON – NEG]**
$\vee_2$ ($unfoldable(G)$	
$\wedge_2$ $definition(KB, G, D)$	
$\wedge_2$ $OutImps \equiv (Cont :: (H \leftarrow (D \wedge Rest), HP))$	**[DMON – UNF]**
$\vee_2$ ($abducible(G)$	
$\wedge_2$ $propagation(\Delta, InImp, OutImps)$))))	**[DMON – ABD]**
$rule\_apply\_imp(Imp)$	
$\leftarrow$ $Imp \equiv (Cont :: (H \leftarrow (G \wedge Rest), HP))$	
$\wedge_1$ ($equality(G)$ $\vee_2$ $inequality(G)$	
$\vee_2$ $unfoldable(G)$ $\vee_2$ $G = \neg G'$	
$\vee_2$ ($abducible(G)$ $\wedge_2$ $can\_propagate(\Delta, G, HP)$)	**[NRA – IMP]**

Table 6.3: Processing implications with contexts

$$
\begin{array}{lll}
before(X,Y,\Delta) & \leftarrow\ contains\_Var(X,\Delta) & \\
& \wedge\ contains\_Var(Y,\Delta) & \\
& \wedge\ \neg(X == Y) & \\
& \wedge\ rbefore(X,Y,\Delta) & \textbf{[BEFORE0]} \\
before(X,Y,\Delta) & \leftarrow\ ground(X)\ \wedge\ contains\_Var(Y,\Delta) & \\
& \wedge\ (Z\ lt\ W) \in \Delta\ \wedge\ ground(Z) & \\
& \wedge\ X < Z & \\
& \wedge\ (before(W,Y,\Delta) \vee W == Y) & \textbf{[BEFORE1]} \\
before(X,Y,\Delta) & \leftarrow\ ground(Y)\ \wedge\ contains\_Var(X,\Delta) & \\
& \wedge\ (Z\ lt\ W) \in \Delta\ \wedge\ ground(W) & \\
& \wedge\ W < Y & \\
& \wedge\ (before(X,Z,\Delta) \vee X == Z) & \textbf{[BEFORE2]} \\
 & & \\
rbefore(X,Y,\Delta) & \leftarrow\ (\ strictly\_in(X\ lt\ Y,\Delta) & \\
& \quad \vee\ strictly\_in(do(\_,X,Y),\Delta)\ ) & \textbf{[RBEF01]} \\
 & & \\
rbefore(X,Y,\Delta) & \leftarrow\ \neg(\ strictly\_in(X\ lt\ Y,\Delta) & \\
& \quad \vee\ strictly\_in(do(\_,X,Y),\Delta)\ ) & \\
& \wedge\ precedes(X,Z,\Delta), rbefore(Z,Y,\Delta) & \textbf{[RBEF02]} \\
 & & \\
precedes(X,Z,(A,\_)) & \leftarrow\ (A = (X1\ lt\ Z) & \\
& \quad \vee\ A = do(\_,X1,Z)\ ) & \\
& \wedge\ X == X1 \wedge\ \neg(X == Z) & \textbf{[PRECE01]} \\
 & & \\
precedes(X,Z,(A,Rest)) & \leftarrow\ \neg(A = (X1\ lt\ Z) & \\
& \quad \vee\ A = do(\_,X1,Z)\ ) & \\
& \wedge\ precedes(X,Z,Rest) & \textbf{[PRECE02]}
\end{array}
$$

Figure 6.2: The predicate **before**.

[BEFORE0] . . . **[BEFORE2]**. These clauses check that the terms involved in the inequality are either variables about which something is known (i.e. variables in Δ) or ground terms. X and Y cannot both be ground terms (that case is dealt with by the rewriting rules in chapter 3). The condition $\neg X == Y$ ensures that X and Y are not the same variable, as required by the anti-symmetry axiom for $<$. The predicate $==$ is the meta-level equivalent of $=$ and refers to syntactic equality (Two variables are the same if their identifier is the same).

[RBEF01] establishes that X is before (or equal to) Y whenever one of the following expressions is in Δ: X lt Y (recall that lt is the encoding of $<$ for the proof procedure) or $do(\_, X, Y)$. The relation *strictly_in* ascertains that its first argument is contained by its second argument. Observe that together with the usual understanding of the inequalities one can exploit the fact that in $do(\_, X, Y)$, X can only be less than Y.

[RBEF02] implements the transitivity rule for $<$ and $\leq$ (both at once for simplicity), by appealing to the predicate *precedes*.

[PRECE01] . . . **[PRECE02]**. Declaratively, the predicate defined by these clauses has the same reading as *rbefore*. One would write *rbefore* instead of *precedes* in the specification. However, we want to emphasize an important implementation detail: *strictly_in* can be used only to test whether $X < Y$ is in Δ. In *precedes*, there is a mechanism to generate a Z such that $X < Z$ is in Δ (the same applies to $do(\_, X, Z)$, of course).

6.3.6.2 Using $before(X, Y, \Delta)$

The program `before` is used by the planner to simulate the factoring rule and the propagation rule with partially instantiated inequalities, as explained below:

- **factoring of inequalities** is an auxiliary inference rule that should be added to *demo_abd*. Whenever the system finds a goal G which is a partially instantiated inequality $A < B$, the prover tests whether $before(B, A, \Delta)$ is the case for the node under analysis. If it is so, the node must be dropped because it contains a contradiction. If it is not so, Δ is updated with $A < B$.

  The fragment of program in fig 6.3.6.2 should replace [DMAB-EQU] in *demo_abd* (table 6.1).

- **propagation of inequalities** is another auxiliary procedure which must be attached to *demo_one_impl*. Whenever the system finds a partially instantiated inequality (ground atoms are dealt with by the rewrite rules) $A < B$ in the body of an implication I, it tests whether it is the case that $before(A, B, \Delta)$ for the node that contains the implication. It also tests whether $A < B$ has not been tried for propagation before. If both conditions hold, a new implication I', identical to I but without $A < B$, will replace I in that nodes' CN.

All the logic programs described in chapter 3 and in this chapter have been implemented in PROLOG. We have used this implementation to perform some experiments with the benchmark example: the elevator controller. Those experiments and the testbed for the elevator controller are the subjects of the rest of the chapter.

$$
\begin{array}{ll}
\ldots & \\
\vee_3 & \\
\quad (\ (\ equality(G) \vee_4 (inequality(G) \wedge_4 ground(G)) \ ) & \\
\quad \wedge_3 \ \Delta' = G \wedge \Delta & \\
\quad \wedge_3 \ NewNode = (\Delta', (Cont :: Rest) \wedge RUC, CN, HF, M) & \\
\quad \wedge_3 \ NextGoals = (NewNode \vee AltGoals) \ ) & [\mathbf{DMAB-EQ1}] \\
\vee_3 & \\
\quad (\ (\ inequality(G) \ \wedge_3 \neg ground(G)) & \\
\quad \wedge_3 \ G = (A < B) & \\
\quad \wedge_3 \ \neg before(B, A, \Delta) & \\
\quad \wedge_3 \ \Delta' = G \wedge \Delta & \\
\quad \wedge_3 \ NewNode = (\Delta', (Cont :: Rest) \wedge RUC, CN, HF, M) & \\
\quad \wedge_3 \ NextGoals = (NewNode \vee AltGoals) \ ) & [\mathbf{DMAB-IQ1}] \\
\vee_3 & \\
\quad (\ (\ inequality(G) \ \wedge_3 \neg ground(G)) & \\
\quad \wedge_3 \ G = (A < B) & \\
\quad \wedge_3 \ before(B, A, \Delta) & \\
\quad \wedge_3 \ NextGoals = AltGoals) \ ) & [\mathbf{DMAB-IQ2}] \\
\ldots &
\end{array}
$$

Figure 6.3: Factoring of inequalities

6.4 GLORIA implemented

6.4.1 The elevator testbed

The elevator testbed is a PROLOG process that interacts with users of a certain interface program, over a network. The PROLOG process and all the others could run on the same machine, but in general they are distributed over the Internet.

The elevator itself is simulated by the PROLOG process that runs GLORIA's *cycle*. The cycle predicate, in turn, uses the implementation of the proof procedure embodied by *demo* and *demo_impl*. These procedures operate on a simplified version of the OPENLOG code and ACTILOG rules described in chapters 4 and 5. This simplified version of the elevator program is shown in figure 6.4.

The program is figure 6.4 correspond to the OPENLOG program in figure 4.2 in page 101, chapter 4. For simplicity we have excluded the actions *open* and *close* that form part of the specification. The action $turnoff$ is enough is illustrate the interaction agent-environment.

This program is activated by means of the integrity constraint:

$$\forall N \ \forall T_1 \ \exists T_2 \ \exists T_f \ (T1 < T2 \wedge serve(N, T2, Tf) \leftarrow on(N, T1) \tag{6.1}$$

The testbed also includes PRIORLOG rules to guide the selection of goals for further processing. A compiled version (not shown) of the PRIORLOG rules in figure 5.3 (in chapter 5) was used, to program the elevator to behave according to policy 3, but giving a higher priority to serving floors whose buttons are found to be "on" when the elevator is passing by those floors.

For the user interface, we have used the World Wide Web standard platform. Each user interacts with a Web Browser which displays a WWW input-form, that acts as the elevator's calling panel. The panel is an array of buttons. By pressing one of these buttons, the user

```
serve(N, T1, T2) :-
  currentfloor(N, T1),
  do(turnoff(N), T1, T2).

serve(N, T1, T2) :-
  currentfloor(M, T1), M lt N, Nx is M + 1, do(up(Nx), T1, Tf),
  Tf lt T3,
  serve(N, T3, T2).

serve(N, T1, T2) :-
  currentfloor(M, T1), N lt M, Nx is M - 1, do(down(Nx), T1, Tf),
  Tf lt T3,
  serve(N, T3, T2).


currentfloor(M, T) :-
  at(M, Te), Te lt T, not move( Te, M,  T).

currentfloor(M, T) :-
  do(up(M), _, Te), Te lt T, not move( Te, M,  T).

currentfloor(M, T) :-
  do(down(M), _,  Te), Te lt T, not move( Te, M, T).


move( T1, M, T2 ) :-
  do(up(N),  _, T),  T1 le T, T lt T2,  not M eq N.

move( T1, M, T2 ) :-
  do(down(N),  _, T), T1 le T, T lt T2,  not M eq N.


X le Y :- X lt Y.
X le Y :- X eq Y.

abd(do).
abd(on).
abd(at).

for_testing_only(currentfloor(_,_)).
```

Figure 6.4: A compiled version of an OPENLOG program

orders the elevator to go to the floor indicated by the button. We make no attempt to simulate the user going or being inside the elevator. The "world" is represented by two files: *which*, containing a list of atoms $on(F)$, one for each floor on which the elevator's button is "on"; the other file is *where*, which contains a record, $at(F)$, of where the elevator currently is. The state of the world is presented to the user via the browser. On the browser's interface the user can see a diagram of the building and the current position of the elevator. The world is updated by the user (by pressing buttons) and by GLORIA, which disconnects the buttons and moves the elevator.

To guarantee proper access to the world's files, the PROLOG system and the Web platform have been extended with a library of semaphores for mutual exclusion. We gratefully acknowledge the use of the PiLLoW library [HC96] to create and manipulate the Web interface with PROLOG programs.

The elevator achieves its goal of serving some floor when it disconnects the "ON" button at that floor Then, the elevator switches the button to "off" (or "Served") as shown in the figures below.

The testbed is, we believe, a faithful simplification of an elevator that allows one to focus on the interaction between a planning agent and a constantly changing environment.

6.4.2 Practical considerations in GLORIA's implementation

"Let the world be its own model"[Bro91b], one of the ideas advanced by the reactive approach, summarizes some of the simplifications of the architecture that we made to implement the testbed.

GLORIA's specification states that the agent records its inputs at every cycle and time-stamps them with the "current time". However, to maintain a history of all its past inputs is unnecessary for the elevator controller. It only serves to overload the planner with unuseful information. As every button stays "ON" from the moment it is pressed until it is disconnected (by the elevator itself), the buttons themselves can "store" their state. One only needs to make sure that every input has a chance to trigger the corresponding conditions of the implications. Recall that one can specify the amount of resources the system will allocate to reason in each *cycle*. The parameter R of *demo* is used with that purpose. With an appropriate value for R, one can ensure that the agent will process all the integrity constraints, including those which allow the observations to activate new *serving* goals.

Notice that if one stores more than one *snapshot* of the environment (for instance, a sequence like $on(1,1), \ldots on(1,4)$ which records that the signal has been on at floor 1 between time-points 1 and 4, both inclusive), then one has to write a more complex specification of the activating integrity constraint (expression 6.1 above) to prevent a particular goal from being activated more than necessary.

Similar considerations apply to the records of failing and successful actions (also prescribed by the specification). When an action succeeds, one must record the value of the starting and finishing time of the action which will possibly instantiate arguments and constraints for subsequent actions in the same plan. This should be done for every plan containing the succeeding action. It is a sort of "factoring" between the particular instance of the action that succeeds and the action specification originally in the plan. For instance:

Example 6.4.1 If the plan that the executive receives is: $P = do(a1, T) \wedge RemActions$ and the agent succeeds with $do(a1, 0)$, then the plan will be changed to $do(a1, T) \wedge T = 0 \wedge RemActions$.

But when an action fails, the elevator controller drops the node (instead of simply recording the negation of the action and waiting for the proof procedure to verify that the node is

equivalent to **false**). This is not the logical thing to do, however, as what has failed is a particular attempt to perform that action (at a particular time) and the node should be kept in the frontier to allow new attempts for that action-type at other times.

We compensate for (correct) the *drop-the-failing-node* strategy, by restoring the frontier to the initial state (one node with the integrity constraint above) whenever it becomes empty because all the nodes have been dropped (which will eventually happen if the actions keep failing). This *clear-the-house* strategy is a sort of replanning strategy that seemed (in our experiments) to help the system to maintain the size of the frontier of goals at a minimum. This, however, requires further investigation.

This type of cleaning-procedure is required because the constant addition of input data and records of successful and failing actions is very inefficient without a mechanism to "forget" them or to prevent them from being considered again by the prover. One can keep extending the histories of propagation and factoring, but these cannot grow forever [10]

In the testbed, the system is "cleaned" (the inputs and all the nodes in the frontier are erased except the first) whenever the first node in the frontier is empty. A node is considered empty when there are no actions in Δ and no more subgoals to process in UC. Notice that this means that, once a goal is achieved (a full plan is completely executed), the agent forgets the alternatives plans to achieve the same goal.

This, however, is not enough. The frontier and the nodes' CN may grow too big before all the goals can be achieved and the first node is emptied. This is especially the case with the elevator, where a continuous flow of users can keep the system permanently activating new goals. One needs a more frequent garbage collection to delete from CN those implications (derived by propagation and case analysis) which are not required anymore. However, one cannot simply restore CN to the initial set of integrity constraints.

