LUND UNIVERSITY

Faculty of Engineering, LTH

Spring 2008

Master's Thesis

Automatic Theorem Proving in Active Logics

Supervisor: JACEK MALEC

Julien Delnaye

Automatic Theorem Proving in Active Logics

Abstract

Classical logic is useful for reasoning and knowledge representation in AI systems. But this theoretical formalism can show some limits when used with real-world agents. Indeed, it can be incompatible with computational constraints, real-time requirements and contradiction handling.

There are many approaches trying to overcome those limitations. One of them are active logics, studied for the last 20 years.

The purpose of this work is to set up an automatic theorem prover suitable for that kind of logics, with its different approaches. This paper first presents the active logics and some of its approaches, then describes the syntax and the methodology of the theorem prover.

Contents

Acknowledgements

I would like to thank my supervisor Jacek Malec for introducing me to the subject of resource-bounded agents and for encouraging me during my work.

Thanks also to Pascal Gribomont for giving me his feedback on my work.

Chapter 1

Introduction

1.1 Problem

Classical logic is a powerful methodology for reasoning and knowledge representation in AI systems. But it is not well-suited for real-world agents. Some important characteristics need to be taken into account when trying to formalize the reasoning of those real agents.

First, classical logic formalism does not make the distinction between the explicit beliefs of the agent and its implicit beliefs. That is, it does not consider the passage of time. All the conclusions following from a belief set - the implicit beliefs - are considered to be instantly known by the agent. This is called *logical omniscience*. But the real-world agents are resource-bounded, and at every point of time, an agent only knows a finite subset of the consequences - the explicit beliefs.

Secondly, classical logic can not handle inconsistency. It is *explosive*: everything follows from a contradiction. This is called the *swamping problem*. An agent reasoning in real world needs to be *paraconsistent* (or *inconsistency-tolerant*), ie it should be able to deal with contradictions rather than infer anything from them.

1.2 A new kind of logics

There are some attempts to constrain the inference process in order to guarantee polynomial-time computability and handle the resource limitations of real-world agents. We can quote for example the attempts to limit the expressive power of the first-order logical calculus, or the polynomial approximations of the reasoning process. However, those methods do not provide tight bounds on the reasoning process.

Another approach consists in retaining control over the inference process. Active logics appeared in this context. It is defined as follows (see [JEDPb]):

Definition 1.2.1. *An active logic consists of a formal language (typically first-order) and inference rules, such that the application of a rule depends not only on what formulae have (or have not) been proven so far but also on what formulae are in the "current" belief set. Not every previously proven formula need to be current; in general the current beliefs are only a subset of all formulae proven so far: each is believed when first proven but some may subsequently have been rejected.*

A number of new kinds of logics were developed to solve the limitations of classical logic. But active logics are the only one able to reason *in* time, and not only *about* time. In other words the agent can reason about the reasoning process itself.

The reasoning of an agent is viewed as a discrete-time ongoing process rather than as a fixed set of conclusions. That means that the conclusions are drawn step-by-step. In addition, the reasoner is able to take into account that time is passing while it is reasoning, and it can keep history of its reasoning. It can then make use of such information in subsequent deductions.

Another advantage of active logics is the contradictions handling. As a matter of fact active logics contain some meta-reasoning abilities - the agent is able to reason *about* its own knowledge. Contradictions handling is a part of this meta-reasoning. Note that meta-reasoning about some inconsistent knowledge still remains consistent.

Those two characteristics make the active logics formalism non-monotonic[1].

Definition 1.2.2. *A logical formalism is non-monotonic if adding a new belief to the knowledge base does not mean that this base will grow for sure, belief retraction being allowed.*

1.3 Some active logics approaches

The first incarnation of active logics was step logic. It was an attempt to formalize parts of a certain memory model - this memory model will be described in section 2.1. Jennifer Elgot-Drapkin proposed eight different step-logics (see [ED88]), with increasing complexity, depending on how they do or do not include some important mechanisms.

The problem of step logics, as we will see it, is that they are an oversimplification of the memory model, leading to computational issues.

In order to avoid this issue, Mikael Asker set up a new kind of active logic (see [Ask03]), based on Gabbay's *Labelled Deductive Systems* (LDS). Using this new formalism, the memory model can be entirely modelled.

[1]For instance if the agent knows at time t that the time is t, this belief has to be removed at time t+1.

The purpose of this thesis is to develop a theorem prover able to implement those different active logics.