Our general (working) criterion is to delete those implications that are not integrity constraints and that contain no variable appearing in Δ or UC. These are the implications that are not restricting the values of any variable in any plan.

Another more informed criterion that we can envisage is to delete the "clipped" constraints (those derived from the "clipped" clauses [EC21] and [EC22], for instance.) which cannot be applied because the time-points involved are already in the past[11].

Further research is required to define the logical form of this operation which may well coincide with rules of inference in other proof procedures.

Another implementation detail is the fact that *the agent uses implications (integrity constraints) in which there is at least one abducible atom.* That is, the implications that really matter for the agent are those that connect inputs (modelled as abducible atoms) to goals that must be activated and reduced to plans. If the programmer writes (in ACTILOG) an integrity constraint that does not have abducibles in its body, the prover will reduce it to an implication with abducibles. This last implication will be kept and used to activate goals. This is important because it is the last implication that has to be restored to CN whenever the node is "cleaned".

There is a methodological lesson to be learnt from these details of implementation. When one programmes an agent like GLORIA, one chooses the integrity contraints that the agent must satisfy. Then, using the prover (*demo*), one derives implications such as the one just described, linking the inputs to the goals. For instance:

Example 6.4.2 One would write:

[10] This of course is a practical consideration. In logical terms they could be seen as growing forever. But the agent has a finite memory. There exist proof procedures (e.g. The connection graph proof procedure [Kow79b], chapter 8) that instead of adding to the (equivalent of our) histories deleted "links" to the clauses and atoms as they are used. More research is required to see if those techniques can be used to solve this problem.

[11] For instance, if the planner has **false** $\leftarrow$ $clipped(0, on(2), 3)$ and the *cycle*'s counter says that current time (T) is 4.

$$\exists T_2\ \exists T_f\ T < T_2 \wedge serve(N, T_2, T_f) \leftarrow\ holds(on(N), T) \tag{6.2}$$

and the system will reduce it to:

$$\begin{aligned} \exists T_2 \exists T_f\ & T < T_2 \wedge serve(N, T_2, T_f) \\ & \leftarrow\ do(A, T_1)\ \wedge\ T_1 < T \\ & \quad \wedge\ initiates(A, T, P)\ \wedge\ \neg clipped(T_1, P, T) \qquad [DO] \end{aligned}$$

$$\begin{aligned} \exists T_2 \exists T_f\ & T < T_2 \wedge serve(N, T_2, T_f) \\ & \leftarrow\ obs(P, T_1) \wedge\ T_1 < T \\ & \quad \wedge\ \neg clipped(T_1, P, T) \qquad [OBS] \end{aligned}$$

What the example illustrates is a form of "partial evaluation"[Hog90] in which the system stops compiling when it reaches an abducible. [DO] and [OBS] above are the implications that the system will maintain permanently to assimilate inputs and activate the *serve* goals. In addition, of course, the system must also maintain a permanent set of OPENLOG programs (such as the one in figure 6.4) to guide the unfolding of $serve(N, T_2, T_f)$ into the atomic actions that constitute the plans.

Thus, an agent like GLORIA is a compromise between partially evaluated and runtime-interpreted rules of behaviour.

Traces of the execution of the simulated elevator are presented, with comments, in the appendix A.5, page 172.

6.5 Conclusion

In this chapter we briefly reviewed the history of research in automatic planning. The history shows how the focus has moved from general purpose, disembodied planners, to reactive agents which achieve their functionality by interacting with their environment.

After the review, we presented the extensions to the logic programs in chapter 3 that transform the **iff**PP into a planner. This planner is an anytime algorithm that can be embedded in the reactive architecture and interleaved with the observing and execution mechanisms.

We then discussed the testbed that has been implemented to simulate an agent in a benchmark context: the elevator.

While presenting the planner, we also described the mechanism for inhibition of abduction, which, we showed, when used by the planner, can render OPENLOG programs and ACTILOG rules equivalent. This basically means that the two ways of programming an agent, with integrity contraints and with definitions, are equivalent in the sense that they generate the same behaviour in the agent. It also means that implications can be embedded in the definition to yield a more general language for logic programming.

Thus, the agent with a planning mechanism as describe in this pages, can move smoothly from being a "purely reactive" agent, that simple checks the environment to decide what to do immediately after, to a more "deliberate" agent, capable of reasoning about its non-immediate future. We still have to answer important questions such as whether the system will be able to guide the behaviour of agents engaged in communications and cooperative problem solving. Also, alternative strategies to store and manipulate goals and inputs should be explored. In theory, however, the logic programs presented in this and the previous chapters, already provide a systematic way of relating reactivity with rational behaviour within an intelligent agent.

Chapter 7

Conclusions

In this thesis we designed a language to describe agents. The language is essentially a logic programming language supplemented with semantics abstractions such as processes, events, fluents, goals, plans and beliefs.

In the introductory chapter 1 we analysed the basic concepts in theories of agents and explained the crucial role that reactivity plays in realistic accounts of agency. We also explained how a logical language expressive enough to be its own meta-language, could be used to formalize another critical notion in agency, namely *resource-bounded reasoning*.

In chapter 2 we showed how a logic program could be used to formalize (and implement) a process. We also showed how a process so formalized, could model the *interleaving* of sensing, (bounded) reasoning and acting, essential for an agent that is both *reactive* and *rational*.

In chapter 3 we presented the specification and implementation of the abductive proof procedure **iff**PP. The logic program implementing the proof procedure constitutes an *any-time* algorithm. The proof procedure can, therefore, be used as the reasoning mechanism of an agent with bounded resources for computation.

In chapter 4 we presented the logic programming language OPENLOG by which procedural knowledge can be embedded into an agent. We based the language on a theory of actions that support the descriptions of dynamic universes with changing fluents, event concurrency and synergistic effects.

In chapter 5 we presented three more logic programming languages ACTILOG, PRIORLOG and USELOG. The first can be used to write instructions for "activation of goals" and integrity constraints for the agent. The second and the third languages allow a programmer to embed heuristics about the priority and utility of goals and plans. These heuristics could guide the system to be more efficient in the reduction of goals to actions to be executed by the agent. Noticeably, the languages presented in these last two chapters are "syntactic sugar" for subsets of first order logic.

In chapter 6, we adapted the proof procedure described in chapter 3 to use it as the planner engine of the agent. The adaptation contemplates a set of techniques to support an efficiency implementation and to allow the agent to behave as a reactive agent. We also referred to some experiments with an implementation of the agent to simulate an elevator controller. The experiments illustrate the openness, reactivity and goal-oriented behaviour that this agent can achieve.

It is enticing to see developments in areas of AI (such as robotics) coinciding with developments in logic oriented towards more practical and expressive languages. It seems as if logic does not have to be synonymous with static, over-simplified or impractical descriptions of the world.

Bibliography

[AC90] P.E. Agre and D. Chapman. What are plans for? In Pattie Maes, editor, *Designing Autonomous agents: theory and practice from biology and engineering and back*, pages 17–34. Elsevier Science Publishers B.V., Amsterdam, Netherlands, first mit press edition, 1990.

[AF94] James Allen and George Ferguson. Actions and events in interval temporal logic. *J. of Logic and Computation*, 4(5), 1994.

[AH87] J.F. Allen and P. Hayes. Moments and points in an interval-based temporal logic. Tr 180, Departments of Computer Science and Philosophy. The University of Rochester, Deparment of AI, 1987.

[AIS88] J. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *AAAI 1988 Proceedings*, 1988.

[AK90] James Allen and Johannes Koomen. Planning using a temporal world model. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 559–565. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. Originally in Proceedings of IJCAI-83.

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–842, 1983.

[All84] James Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[All87] Newell Allen. *Unified Theories of Cognition.* Harvard University Press, 5 edition, 1987.

[All91] James F. Allen. Temporal reasoning and planning. In J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning About Plans.* Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991. ISBN 1-55860-137-6.

[Bak91] Andrew .B. Baker. Nonmonotonic reasoning in the framework of the situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[BK82] K. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S-A. Tarnlünd, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[BK96] Anthony Bonner and Michael Kifer. Concurrency and communication in transaction logic. In Dino Pedreschi and Carlo Zaniolo, editors, *International Workshop on Logic In Databases.* Area di Ricerca di Pisa del CNR, S. Miniato, Pisa. Italy, July 1996.

[Bra87] Michael Bratman. *Intention, Plans and Practical Reasoning.* Harvard University Press, Cambridge, Massachusetts and London, England, 1987.

[Bro85] Lee Brownston. *Programming expert systems in OPS5.* Addison-Wesley Inc., USA, 1985.

[Bro86] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23, 1986.

[Bro91a] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, pages 139–159, 1991.

[Bro91b] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the 12th Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991. IJCAI Inc.

[CDT91] L. Console, T.. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 2(5):661–690, 1991.

[Cha90] David Chapman. Planning for conjunctive goals. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 537–558. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. (First appeared in 1987).

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minder, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[CM85] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Menlo Park, CA, 1985.

[CT91] K. Currie and A. Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[Dav67] Donald Davidson. The logical form of action sentences. In Nicolas Rescher, editor, *Logic of Decision and Action.* University of Pittsburgh Press, 1967.

[Dav80] Donald. Davidson. *Essays on Actions and Events.* Clarendon Press, Oxford, England, 1980.

[DB88] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *AAAI 88: The Seventh National Conference on AI*, volume 1, Saint Paul, Minnesota, August 1988.

[DDS92] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for normal abductive programs. *Proc. International Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[DDS95] M. Denecker and D. De Schreye. Sldnfa: an abductive procedure for abductive logic programs. 1995.

[Den87] Daniel Denett. *The Intentional Stance.* The MIT Press, Cambridge, MA, 1987.

[dK86] J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 32, 1986.

[DMB92] Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with the abductive event calculus. In *Proc. European Conference on Artificial Intelligence*, 1992.

[DP87] Rina Dechter and Judea Pearl. The optimality of a^*. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 167–198. Springer-Verlag, 1987.

[DQ94] Jacinto A. Dávila Quintero. Knowledge assimilation in multi-agents system. Master's thesis, Imperial College, London, September 1994.

[DQ96] Jacinto A. Dávila Quintero. A logic-based agent. Technical report, Imperial College, London, February 1996.

[EK88] K. Eshghi and R. Kowalski. Abduction through deduction. Technical report, Deparment of Computing. Imperial College, London, UK, 1988.

[EK89] K. Eshghi and R. Kowalski. Abduction compare with negation as failure. In G. Levi and M. Martelli, editors, *Proceedings of the International Conference on Logic Programming*, pages 234–255, Lisbon, Portugal, 1989. MIT Press.

[Esh88a] K. Eshghi. Abductive planning with the event calculus. In R. A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming*, Seattle, USA, 1988.

[Esh88b] Kave Eshghi. Abductive planning with event calculus. In *Proceedings 5th International Conference on Logic Programming*, 1988. pg. 562.

[Eva89] C.A. Evans. Negation as failure as an approach to the hanks and mcdermott problem. In F.J. Cantu-Ortiz, editor, *Proc. 2nd. International Symposium on Artificial Intelligence*, Monterrey, México, 1989. McGraw-Hill.

[Fit85] Melvin R. Fitting. A kripke-kleene semantics for logic programs. *The Journal of Logic Programming*, 2:295–312, 1985.

[FK96] T Fung and R Kowalski. The iff proof procedure for abductive logic programming. July 1996. to appear.

[Flo67] R.W. Floyd. Assigning meanings to programs. In J.T Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[FN71] R.E Fikes and N.J. Nilsson. Strips:a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fun96] Tze Ho Fung. *Abduction by deduction.* PhD thesis, Imperial College, London, January 1996.

[Gab93] Dov Gabbay. What's a logical system?. In Dov Gabbay, C.J. Hogger, and J.A Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1. Oxford University Press Inc., 1993.

[Gal91] Antony Galton. Reified temporal theories and how to unreify them. 1991.

[Gal95] Antony Galton. Time and change in ai. In Dov Gabbay, C.J. Hogger, and J.A Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming (Epistemic and Temporal Reasoning)*, volume 4, pages 175–240. Oxford University Press Inc., New York, 1 edition, 1995.

[Gin89] Matthew L. Ginsberg. Universal planning: An (almost) universally bad idea. *AI MAGAZINE*, pages 40–44, Winter 1989.

[GL90] Michael Georgeff and Amy Lansky. Reactive reasoning and planning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 729–734. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[GLR90] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus. In Dov Gabbay, editor, *Essays for Bledsoe*. 1990.

[GN88] Michael R. Genesereth and Nils Nilsson. *Logical foundations of Artificial Intelligence*. Morgan Kauffman Pub., California. USA, 1988.

[Gor88] Michael J.C. Gordon. *Programming Language Theory and its implementation*. Prentice Hall, Englewood Cliffs, NJ 07632, 1988.

[Gre69] C. Green. Application of theorem proving to problem solving. In *Proc. IJCAI-69*, pages 219–239, Washington D.C., 1969.

[Hay85] P.J. Hayes. The second naïve manifesto. In J.R. Hobbs and R.C. Moore, editors, *Formal Theories of the Commonsense World*, pages 71–107. Ablex, 1985.

[HC96] Manuel Hermenegildo and Daniel Cabeza. Internet and www programming using computational logic systems. http://www.clip.dia.fi.upm.es/, 1996.

[Hew91] Carl Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47:79–106, 1991.

[HM87] S Hanks and D McDermott. Nonmonotonic logics and temporal projection. *Artificial Intelligence*, 33(3):379–412, November 1987.

[HMP92] Zhisheng Huang, Michael Masuch, and L. Pólos. Alx, an action logic for agents with bounded rationality. Ccsom report 92-70, University of Amsterdam (PSCW), 1992.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEE Trans. System Science and Cybernatics*, SSC-4(2):100–107, 1968. A Start.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12:576–583, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hog90] Christopher John Hogger. *Essentials of Logic Programming*. Claredon Press, Oxford, 1990.

[Isr93] David Israel. The role(s) of logic in artificial intelligence. In Dov Gabbay, C.J. Hogger, and J.A Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, pages 1–30. Oxford University Press Inc., New York, 1993.

[Jon75] Lyn Jones. *Systems Modelling: Decision Analysis*. The Open University, Walton Wall, Milton Keynes. UK, first edition, 1975.

[Kae87] L. P. Kaelbling. Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings AIAA Conference on Computers in Aerospace*, Wakefield, MA, 1987.

[Kae90] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. *Planning*, 1990.

[Kar94] G. Neelakantan Kartha. Two counterexamples related to baker's approach to the frame problem. *Artificial Intelligence*, 69:379–391, 1994.

[KKT93] A.C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kle38] S.C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[Kle52] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, 1952.

[KM90] A.C. Kakas and P Mancarella. Abductive logic programming. In W. Marek, A. Nerode, D. Pedreschi, and V.S. Subrahmanian, editors, *Proc. NACLP Workshop on Non-monotonic Reasoning and Logic Programming*, Austin, Texas, 1990.

[Kow79a] Robert A. Kowalski. Algorithm = logic + control. *Comm. of the ACM*, 22:424–431, 1979.

[Kow79b] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North Holland, New York, 1979.

[Kow84] Robert Kowalski. The relation between logic programming and logic specification. *Phil. Trans. R. Soc. London*, A(312):345–361, 1984.

[Kow94] Robert Kowalski. Logic without model theory. In Dov Gabbay, editor, *What is a logical system?*, chapter 2, pages 35–71. 1994. (Also at http://www-lp.doc.ic.ac.uk/⊥p/Kowalski/models.ps).

[Kow95] Robert Kowalski. Using metalogic to reconcile reactive with rational agents. In K. Apt and F. Turini, editors, *Meta-Logics and Logic Programming*. MIT Press, 1995. (Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/recon-abst.html).

[KR90] L.P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In Pattie Maes, editor, *Designing Autonomous agents: theory and practice from biology and engineering and back*. Elsevier Science Publishers B.V., Amsterdam, Netherlands, first mit press edition, 1990.

[KS86] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KS94] Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In M. Bruynooghe, editor, *Proc. International Logic Programming Symposium*, pages 539–553. MIT Press, 1994. (Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/fs/ilps94.html).

[KS97] Robert Kowalski and Fariba Sadri. Towards a unified agent architecture that combines rationality with reactivity. 1997. To appear. (Also at http://www-lp.doc.ic.ac.uk/UserPages/staff/fs/unify.html).

[Kun87] Kenneth Kunen. Negation in logic programming. *The Journal of Logic Programming*, 4(4):289–308, December 1987.

[Lan87] Amy L. Lansky. A representation of parallel activity based on events, structure, and causality. 1987.

[Lif90a] Vladimir Lifschitz, editor. *Papers by John McCarthy*, chapter Ascribing Mental Qualities to Machines, pages 93–118. Ablex Publishing Corp., Norwood, New Jersey, 1990. First printed in Philosophical Perspectives in Artificial Intelligence. 1979. M. Ringle (Ed.) pp. 161-195.

[Lif90b] Vladimir Lifschitz, editor. *Papers by John McCarthy*, chapter Mathematical Logic in Artificial Intelligence, pages 237–252. Ablex Publishing Corp., Norwood, New Jersey, 1990. First printed in Daedalus, Winter 1988, pp 297-311.

[Lif91] Vladimir Lifschitz. Toward a metatheory of action. In *Proceedings Knowledge Representation Conference*, pages 376–386, 1991.

[Lin93] Andrew R. Lingard. *Towards the Efficient Generation of Plans Containing Overlapping Actions*. PhD thesis, Imperial College, London, October 1993.

[LO83] A. Lansky and S. Owicki. Gem: A tool for concurrency, specification and verification. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 198–212, August 1983.

[LRL+95] H. Levesque, R. Reiter, Y. Lespérance, L. Fangzhen, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *(forthcomming)*, 1995. (Also at http://www.cs.toronto.edu/~ cogrobo/).

[Mae91] P Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. The MIT Press, Cambridge, MA, 1991.

[MBD95] Lode Missiaen, Maurice Bruynooghe, and Marc Denecker. Chica, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5):579–602, October 1995.

[McC86] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26:89–116, 1986.

[McC91] F.G. McCabe. *Logic and objects*. Prentice Hall, UK, 1991.

[McC95] John. McCarthy. Making robots conscious of their mental states. *Machine Intelligence*, 15, 1995. Also at: http://www-formal.stanford.edu/jmc/consciousness.html.

[McD82] Drew McDermott. A temporal logic for reasoning about processes and plans. 1982.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Mic95] Sun Microsystems. Hotjava home page. http://webrunner.neato.org/, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil96] Rob Miller. Notes on deductive and abductive planning in the event calculus. http://www-lp.doc.ic.ac.uk/UserPages/staff/rsm/rsm1.html, July 1996.

[MM96] S. Muggleton and D. Michie. Machine intelligibility and the duality principle. *British Telecom Technology Journal*, 14(4):15–23, 1996.

[Moo95] Moore. *Logic and Representation.* Center for the Study of Language and Information (CSLI), 333 Ravenswood Avenue, Menlo Park, CA 94025, 1995.

[Mos81] Christopher D.S. Moss. *The Formal Description of Programming Languages using Predicate Logic.* PhD thesis, Imperial College, London, July 1981.

[Mos92] Peter Mosses. *Action Semantics.* Cambridge University Press, Cambridge, 1992. P. Mosses is at Aarhus University, Denmark.

[MSae90] K. A. Mohyeldin Said and al et, editors. *Modelling the Mind.* Clarendon Press, Oxford, 1990.

[NSS60] A. Newell, J.C. Shaw, and H.A. Simon. Report on a general problem solving program. In *Proceedings International Conference on Information Processing*, pages 256–264, Paris, 1960. UNESCO.

[Ped87] E.P.D Pednault. Formulating multiagent, dynamic-world problems in the classical planning framework. In M.P. Georgeff and A.L. Lansky, editors, *Reasoning about actions and plans: proceedings of the 1986 workshop*, pages 47–82, Los Altos, California, 1987. Morgan Kaufmann.

[Pei55] C.S. Peirce. *Philosophical Writings of Pierce.* Dover Publications, New York, 1955.

[Pel91] Richard N. Pelavin. Planning with simultaneous actions and external events. In J. F. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning About Plans.* Morgan Kauffmann Publishers, Inc., San Mateo, California, 1991. ISBN 1-55860-137-6.

[PIB87] M. Pollack, D. Israel, and M. Bratman. Toward an architecture for resource-bounded agents. *CSLI*, pages 1–19, 1987.

[Pin94] Javier Andrés Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Toronto, 1994.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems. a survey of current trends. In J.W. deBakker, W. de Roever, and G. Rozenberg, editors, *Current trends in Concurrency, Lecture Notes in Computer Science*, volume 224, pages 510–584. Springer-Verlag, Berlin, 1986.

[Poo89] D. Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence Journal*, 5:97–110, 1989.

[Poo95] David Poole. Logic programming for robot control. In Chris S. Mellish, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 150–157, San Mateo, California, 1995. Morgan Kaufmann Publishers, Inc.

[PW80] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[Rai70] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices under Uncertainty.* Addison-Wesley, Reading, Massachussetts, July 1970.

[Rei96] Raymond Reiter. A formal account of planning with concurrency, continous time and natural actions. In Ute Sigmund and Michael Thielscher, editors, *Reasoning About Actions and Planning in Complex Environments*, Alexanderstrasse 10, D-64283 Darmstadt, Germany, 1996. Technische Hochschule Darmstadt. (Also at http://www.cs.toronto.edu/~ cogrobo/).

[Res66] Nicholas Rescher, editor. *The Logic of Decision and Action*. University of Pittsburgh Press, 1966.

[RG95] Anand Rao and Michael Georgeff. Formal models and decision procedures for multi-agent systems. Technical note 61, Australian Artificial Intelligence Institute, June 1995.

[RK95] Stanley J. Rosenschein and Leslie Pack Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73:149–173, February 1995.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs - New Jersey, 1995.

[Rob79] J.A. Robinson. *Logic: Form and Function*. Edinburgh University Press, Edinburgh, Scotland, 1979.

[Ros89] Stanley Rosenschein. Synthesizing information-tracking automata from environment descriptions. In R. Brachman, H Levesque, and R Reiter, editors, *Proceedings of the First International Conference on Principles of Representation and Reasoning*. Morgan Kaufmann Publishers, Inc, 2929 Campus Drive. San Mateo, CA 94403, 1989.

[RW91] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

[Sac74] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sac75] E.D. Sacerdoti. A structure for plans and behaviour. Technical note 09, SRI, Menlo Park, CA, 1975. (Also in: American Elsevier, New York 1977).

[San93] Erik Sandewall. The range of applicability of nonmonotonic logics for the inertia problem. In R. Bajcsy, editor, *Proc. of the IJCAI*, pages 738–743, 1993.

[San94] Erik Sandewall. The range of applicability of some non-monotonic logics for strict inertia. *J. of Logic and Computation*, 4(5):581–615, 1994.

[Sch94] Lenhart K. Schubert. Explanation closure, action closure and the sandewall test suite for reasoning about change. *J. of Logic and Computation*, 4(5):679–700, 1994.

[Ser83] Marek Sergot. A query-the-user facility for logic programming. In Degandp and Sandwell, editors, *Integrated interactive computer systems*, pages 27–41. North Holland Press, 1983.

[Sha89] Murray Shanahan. Prediction is deduction but explanation is abduction. In N.S. Sridharan, editor, *Proc. International Joint Conference on Artificial Intelligence*, pages 1055–1060. Morgan Kaufmann, Detroit. Mi, 1989.

[Sha93] Murray Shanahan. Explanation in the situation calculus. In *Proc. International Joint Conference on Artificial Intelligence*, pages 160–165. Morgan Kaufmann, 1993.

[Sha96] Murray Shanahan. Robotics and the common sense informatic situation. *Working Notes of Common Sense 96, The Third Symposium on Logical Formalizations of Commonsense*, pages 186–198, 1996. Also in Proceedings ECAI 96 and at http://www.dcs.qmw.ac.uk/~ mps/pubs.html.

[Sha97] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[Sho89] Yoav Shoham. *Reasoning About Change.* The MIT Press, Cambridge, Massachusetts - London, England, 1989.

[Sho90] Yoav Shoham. Agent0: A simple agent language and its interpreter. 1990.

[Sho95] Yoav Shoham. Agent oriented programming. *Artificial Intelligence*, 1995.

[Sim55] Herbert A. Simon. A behavioral model of rational choice. *Quarterly Journal of Economics*, pages 99–118, 1955.

[SMM96] Kenji Sasaki, Sandor Markon, and Masami Makagawa. Elevator group supervisory control system using neural networks. *ELEVATOR WORLD*, XLIV(2):81–90, February 1996. http://www.enews.com/magazines/elevator/archive.

[Sri91] Suryanarayana Murthy Sripada. *Temporal Reasoning in Deductive Databases.* PhD thesis, Imperial College, London, January 1991.

[Ste81] M. Stefik. Planning with constraints (molgen: Part 1). *Artificial Intelligence*, 16:111–140, 1981.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts and London, England, 1977.

[Tan87] Andrew S. Tanenbaum. *Operating systems : design and implementation.* Prentice-Hall International, London, 1987.

[Tat76] A. Tate. Project planning using hierarchical nonlinear planner. Research report 25, Edinburgh University, Deparment of AI, 1976.

[Tat77] A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893, Cambridge, MA, 1977.

[Ton95] Francesca Toni. *Abductive Logic Programming.* PhD thesis, Imperial College, London, July 1995.

[Tur50] Alan M. Turing. Computing machinery and intelligence. *Mind 59*, pages 433–460, Octuber 1950. also in [?].

[Ver83] S.A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:246–267, 1983.

[vW63] G.H. von Wright. *Norm and Action.* Routledge and Kegan Paul,, London, 1963.

[War74] D.H.D. Warren. Warplan: a system for generating plans. Logic memo 76, Deparment of Computational Logic, University of Edinburgh, Edinburgh, Scotland, 1974.

[War76] D.H.D. Warren. Generating conditional plans and programs. In *Proceedings of the AISB Summer Conference*, pages 344–354, 1976.

[Wet97] Gerhard Wetzel. *Abductive and Constraint Logic Programming.* PhD thesis, Imperial College, London, March 1997.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, Cambridge, Massachusetts - London, England, 1993. Foundations of Computing.

[WJ84] Niklaus Wirth and Kathy Jensen. *PASCAL:User Manual and Report.* Springer-Verlag, 3rd. edition, 1984.

[WJ95] Michael Wooldridge and Nicholas Jennings. *Intelligent Agents: Theory and Practice.* Knowledge Engineering Review, 1995.

[WKT95] Gerhard Wetzel, Robert Kowalski, and Franchesca Toni. A theorem-proving approach to clp. In A. Krall and U. Geske, editors, *Workshop Logische Programmierung*, number 270 in -, pages 63–72. GMD-Studien, September 1995.

Appendix A

Appendix

A.1 Proof of proposition about memory required by SC

In SC the chronological order of actions and situations is conditioned by the syntax of the situational terms. Information about the relative order of two actions must always be explicit and decisions must be made as early as possible. For instance, if one knows that action a occurred first, and then b and c occurred in some unknown order, this would be represented in SC as:

$do(c, do(b, do(a, s_o))) \vee do(b, do(c, do(a, s_o)))$

not as:

$do\_or(a, b, do(a, s_o))$

which would be a more "economical" representation, but which would also require the introduction of the logical functor *do_or*. This type of patching of the representation would eventually lead to a "reified" version of AEC built into SC.

For the implementors, the explicit representation in SC means that instead of something like[1]:

$[\{b, c\}, a]$

they would have to have something like:

$\{[c, b, a], [b, c, a]\}$

It also means that the history of actions will always be stored in a "flat" or-list of and-lists, if one wants the agent to have all the possible plans available for comparison.

But more important, it means the the memory space required to store plans (sequences of actions and their orderings) in SC is a function 1) of the number of bits required to distinguish among Z action types, 2) of N, the number of action occurrences, and 3) of K, *the number of actions whose absolute ordering is known* (for instance, if as above, a, b, c occurred and it is known that a is before b and a is also before c, one knows the absolute ordering of a because one knows its position with respect to all the other actions).

For N occurrences there are $N!$ possible arrangements P_a or permutations, ($P_a = N!$) when no order is considered. If one knows the absolute ordering of one action, P_a reduces by 1 ($P_a = (N-1)!$. In general, $P_a = (N-K)!$. So, in the example above, $P_a = (3-1)! = 2$.

Considering that to distinguish between Z action-types one needs $aprox(log_2(Z))$ bits[2], then the total space required to store a plan with N actions, K of which have their ordering

[1] where {} indicates an or-list and [] an and-list.

[2] $aprox(X)$ is a function that returns the nearest greater integer with respect to X. For instance, if $X = 1.3$, $aprox(X) = 2$.

completely defined, is given (in bits) by:

$$SCStorage \quad = \quad (N * (N-K)! + (N-K)! - 1) * aprox(log_2(Z+1))$$

where $Z+1$ is due to the fact that one has to distinguish "$\vee$" symbols separating the lists (i.e. the number of actions (N) times the number of possible arrangements (P_a) plus the number of "$\vee$" symbols ($P_a - 1$)).

One can see that with full information $K = N$ and so:

$$SCStorage_{fullinfo} \quad = \quad N * aprox(log_2(Z+1))$$

which yield $SCStorage_{fullinfo} = 3 * aprox(log_2(Z+1))$, as one would expect in the example above.