1.4 Structure of this report

This report is structured as follows:

- Chaper 1: contains this introduction.
- Chaper 2: an introduction to step logics and the formalization of a memory model using LDS with active logics.
- Chaper 3: the description of the automatic theorem prover implementing LDS.
- Chaper 4: conclusions and future work.
- Appendix A: the documentation about the code of the theorem prover.
- Appendixes B and C: the results after the application of the prover on simple problems.

Chapter 2

Backgrounds

This chapter is an overview of Mikael Asker's thesis (see [Ask03]). We will first take a look at the memory model inspiring active logics. Then we will describe the step logic and LDS approachs.

2.1 A memory model inspired from cognitive psychology

During the 1980s, a model of the memory based on cognitive psychology was studied in the University of Maryland (see [JDP]). Cognitive psychology claims that one can infer some representations, structures and mental processes from the study of human behavior.

The model basically consisted of three parts coming from cognitive psychology:

- LTM, the *long term memory*, which contains beliefs that can be retrieved when it is necessary.
- STM, the *short term memory*, which acts as the current focus of attention.
- ITM, the *intermediate term memory*, which contains all facts that have been pushed out of the STM. The content of the ITM provides the history of the reasoner's reasoning process. ITM provides support for goal-directed behavior.

Classical logic is not suitable to formalize this memory model. Active logics started as an attempt to do it.

For practical reasons, two new parts have been added to the original model:

- QTM, the *quick term memory*, which is a technical device for buffering the next cycle's STM content.

- RTM, the *relevant term memory*, which is the repository for default reasoning and relevance. It contains all the beliefs of STM and the beliefs that have been pushed out of the STM but may still be important for default resolution.

Figure 2.1 shows how the parts are connected to each other.

The model is demonstrated to work on some examples of simple default reasoning.

Note that there is an interesting observation about the size of the STM. Experimentation has shown that a size of roughly eight is the smallest that leads to effective task-oriented behavior over several domains, and that larger sizes offer no advantage. This is in surprising agreement with psychological data on human short-term memory which has been measured to hold seven plus or minus two "chuncks" of data at any one time. It might be pure coincidence, but it may also indicate some important similarity between the memory model and the human short-term memory mechanism.

2.2 Step logic

Step logic was the first form of active logics. In fact, the term *step logic* is older than the term *active logic*. It was an attempt to formalize parts of the memory model described in previous section, by modelling the on-going process of reasoning.

There are 8 different step logics, with increasing complexity, depending on how they do or do not include the three following mechanisms:

- self-knowledge: capability of the agent to introspect what it does or does not know
- time: capability of the agent to allow the on-going process of deduction to be part of its reasoning
- retraction: non-monotonic reasoning

The different step logics and their respective included mechanisms are organized as follows (S = self-knowledge, T = time, R = retraction):

SL_0: none	SL_4: S, R
SL_1: S	SL_5: S, T
SL_2: T	SL_6: R, T
SL_3: R	SL_7: S, T, R

This is SL_7 that we are interested by in this thesis.

Intuitively, an agent is an inference mechanism that may receive new hypothesizes under the form of observations. A well-formed formula observed or inferred by the agent is called a belief. In order to present more precisely