On the other hand, with no information about ordering $K = 0$, the function is:

$$SCStorage_{noinfo} \quad = \quad (N * N! + N! - 1) * aprox(log_2(Z+1))$$

yielding $SCStorage_{noinfo} = 23 * aprox(log_2(Z+1))$ in the example, which is indeed the minimal amount required to store the list: $a,b,c \ - \ a,c,b \ - \ b,a,c \ - \ b,c,a \ - \ c,a,b \ - \ c,b,a$

□

A.2 Proof of proposition about memory required by EC

Unlike SC, EC does not require explicit representation of all permutations of the actions in a plan. If N actions/events occur, then one has an store with N records, each of these with an identifier of the action type and of the time-point(s) of occurrence. The store must also contain records of the ordering on times points. So, in the case of full information, EC will easily consume more space that SC, as this example shows:

Example A.2.1 Three actions, a, b and c, occur in this order. In SC, this is straightforwardly represented as $[a,b,c]$. In EC, the minimal representation would be something like: $[(a,t_1),(b,t_2),(c,t_3)]$ plus $[t_1 < t_2, t_2 < t_3]$.

As one can see, there are many more "symbols" (and symbol occurrences) involved in the second case than in the first. This situation, however, tends to reverse when information is scarce. To prove this claim, we will try to device an storage requirement function, $ECStorage$, for EC, similar to $SCStorage$ in the previous section.

With EC, however, we have an inconvenience we did not have in SC. The number of bits required to distinguish among the "symbols" in the representation is, strictly speaking, *variable*. It is not $aprox(log_2(Z+1))$ anymore because we now also have the time-points (the t_i's in the example) to be distinguished. One should notice that the number of *time-point identifiers* is $2*N$, twice the number of action occurrences, when one wants to represent every action's start and finish time with a different symbol (e.g. in the example above: $[(a,t_1,t_2),(b,t_3,t_4),(c,t_5,t_6)]$). But, if one simply wants to match the expressiveness of SC, N would be a sufficient upper bound[3]. That is, one time-point per occurrence.

[3] N is an "upper bound" because one may actually need less than that if the "occurrences" share time-points like, for instance: $[(a,t_1),(b,t_1),(c,t_2)]$. But this requires some additional information about actions' orderings.

So, in EC one has at least Z action types plus N time-point symbols among which one needs to distinguish (one does not need "$\vee$" identifiers as in the previous section). That is, we need a *data unit*[4], with, at least, $aprox(log_2(Z+N))$ bits.

Things are more complicated, though. To maintain a data unit with variable length is difficult in practice, specially if the data unit is being used to store different types of data, as is the case with time-points and action types in this discussion. To avoid that sort of complications, we penalize EC (we discuss the penalisation below) by assuming that we will use different data units for different type of data. The action-type data unit will require $aprox(log_2(Z))$ bits. And a time point will be kept in cell of $aprox(log_2(N))$ bits.

Thus, to record N action occurrences, we will use N times the size of the action-types data unit plus the size of the time-point data unit[5]. In addition to this, we will need the list with the pairs indicating the order between time-points. These considerations lead to:

$$\begin{aligned} ECStorage &= N*(aprox(log_2(Z)+aprox(log_2(N))) \\ &\quad +M*(aprox(log_2(N))) \end{aligned}$$

where M is the number of entries indicating ordering between pairs of time-points. For instance, with $[(a,t_1),(b,t_2),(c,t_3)]$, we will need a list such as $[(t_1<t_2),(t_1<t_3),(t_2<t_3)]$ which could obviously be reduced, using the transitive law on $<$ to: $[(t_1<t_2),(t_2<t_3)]$. Normally then, $M \leq N-1$. But if we do not consider the transitivity law[6], $M \leq \frac{N*(N-1)}{2}$.

Thus, with full information about the ordering of the actions and without using the transitivity law (so that, $M=\frac{N*(N-1)}{2}$), the storage requirements of EC are given by:

$$\begin{aligned} ECStorage_{fullinfo} &= N*(aprox(log_2(Z)+aprox(log_2(N))) \\ &\quad +N*\frac{N-1}{2}*(aprox(log_2(N))) \end{aligned}$$

And with no information about action ordering ($M=0$):

$$ECStorage_{noinfo} = N*(aprox(log_2(Z)+aprox(log_2(N)))$$

□

A.3 Proof of proposition comparing EC and SC

In the two previous sections, we obtained the basic results to prove this proposition. They are:

$$SCStorage = ((N+1)*(N-K)!-1)*aprox(log_2(Z+1))$$

and

$$\begin{aligned} ECStorage &= N*(aprox(log_2(Z)+aprox(log_2(N))) \\ &\quad +M*(aprox(log_2(N))) \end{aligned}$$

[4] A **data unit** is a cell to store a unit of information such as a symbol or a number

[5] Observe that this is a penalisation of EC, because in the useful cases (with $Z>2$ and $N>2$) $log_2(Z+N) < log_2(Z)+log_2(N)$. Actually, one can define $aprox$ so that: $aprox(log_2(Z+N)) \leq aprox(log_2(Z))+aprox(log_2(N))$.

[6] Recall that $\binom{N}{2}=\frac{N*(N-1)}{2}$.

In each equation, however, we have a different unit to quantify the lack of information. Recall that K is the number of action occurrences whose absolute position (chronological position with respect to all the other occurrences) is known, whereas M is the number of atoms ($t_i < t_j$), of the relation $<$, that are known at a particular moment. Fortunately, we can express M in terms of K (and N) by noticing that:

$$M = K * N - \sum_{i=1}^{K} i$$

Thus, $ECStorage$ can now be re-stated as:

$$\begin{aligned} ECStorage &= N * (aprox(log_2(Z) + aprox(log_2(N))) \\ &\quad +(K * N - \sum_{i=1}^{K} i) * (aprox(log_2(N))) \end{aligned}$$

To prove the proposition, it suffices to show that $SCStorage > ECStorage$ for $K \rightarrow 0$, or equivalently $SCStorage - ECStorage > 0$ (for $K \rightarrow 0$). The difference is given (writing $aprox(log_2(X))$ as $aproxlog(X)$ for simplicity) by:

$$\begin{aligned} SCStorage - ECStorage &= (N+1) * ((N-K)!) * aproxlog(Z+1) \\ &\quad +(\sum_{i=1}^{K} i) * aproxlog(N) \\ &\quad -aproxlog(Z+1) - N * aproxlog(Z) \\ &\quad -(K+1) * (N) aproxlog(N) \end{aligned}$$

When $K = 0$ (no information), this expression becomes:

$$\begin{aligned} SCStorage - ECStorage &= ((N+1)!) * aproxlog(Z+1) \\ &\quad -aproxlog(Z+1) - N * aproxlog(Z) \\ &\quad -(N) aproxlog(N) \end{aligned}$$

where it is evident that the first (positive) term in the equation growths considerably faster that the negative terms.

□

For the sake of fairness, one should also show that when $K = N$ the difference is given by:

$$\begin{aligned} SCStorage - ECStorage &= (N+1) * aproxlog(Z+1) + (\sum_{i=1}^{N} i) * aproxlog(N) \\ &\quad -aproxlog(Z+1) - N * aproxlog(Z) \\ &\quad -(N+1) * (N) aproxlog(N) \end{aligned}$$

where the negative terms dominate the result. That is, when there is full information SC's consumption of space is lower than EC's. Nevertheless, one must indicate that the analysis is

these proofs is considerably biased against EC: 1) We penalised it for the sake of practicality (as explained above), 2) We ignore the transitivity law when storing information about time-points and 3) we have not considered that time-points could be represented by numerical symbols and therefore EC is ready to be part of a system were numbers are "built-in" (because they are also required with some other purposes).

A.4 Proof of proposition [ELEVA]

The style of following proof is taken from [Fun96]. Resolvents are called nodes (N_i is the i-th node in the derivation frontier) and they can contain implications. A node is divided into non-conditional goals, implications and residue: $N_i = \{\ldots\} + \{\ldots\} + \{\ldots\}$. Every item below represents the state of the *frontier* of nodes at a particular time. For the sake of brevity, derivations are not exhaustively described, but how to fill the gaps must be progressively evident from the context. The literal selected for the next resolution step is always indicated **like this**. Also, long predicates have been abbreviated and $\wedge$ is substituted by "," as in PROLOG.

1.- $N_1 = \{\mathbf{done(control, t_4, t_{100})}\} + \{\} + \{\}$
↓ with [DN01]

2.- $N_1 = \{(\mathbf{proc\ control\ begin\ C\ end}), done(C, t_4, t_{100})\} + \{\} + \{\}$
↓ "resolving against" ELEVATOR

3.- $N_1 = \{\mathbf{done(while\ on(N)\ do\ serve\_a\_floor\ ;\ park, t_4, t_{100})}\} + \{+\{\}$
↓ with [DN02]

4.- $N_1 = \{\mathbf{done(while\ on(N)\ do\ serve\_a\_floor, t_4, T_1)}, T_1 \leq T_2, done(park, T_2, t_{100})\} +$
$\{\} + \{\}$
↓ with [DN07]

5.- $N_1 = \{\neg\mathbf{holdsAt(on(N'), t_4)}, T_1 = t_4, done(park, T_1, t_{100})\} + \{\} + \{\}$
$N_2 = \{holdsAt(on(N'), t_4), done(serve\_a\_floor, t_4, T_1'), T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1), T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
↓ the negated literal is "transformed" into a conditional goal

6.- $N_1 = \{T_1 = t_4, done(park, T_1, t_{100})\} + \{\mathbf{false} \leftarrow \mathbf{holdsAt(on(N'), t_4)}\} + \{\}$
N_2 as before
↓

7.- $N_1 = \{\mathbf{false}\}$ because of step 8 to 12
$N_2 = \{holdsAt(on(N'), t_4), done(serve\_a\_floor, t_4, T_1'), T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1), T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
↓ with [DN13] an after dismissing N_1 (N_2 takes its place)
↓ and also thanks to the definition of $isfluent(F)$

8.- $N_1 = \{\mathbf{holds(on(N'), t_4)}, done(serve\_a\_floor, t_4, T_1'), T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1), T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
↓ with [EC1] (for simplicity, we use AEC not OAEC).

9.- $N_1 = \{do(Ag, A, T', T), \mathbf{init(A, T, on(N'))}, T < t_4, \neg clip(T, on(N'), t_4),$
$done(serve\_a\_floor, t_4, T_1'), T_1' < T_2', done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),$
$T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
↓ with [INI-03]

10.- $N_1 = \{\mathbf{do(Ag, turnon(N'), T', T)}, T < t_4, \neg clip(T, on(N'), t_4),$
$done(serve\_a\_floor, t_4, T_1'), T_1' < T_2',\ done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),$
$T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
$\downarrow$ with [DO-02] and [DO-03]

11.- $N_1 = \{\mathbf{t_3 < t_4}, \neg clip(t_3, on(5), t_4), \ldots\} + \{\} + \{\}$
$N_2 = \{t_3 < t_4, \neg clip(t_3, on(3), t_4), \ldots\} + \{\} + \{\}$
$\downarrow$

12.- $N_1 = \{done(serve\_a\_floor, t_4, T_1'), \ldots\} + \{\mathbf{false \leftarrow\ clip(t_3, on(5), t_4)}\} + \{\}$
N_2 as before
$\downarrow$ by using [EC2] and [ELE_H] the *clip* is dismissed.

13.- $N_1 = \{\mathbf{done(serve\_a\_floor, t_4, T_1')}, T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),\ done(park, T_1, t_{100})\} + \{\} + \{\}$
N_2 as before
$\downarrow$ [DN01] again

14.- $N_1 = \{\mathbf{(proc\ serve\_a\_floor\ begin\ C\ end)}, done(C, t_4, T_1'), T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),\ done(park, T_1, t_{100})\} + \{\} + \{\}$
N_2 as before
$\downarrow$

15.- The process continues and after a while one has:
$N_1 = \{\mathbf{done(\underline{if}\ currentfloor(C)\ \underline{then}\ if\ C = 5\ then}$
$\mathbf{begin\ turnoff(5)\ par\ open\ ;\ close\ end\ else\ if\ C < 5}$
$\mathbf{then\ begin\ addone(C, Nx)\ ;\ up(Nx)\ ;\ serve(N)}$
$\mathbf{serve(5)\ end\ else\ begin\ subone(C, Nx)\ ;\ down(Nx)\ ;end\ \underline{end}, t_4, T_1')}, T_1' < T_2',$
$done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),\ T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
$N_2 = \ldots$
$N_3 = \ldots$
$\downarrow$

16.- Eventually it reaches an expression like:
$N_1 = \{\mathbf{done(addone(4, Nx), t_4, T_1''')}, done(up(Nx), T_1''', T_1''),$
$done(serve(5), T_1'', T_1'), T_1' < T_2',\ done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),$
$T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
$N_2 = \ldots$
$N_3 = \ldots$
$N_4 = \ldots$
$\downarrow$ By abducing and executing *addone* ([DNEC0])
$\downarrow$ which takes the agent to time t_5

17.- $N_1 = \{\mathbf{done(up(5), t_5, T_1'')},$
$done(serve(5), T_1'', T_1'), T_1' < T_2',\ done(while\ on(N)\ do\ serve\_a\_floor, T_2', T_1),$
$T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{\}$
$N_2 = \ldots$
$N_3 = \ldots$
$N_4 = \ldots$
$\downarrow$ with [DNEC0] (base case of *done*, the definition of
$\downarrow$ *primitive* and the abductive strategy

18.- $N_1 = \{\mathbf{done(serve(5), T''_1, T'_1)}, \ldots$
$T_1 \leq T_2, done(park, T_2, t_{100})\} + \{\} + \{do(self, up(5), t_5, T''_1)\}$
$N_2 = \ldots$
$N_3 = \ldots$
$N_4 = \ldots$

This, of course, is only a selected trace of a proof showing how to "generate" one of the elements of [ELE_PLAN] $\{do(self, up(5), t_5, T''_1)\}$. The rest of [ELE_PLAN], however, can be similarly generated to complete the proof.

□

A.5 Traces of the simulated elevator

The figures in this subsection illustrate the behaviour of the elevator controller as simulated by the testbed described in chapter 6. We describe only two sequences of actions and events. The intention is to show that the GLORIA-like agent react to inputs from its environment and at the same time maintains a goal-oriented behaviour.

The program being traced is that described in figure 6.4, activated by integrity contraint 6.1. Recall that the elevator is an extended version of policy 3, as explain before in this chapter.

A.5.1 An agent that reacts to opportunities

The first sequence shows how the elevator controller takes new inputs into account, while it is trying to achieve some previously activated goal.

At the beginning, the situation is as shown in figure A.1. The elevator is at floor 1, having just served it, and there is no button on. At that time[7], the button at floor 5 is pressed (fig. A.2). Having processed that input ($on(5)$) and activated the goal of serving the fifth floor, the elevator starts moving up (fig. A.3). As the elevator is leaving the second floor in its way up, the button at floor 4 is pressed (fig. A.4). The system continues it upward movement until it reaches floor 4 (fig. A.5). Then, having attached a higher priority to the goal of serving floor 4, the elevator actually serves it (fig. A.6). Finally, the elevator reaches floor 5 (figure A.7) and serves it (figure A.8).

A.5.2 An agent that is faithful to its policy

The previous sequence does not challenge the specification of policy 3. The elevator is never in a situation that presents the dilemma of following or not the policy.

The following sequence presents that dilemma and it shows how the elevator controller is faithful to that specification of policy 3.

Policy 3 basically says that the elevator must visit first the next floor in its current direction of movement (up or down), if there is any. We can program this policy into the elevator controller by means of integrity contraints and OPENLOG procedures (as shown in chapter 4) or as PRIORLOG and ACTILOG rules (chapter 5).

Once again, the elevator is initially at floor 1 and it has been called to serve the fifth floor (figure A.9 and A.10). The elevator starts moving upwards (fig. A.11) and when it reaches floor 3, it realizes that the button at floor 1 has been pressed (fig. A.12). The elevator seems to ignore floor 1 (it does not. The goal has been activated but it has been given a lower priority

[7] The time is given by the "Current time" on the window. There are considerable gaps between the pictures' "current time". The reason is, of course, that we suspended (froze) the system to "take the picture".

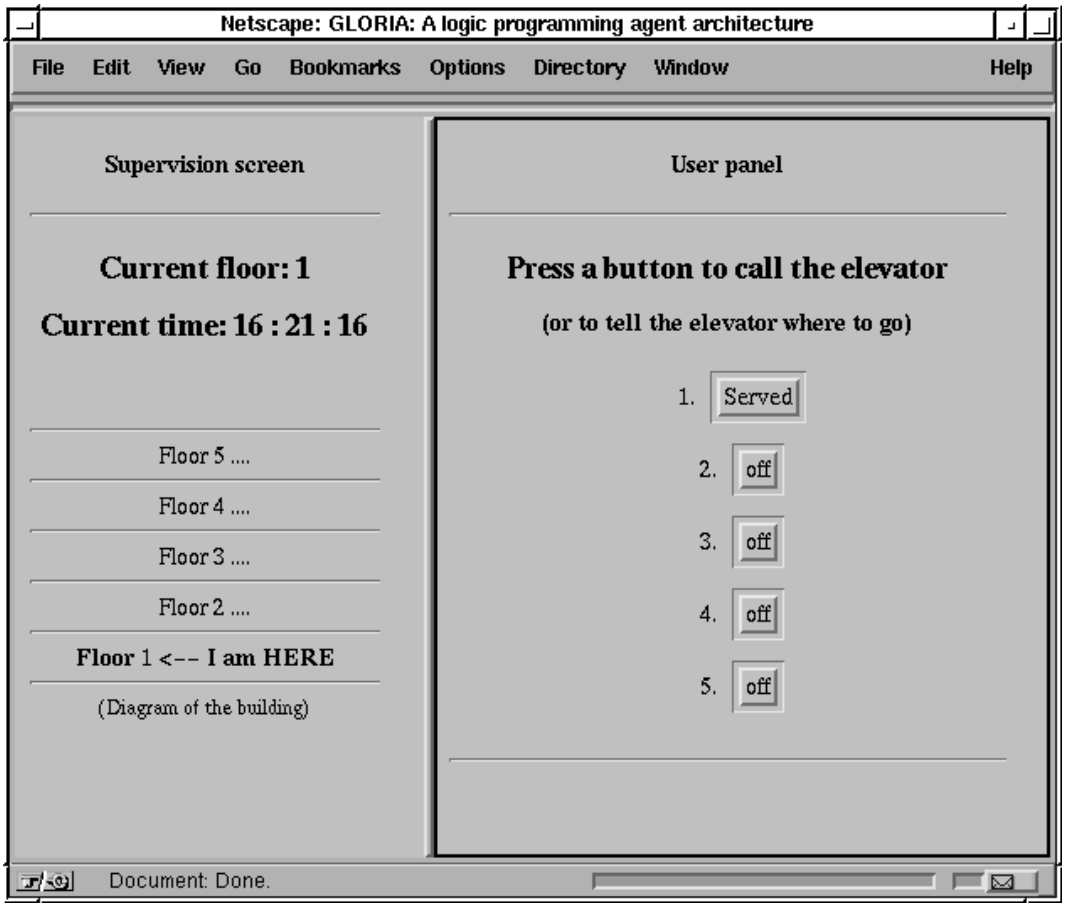


Figure A.1: The initial situation: the elevator at floor 1

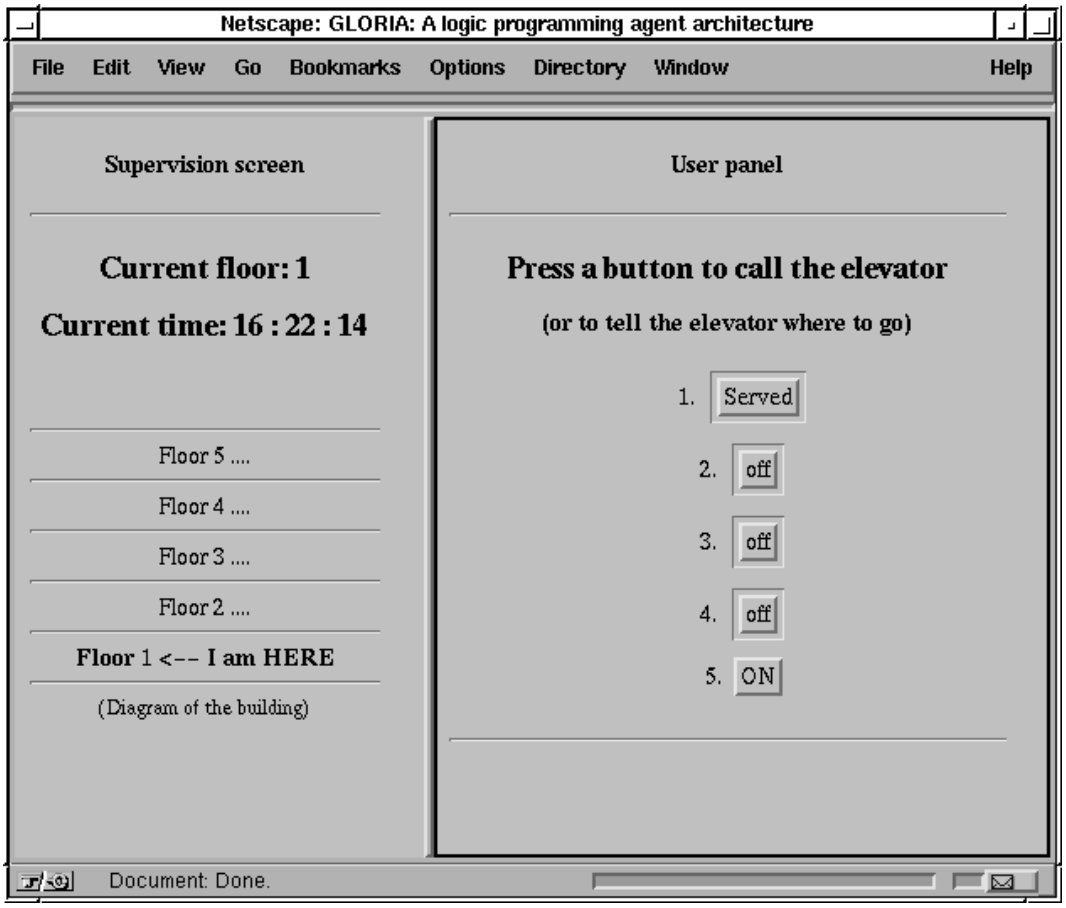


Figure A.2: The elevator has been called to serve the fifth floor

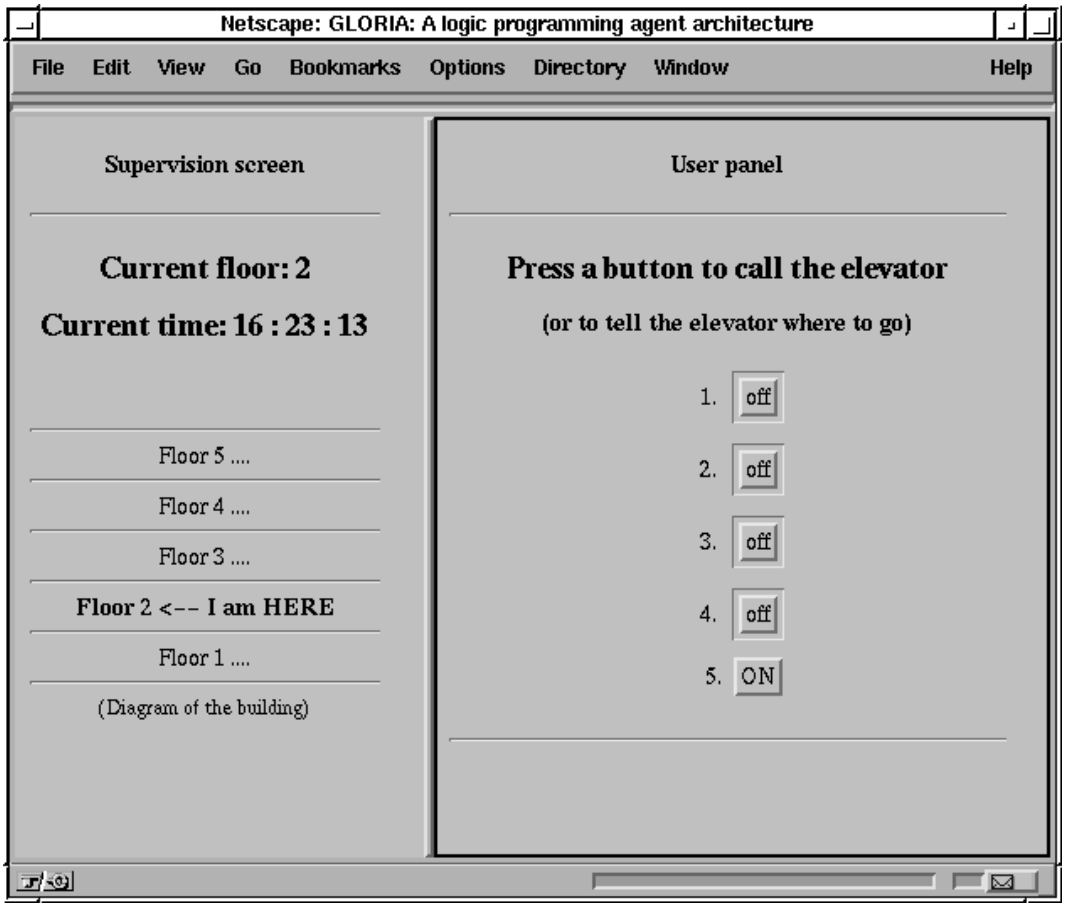


Figure A.3: At floor 2, moving towards the fifth floor

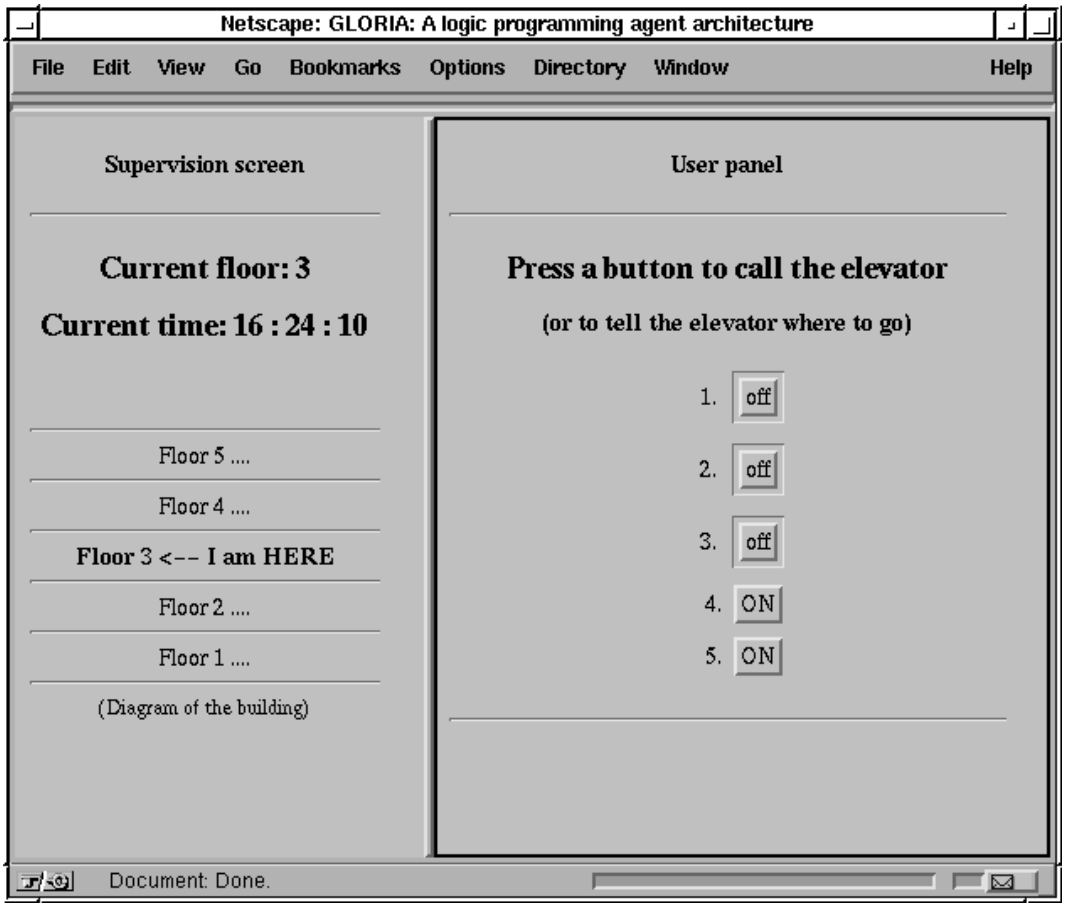


Figure A.4: The elevator has been called at floor 4

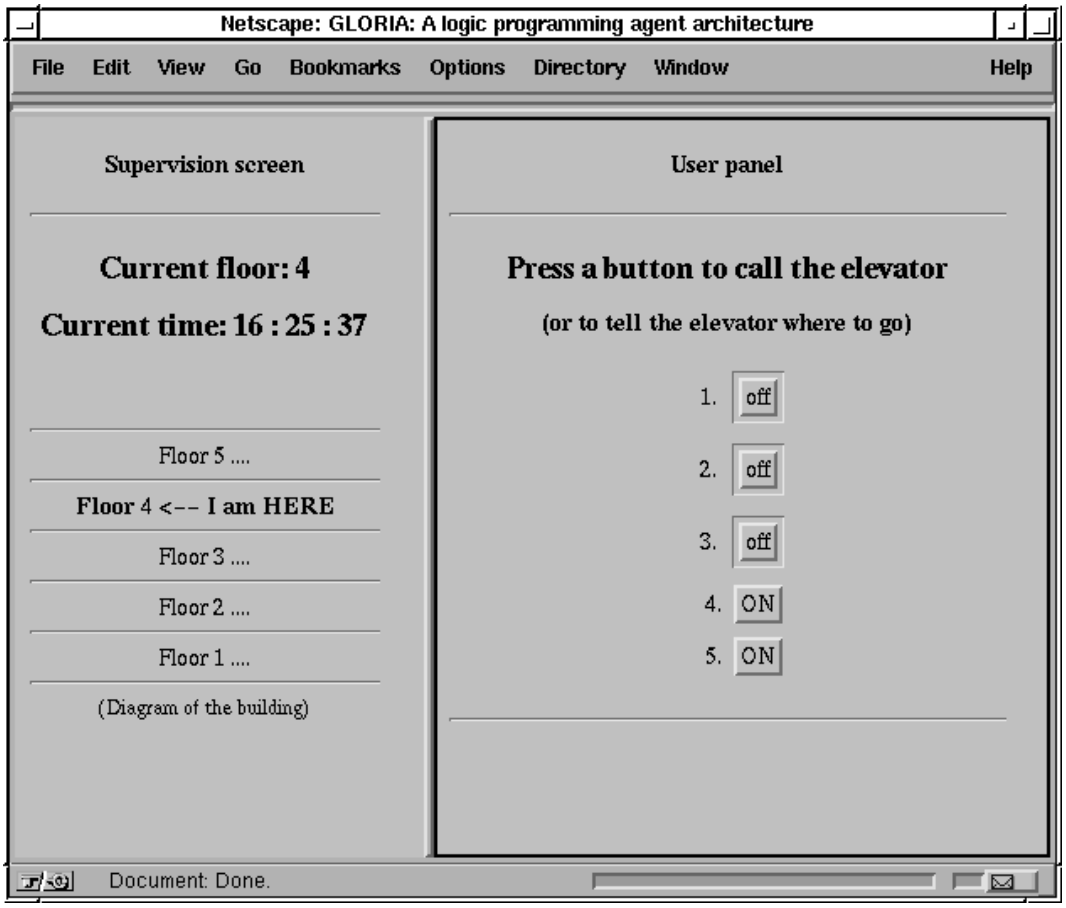


Figure A.5: The elevator reaches floor 4

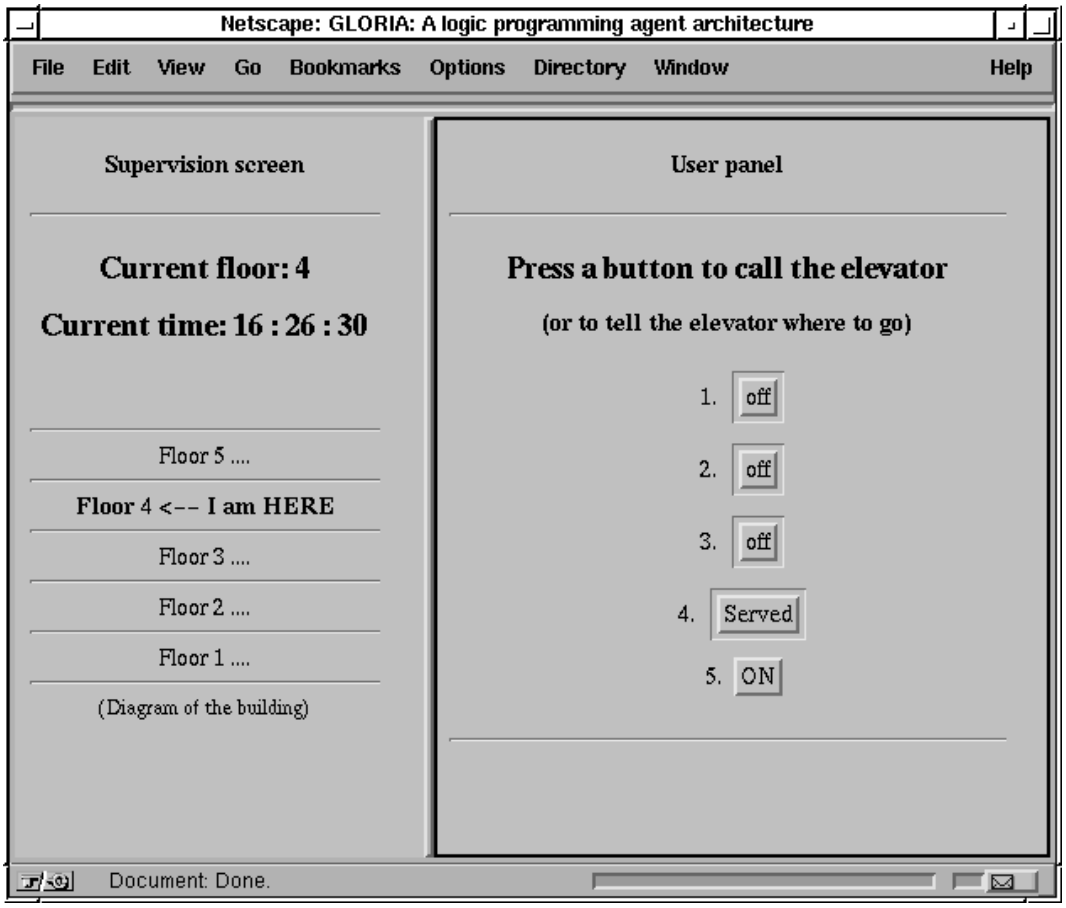


Figure A.6: The elevator serves floor 4

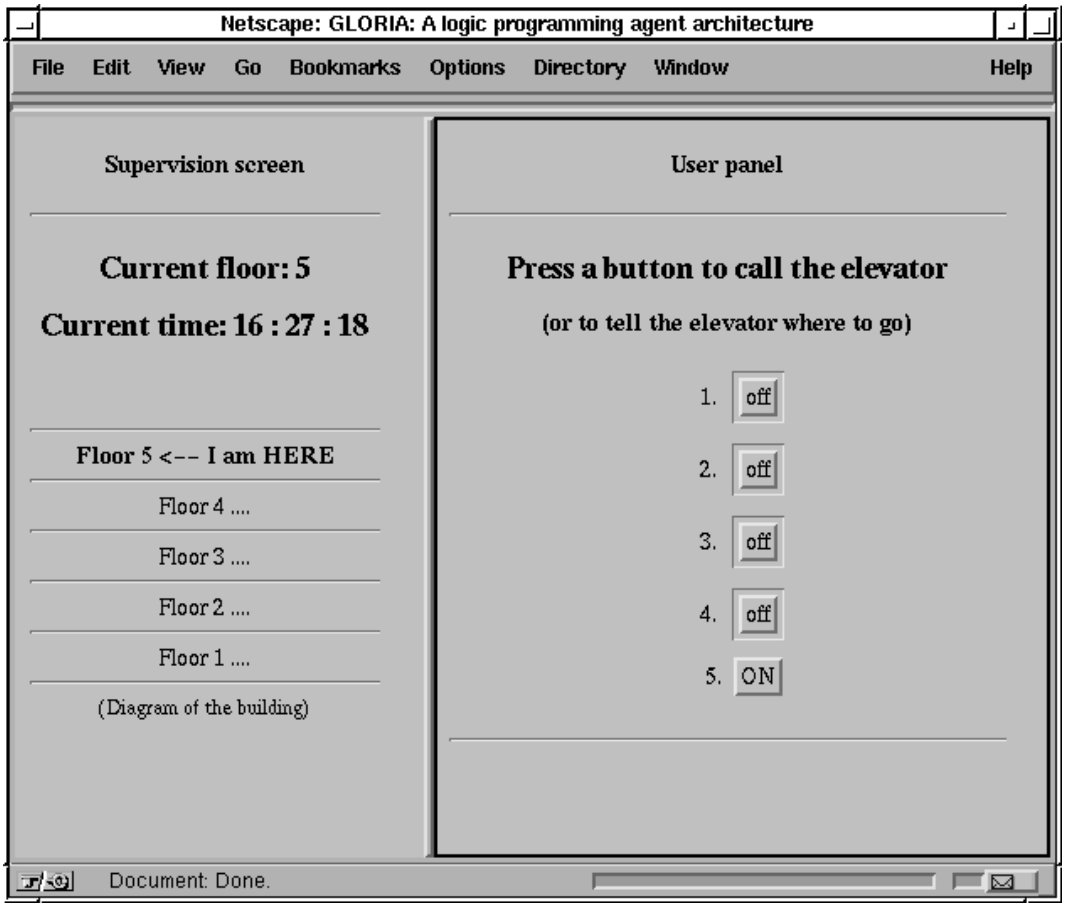