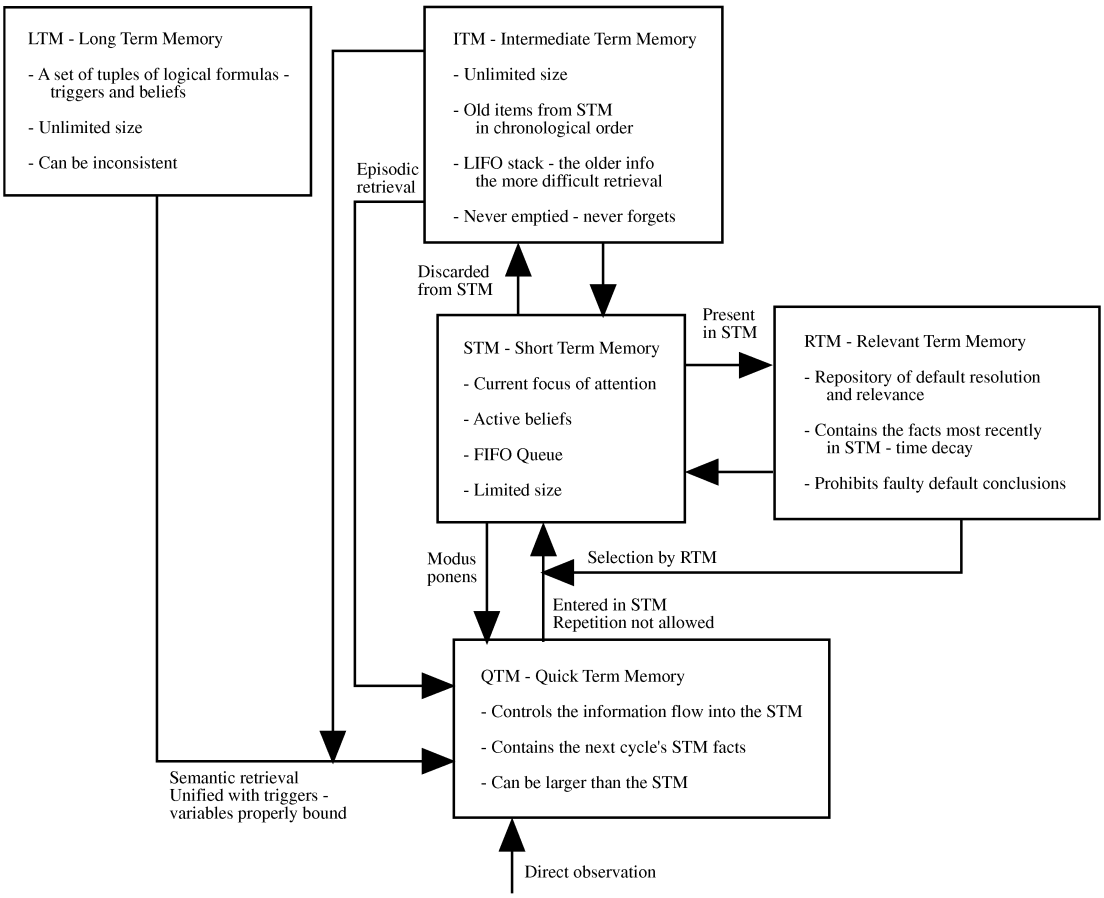

Figure 2.1: The memory model

step logic, some definitions are given, in which S_{wff} is the set of well-formed formulae of a first-order or propositional language.

Definition 2.2.1. *An observation function over a language L is a function* $OBS : \mathbb{N} \to \mathcal{P}(S_{wff})$*, where* $\mathcal{P}(S_{wff})$ *is the powerset of* S_{wff}*, which is the set of all well-formed formulae over the language L, and where for each* $i \in \mathbb{N}$*, the set* $OBS(i)$ *is finite. If* $\alpha \in OBS(i)$*, then* α *is called an i-observation.*

Definition 2.2.2. *A history is a finite tuple of pairs belief set/observation set, which are both finite subsets of* S_{wff}*.* $\mathcal{H}$ *denotes the set of all histories.*

Definition 2.2.3. *An inference function is a function* $INF : \mathcal{H} \to \mathcal{P}(S_{wff})$[1]*, where for each* $h \in \mathcal{H}$*, the set* $INF(h)$ *is finite.*

Intuitively, a history represents the temporal sequence up to a certain point in time. The inference function extends the temporal sequence of belief sets by one more step beyond the history.

Now we can define the step logic theories:

Definition 2.2.4. *An* SL_n*-theory over a language L is a triplet* $< L, OBS, INF >$*, where L is a first-order language, OBS is an observation function over L and INF is an inference function over L. L is implicitly defined by OBS and INF, so we can use the notation* $SL_n(OBS, INF)$*. At each step all the immediate consequences of the rules of inference applied to the history are drawn.*

For the following sections, it is also important to define what is an *i-theorem*:

Definition 2.2.5. *Let the set of 0-theorems, denoted* thm_0*, be* $INF(<< \emptyset, OBS(0) >>)$*. For* $i > 0$*, let the set of i-theorems, denoted* thm_i*, be* $INF(<< thm_0, OBS(1) >, < thm_1, OBS(2) >, ..., < thm_{i-1}, OBS(i) >>)$[1]*. We write* $SL_n(OBS, INF) \vdash_i \alpha$ *to mean that* α *is an i-theorem of* $SL_n(OBS, INF)$*.*

2.2.1 Inference rules

Here the inference function INF_B, defined in [JEDPa], is detailed.

The first two rules are the axioms. They express the awareness of time and the sets of observations returned by the observation function.