Figure A.7: The elevator reaches floor 5

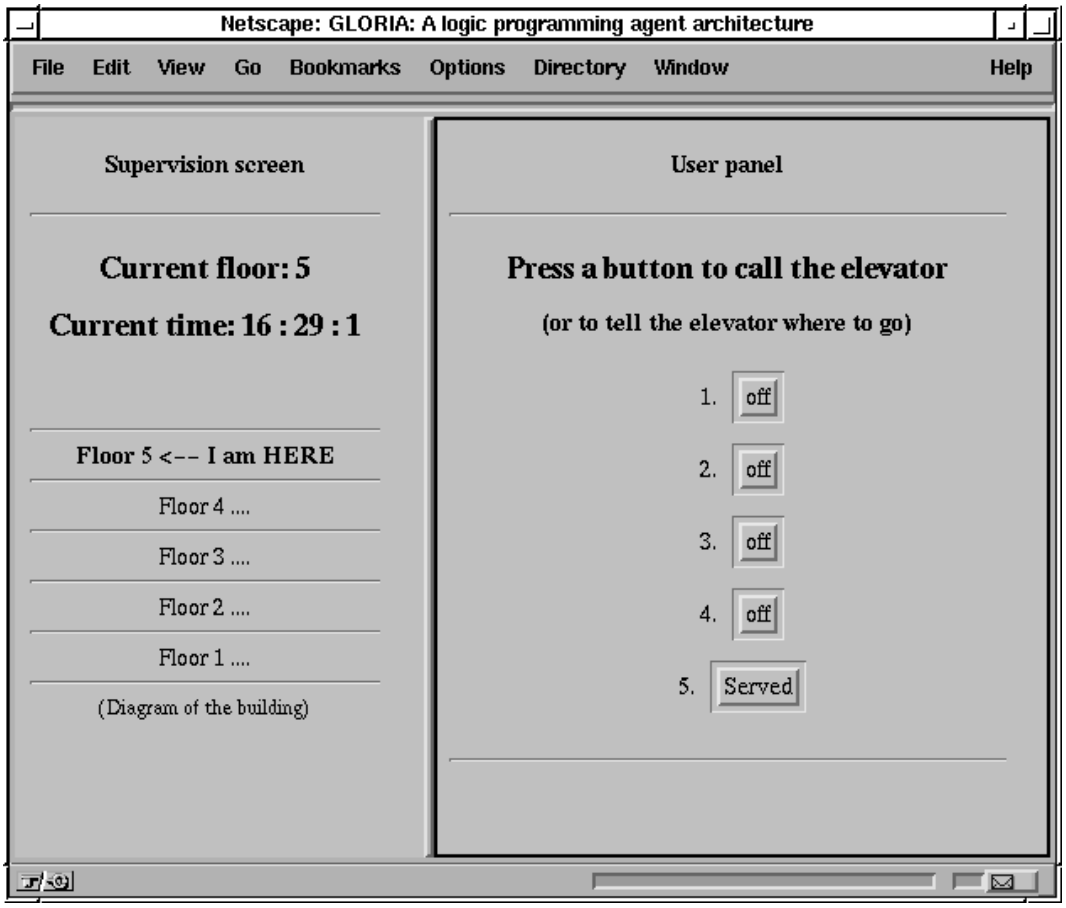


Figure A.8: The elevator serves floor 5

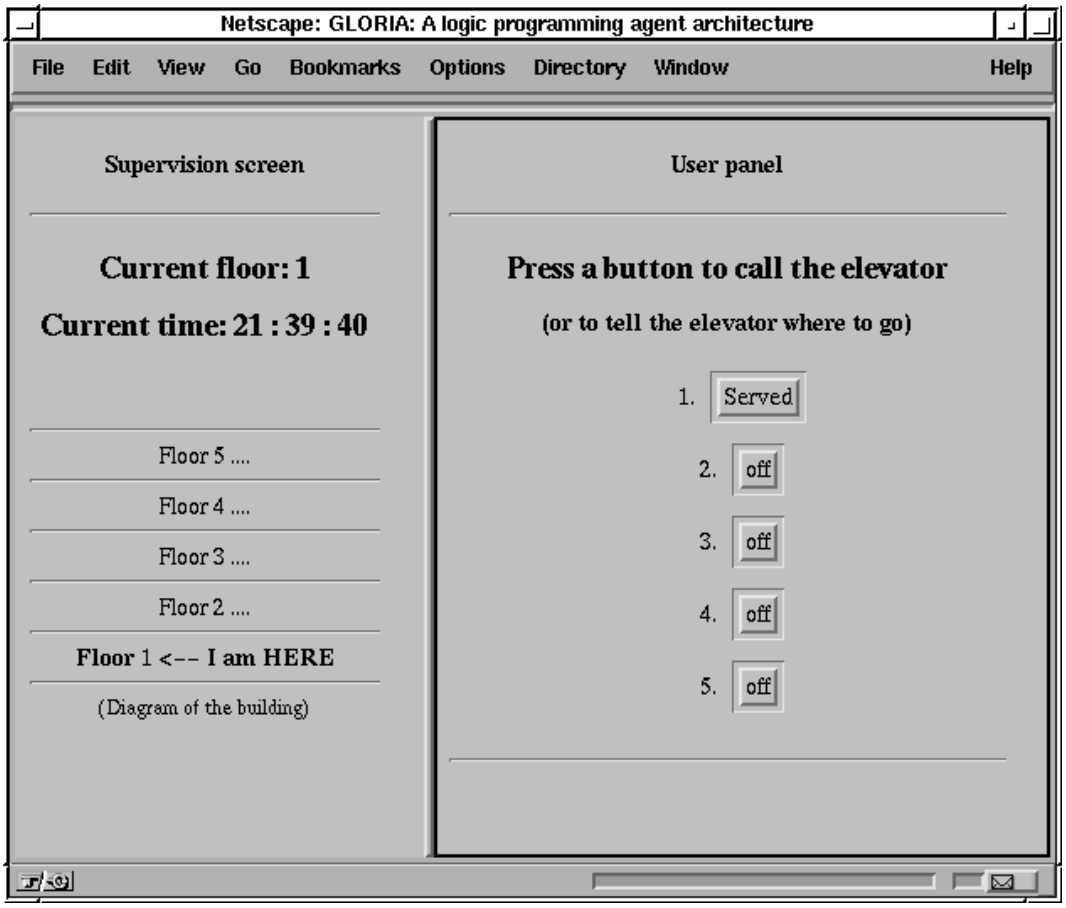


Figure A.9: Once again, the elevator is at the first floor

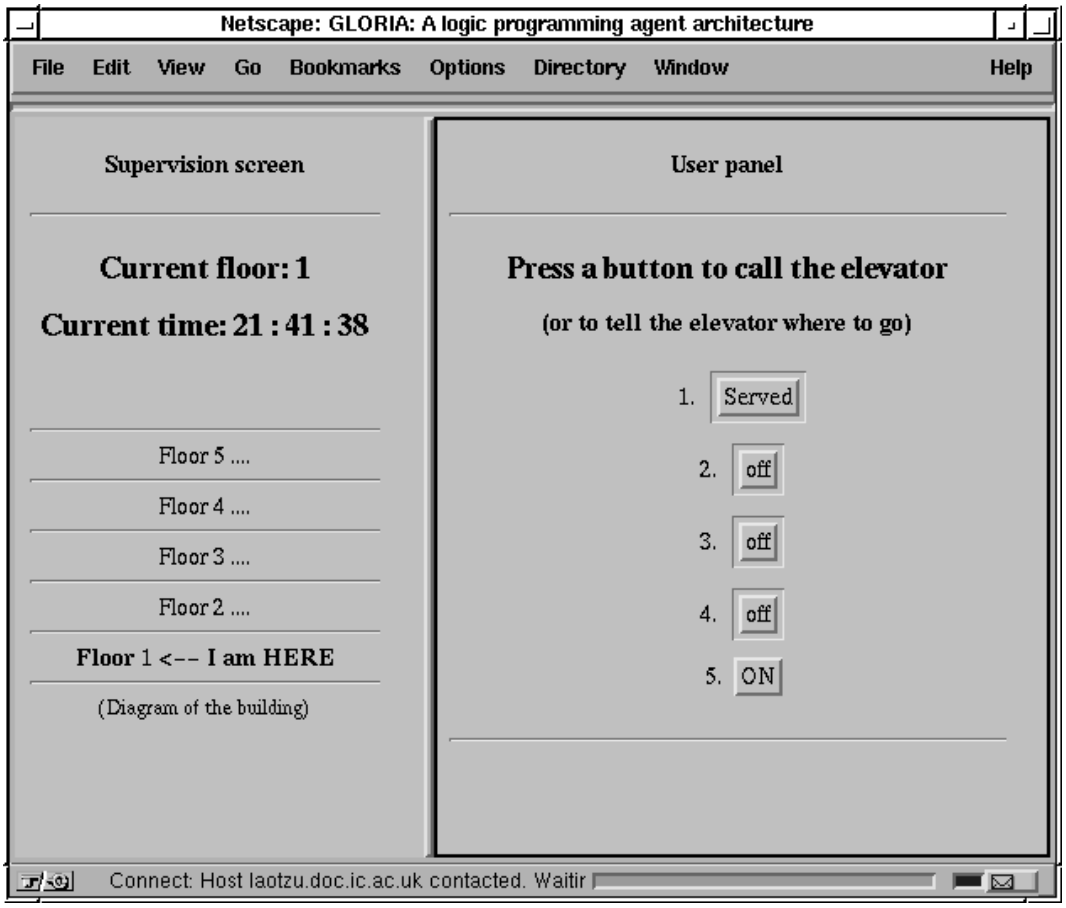


Figure A.10: .. and has to serve the fifth floor

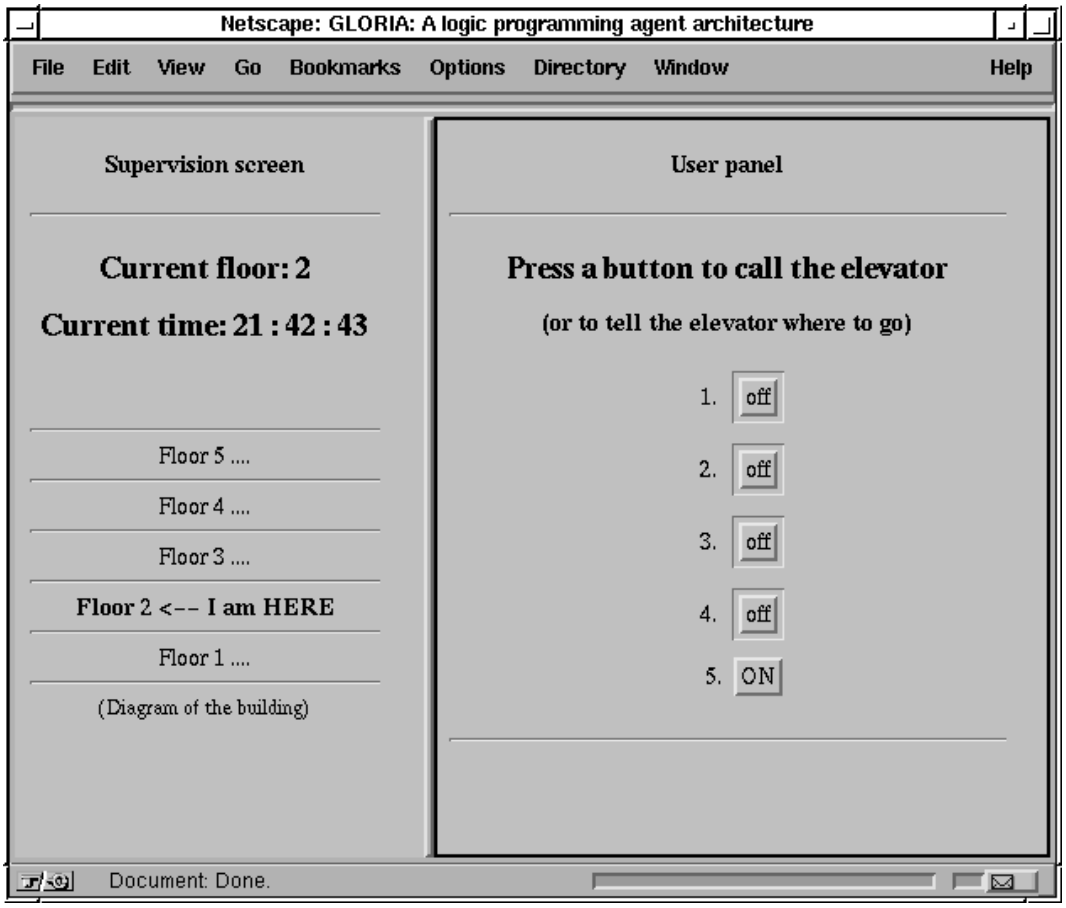


Figure A.11: It starts moving upwards

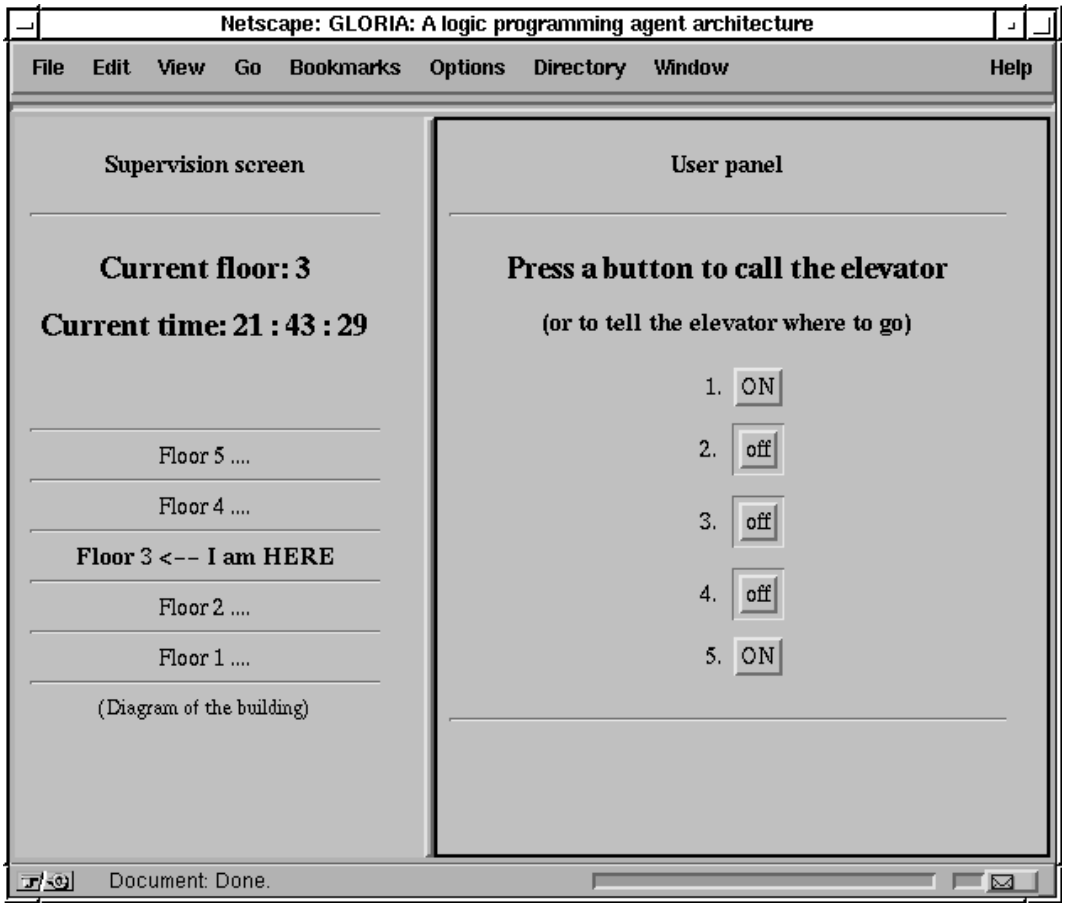


Figure A.12: The button is pressed at floor 1

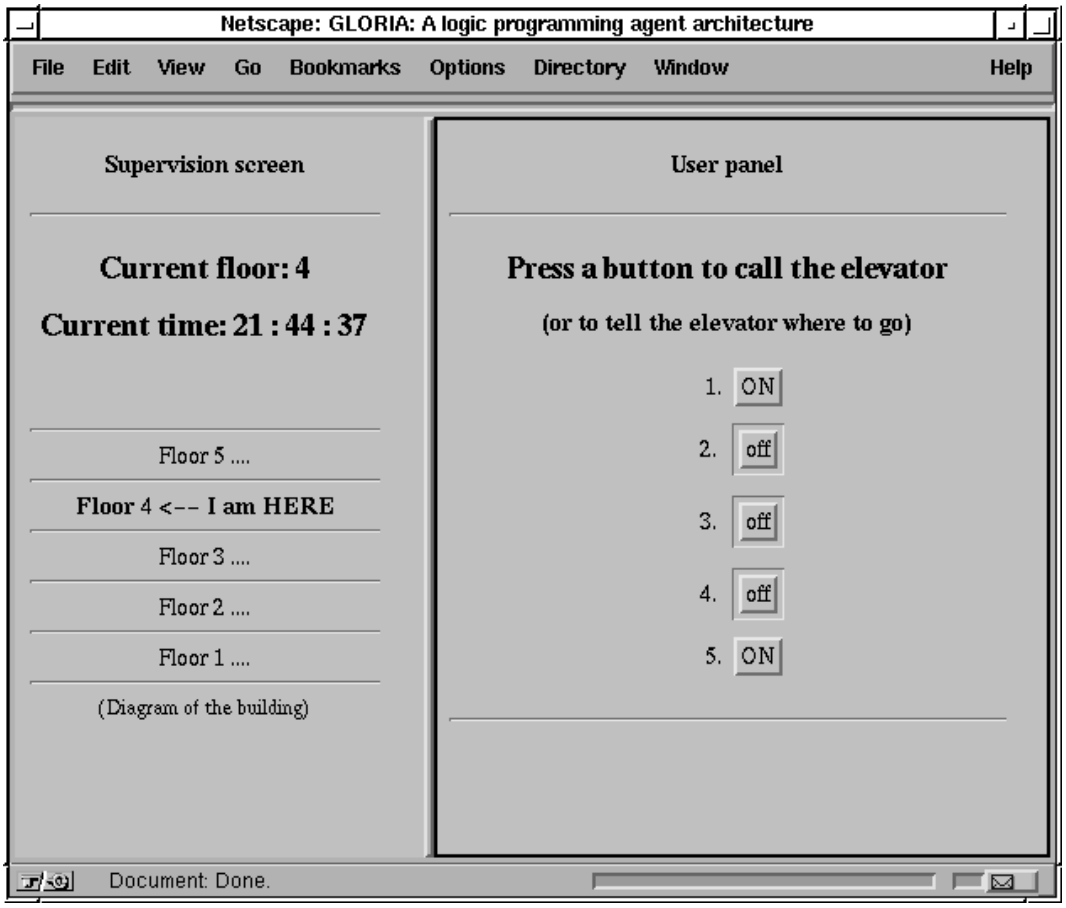


Figure A.13: .. but the elevator continues it movement towards the fifth floor

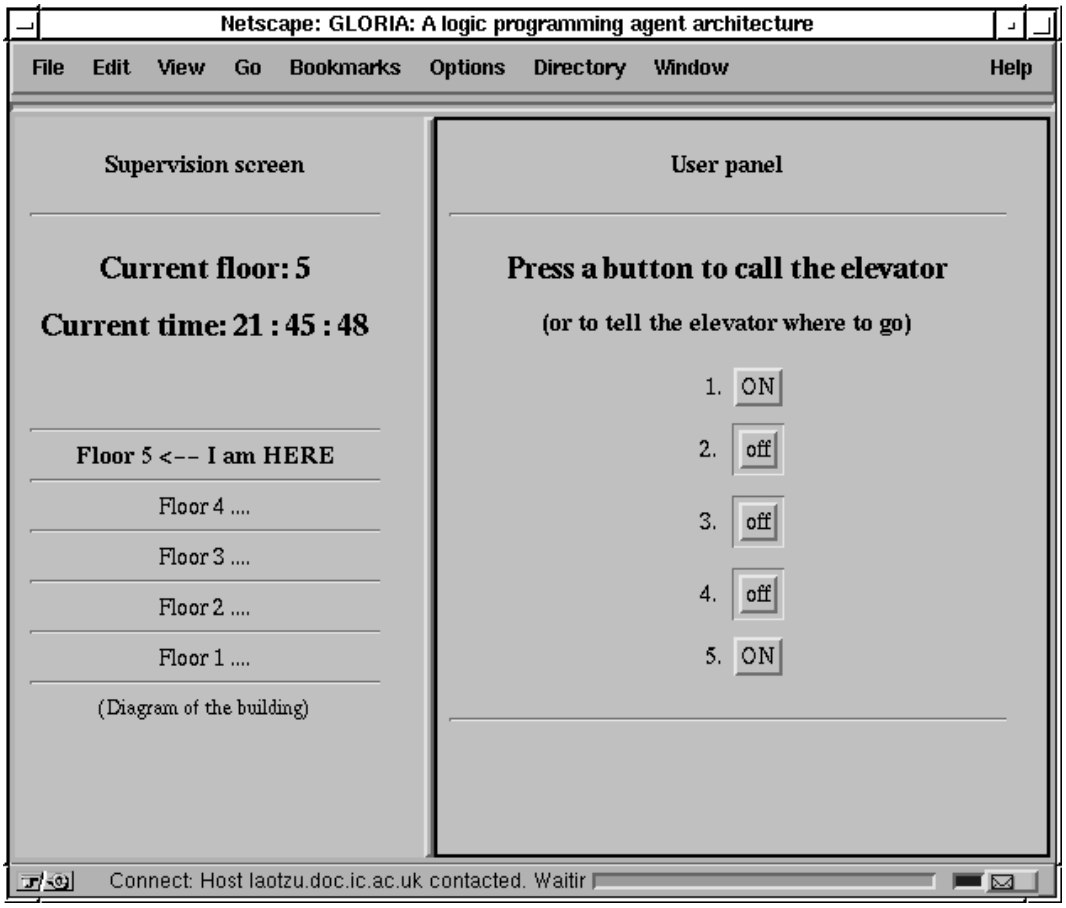


Figure A.14: Once again, it reaches the fifth floor

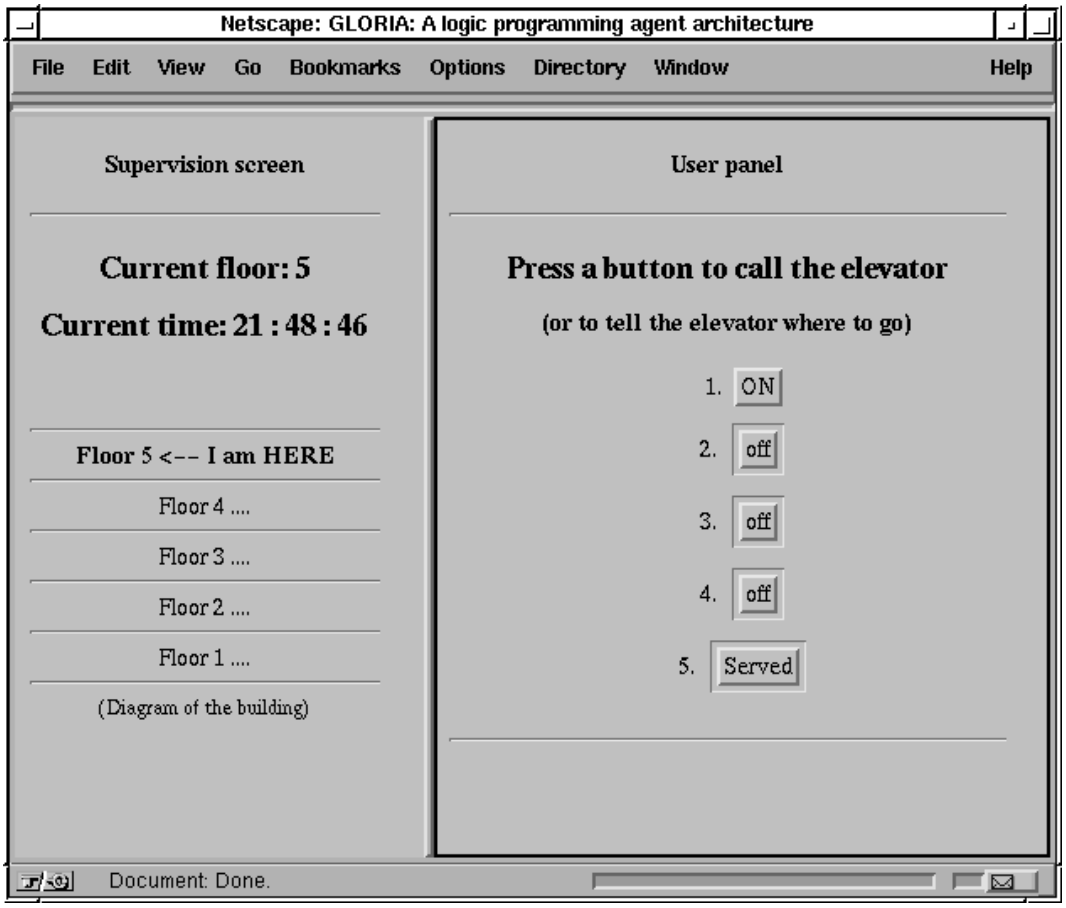


Figure A.15: And it serves the fifth floor

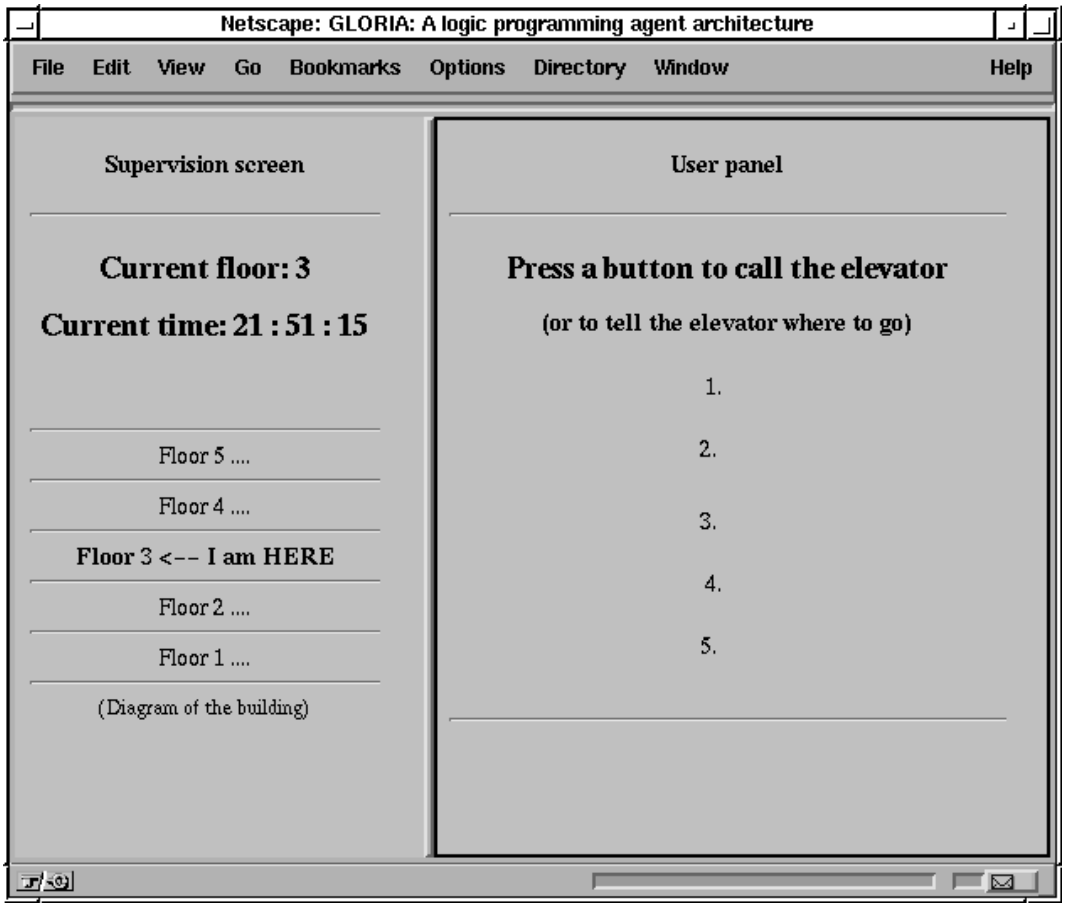


Figure A.16: Only then, it moves down to serve the first floor

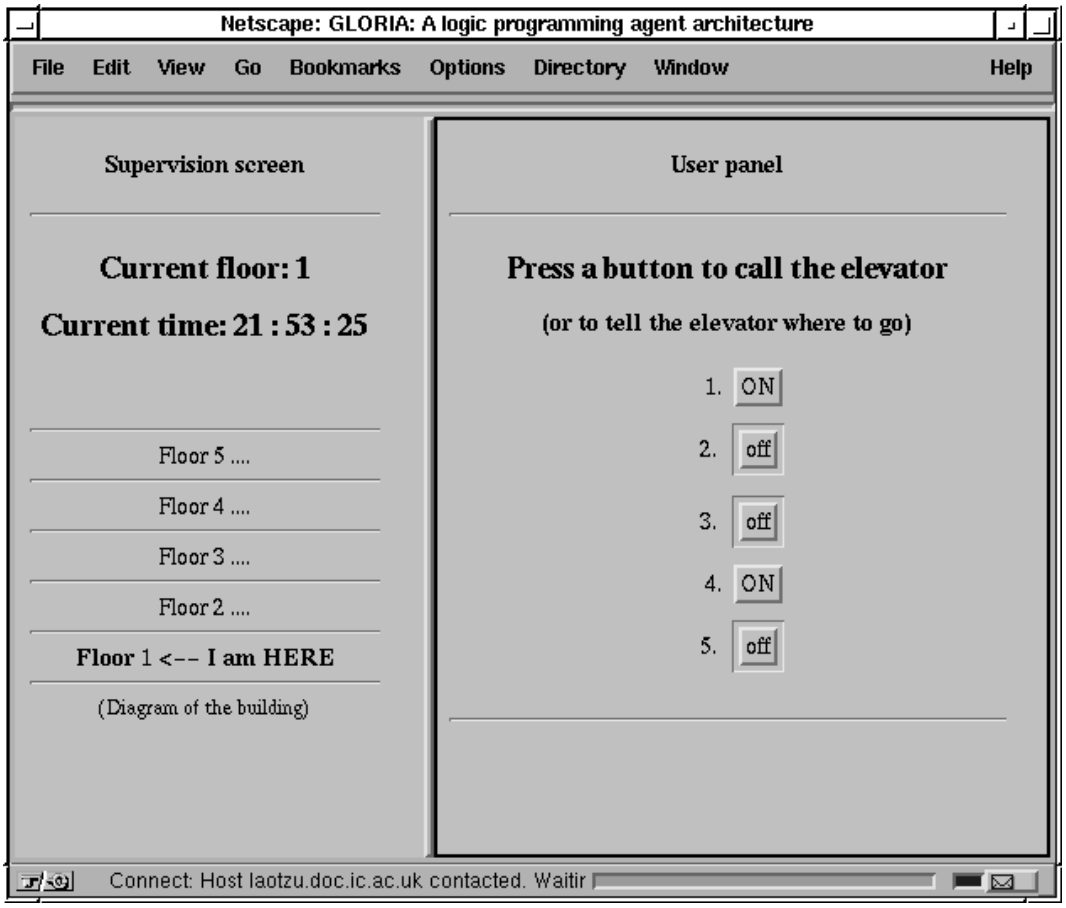


Figure A.17: Just before reaching the first, the button is pressed at the fourth

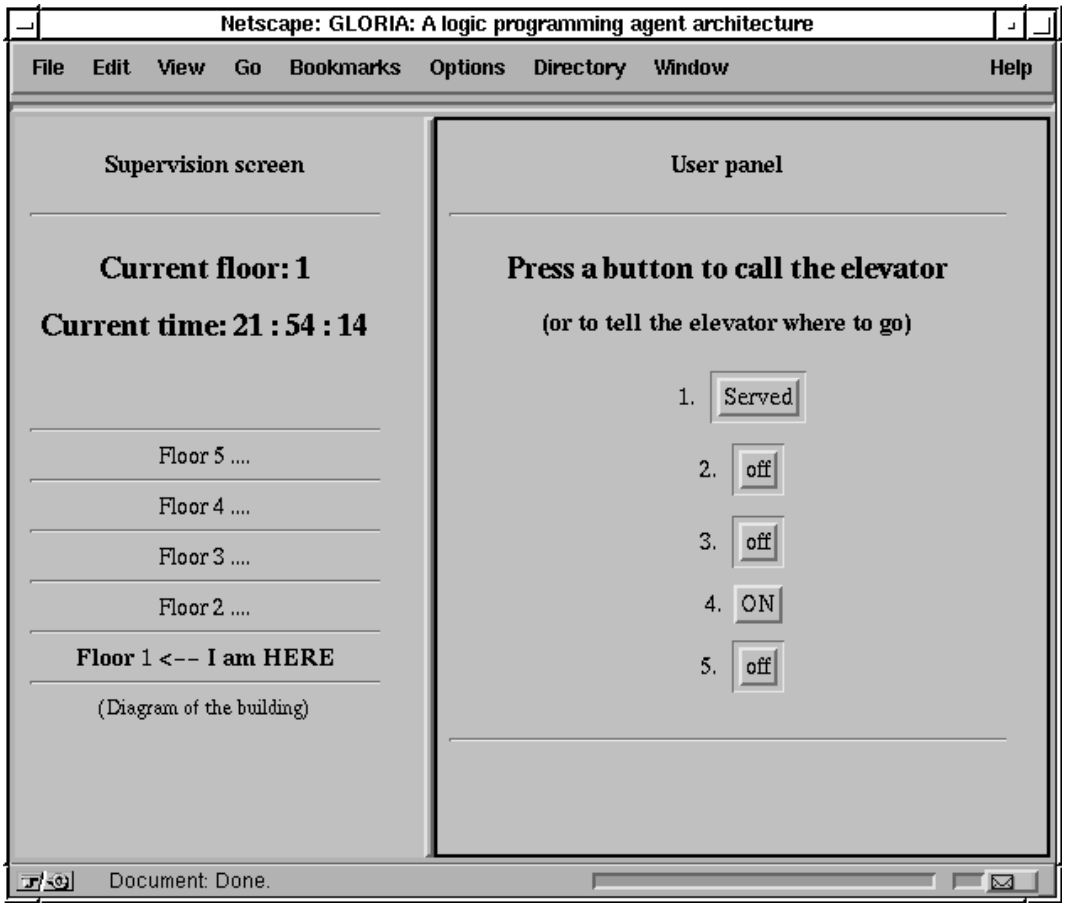


Figure A.18: But this agent will serve those on its way first

because it implies a change of direction) and it continues in the same direction to serve floor 5 (figures A.13, A.14 and A.15). Having served the fifth floor, the elevator comes down to serve the first floor (figures A.16, A.17 and A.18).

The sequence of events depicted by figures A.1 to A.8 and figures A.9 to A.18 may seem trivial. Surely, any system must be able to perform in that way. However, not any system can provide a logical description of this interaction: how inputs are assimilated and contribute to the activation of goals that determined the subsequent behaviour of the system. These are, we believe, the foundations of logical descriptions of more complex systems in which interactions involve communication and social behaviour.