(A1)	$i : Now(i)$	for all $i \in \mathbb{Z}_+$	Clock
(A2)	$i : \alpha$	for all $\alpha \in OBS(i), i \in \mathbb{Z}_+$	Observations

[1] In practice, and in the other active logics in general, the inference function is only applied on the current belief set, and not on all the history.

A1 calls for a special predicate, $Now(i)$, meaning that the time of the current step is i.

Then come two versions of Modus Ponens.

(I1) $\dfrac{i : \alpha, \alpha \rightarrow \beta}{i+1 : \beta}$ MODUS PONENS

(I2) $\dfrac{i : P_1 a, ..., P_n a, (\forall x)[(P_1 x \wedge ... \wedge P_n x) \rightarrow Qx]}{i+1 : Qa}$ EXTENDED MODUS PONENS

The next rule, Negative Introspection, allows one to infer lack of knowledge about a particular formula that it is aware of at time i.

(I3) $i+1 : \neg K(i, \alpha)$ NEGATIVE INTROSPECTION[1]

The $K(i, \alpha)$ predicate means that the agent knows the formula α at the time step i (α is an *i-theorem*). The rule I3 uses meta-reasoning (this is the self-knowledge mechanism that is not included in each step logic). In any other inference rule of the inference function INF_b, the K predicate has no special status and is used just like any other predicate.

The next rule concerns the direct contradiction detection. This is a retraction mechanism used to avoid the swamping problem, which is inherent to standard logics.

(I4) $\dfrac{i : \alpha, \neg\alpha}{i+1 : Contra(i, \alpha, \neg\alpha)}$ CONTRADICTION DETECTION

Once again, the rule uses meta-reasoning. But here we can consider that the predicate Contra has a special status in the system regarding the last inference rule of INF_b. This is the inheritance rule. It propagates the knowledge from one time step to the next. It handles the detected contradictions through the Contra predicate and prevents the clock axioms from being inherited. We can see this rule as the non-monotonic part of the step logic.

(I5) $\dfrac{i : \alpha}{i+1 : \alpha}$ INHERITANCE[2]

[1] where α is not an *i-theorem*, but is a closed sub-formula at step i.

[2] where nothing of the form $Contra(i-1, \alpha, \beta)$ nor $Contra(i-1, \beta, \alpha)$ is an *i-theorem*, and where α is not of the form $Now(\beta)$.

2.3 The memory model as an LDS

Step logics are an oversimplification of the memory model, abandoning notably the focus of attention concept. So the set of beliefs grows rapidly and this growing never stops. This can lead to some computational issues. Those issues are of two kinds: it concerns a lack of memory and the impossibility for the agent to apply each inference rule to all the belief set during the fixed interval of time.

In order to avoid this issue, Mikael Asker set up a new kind of active logic (see [Ask03]), based on Gabbay's *Labelled Deductive Systems* (LDS). Using this new formalism, the memory model can be entirely modelled so that the belief set never exceeds a limited size.

In this section we will describe what are the Labelled Deductive Systems. Then we will show how we can use this tool to extend step logics in order to formalize completely the memory model.

2.3.1 Introduction to LDS

Traditionally a logic was perceived as a consequence relation on a set of formulae. Problems arising in some application areas have emphasized the need for consequence relations between *structures* of formulae. This approach called for an improved general framework in which many of the new logics arising from Artificial Intelligence applications can be presented and investigated. As a result of this, Gabbay presented the *Labelled Deductive Systems* (LSD) as such a unifying framework (see [Gab96]).

The first step in understanding LDS is to understand the intuitive message, which is very simple. Traditional logics manipulate formulae. An LDS manipulates *declarative units*, pairs *label:formula*. Although it sounds very simple, this formalism is considered to be a big step. As an analogy, the difference it makes is compared to the difference between using one hand only and allowing for the coordinated use of two hands.

The labels should be viewed as more information about the formulae that is not encoded inside the formulae.

LDS is a methodology, not a single system. Gabbay recommends usage of LDS only if it is convenient for the application. It should be used to simplify, not to complicate.

Definition 2.3.1. *An LDS proof system is a triplet* $(\mathcal{A},\boldsymbol{L},\mathbb{R})$ *where:*

- $\mathcal{A}$ *is an algebra of labels. It can be described by a labelling language, which can contain some operations, for example the addition or the maximum function.*
- $\boldsymbol{L}$ *is a logical language (connectives and well-formed formulae).*

- $\mathbb{R}$ *is a discipline of labelling formulae of the logic (with labels from the algebra $\mathcal{A}$), together with a notion of a database and a family of deduction rules and with agreed ways of propagating the labels via the application of the deduction rules.*

[Gab96] contains many examples of formulating major logics in LDS. But there were no attempt to apply it to active logics until Mikael Asker did it.

We could express the *SL_7-theory* as an LDS without much effort. The labelling algebra would be reduced to one single element: the time step i corresponding to the *i-theorem* in question. So we would obtain: $\mathbb{L}_{SL_7} \stackrel{df}{=} (\mathbb{N}, \mathbf{L}, \mathbb{R}_{SL_7})$, where $\mathbf{L}$ is a first-order language and $\mathbb{R}_{SL_7}$ is built around the LDS version of the inference rules (I1)-(I5) and the axioms (A1) and (A2). For instance, the Modus Ponens rule would become:

$$\frac{i : \alpha, i : \alpha \rightarrow \beta}{i + 1 : \beta}$$

We can see here, through the expression of step logics into LDS formalism, the concretization of the unifying formalism that Gabbay had in mind.

Note that all those new kinds of logics have not convinced the traditional logic community. Actually, they have not even accepted non-monotinic reasoning as logic yet. They believe that all this excitement is temporary generated by computer science, and that it will disappear sooner or later.

2.3.2 An extension of SL_7

Step logics are an oversimplification of the memory model. In particular, in the memory model the *short term memory* (STM) simulates the focus of attention of human reasoning. The limited size of the STM limits the number of inferences per step and allows to avoid the explosion of beliefs during the reasoning process. In step logics this limitation is omitted so that the number of formulae in each step may increase rapidly.

Our next step is to extend the SL_7 with LDS to include all aspects of the memory model. The following definitions describe the LDS system set up by Mikael Asker as part of his Master's Thesis (see [Ask03]).

In order to encompass all the complexity of the memory model, the label policy (the algebra of labels) here needs to be more complex as well. Namely,

$$S_{labels} \stackrel{df}{=} \{LTM, QTM, STM, ITM\} \times S_{wff} \times \{C, U\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

where the interpretation of a tuple in S_{labels} is the following. If

$$(location, trigger, certainty, time, position, time\text{-}left\text{-}in\text{-}rtm) \in S_{labels}$$

is a label, then:

- *location* encodes the memory bank location of the formula (LTM, QTM, STM or ITM).
- *trigger* is used for encoding the triggering formula for LTM items (in particular, ϵ or 0 is used to denote the empty triggering formula). The triggering formula is used for belief retrieval.
- *certainty* encodes the degree of certainty of a formula (certain or uncertain). It is used in case of defeasible reasoning.
- *position* denotes the formula's position in STM or ITM.
- *time-left-in-rtm* denotes the time the formula should remain in RTM. The RTM has no *position* value. A labelled formula is present in RTM and available for resolving contradictions when its *time-left-in-rtm* field is non-zero. $R \in \mathbb{N}$ is a constant used to limit the time a formula remains in RTM after it has left STM. The inference rules enter a formula into RTM by setting *time-left-in-rtm* to R when the formula enters STM. *time-left-in-rtm* remains at R until the formula leaves STM. When the formula has left STM and moved to ITM, *time-left-in-rtm* is decremented by one at each time step until it reaches zero, after which the formula should be pushed out of the RTM.

The set of axioms, S_{axioms}, is determined by the following rules:

(A1)	$(STM, \epsilon, C, i, i, 0) : Now(i)$	for all $i \in \mathbb{Z}_+$	Clock
(A2)	$(QTM, \epsilon, C, i, 0, 0) : \alpha$	for all $\alpha \in OBS(i), i \in \mathbb{Z}_+$	Obs
(A3)	$(LTM, \alpha, c, 0, 0, 0) : \beta$	for all formula tuples (α, c, β) present in LTM at time 0	LTM

The rules of inference describe not only which formulae may be derived from others but also the memory banks for the source and result formulae.

The first rule describes retrieval from LTM into QTM:

$$\text{(SR)} \quad \frac{(STM, \epsilon, c_1, i, p, R) : \alpha, (LTM, \beta, c_2, i, 0, 0) : \gamma, \alpha \mathcal{R}_{csf} \beta}{(QTM, \epsilon, c_2, i, 0, 0) : \gamma} \quad \text{Semantic Retrieval}$$

where $\alpha \mathcal{R}_{csf} \beta$ means 'α is a closed sub-formula of β'. The trigger formulae in the LTM labels together with $\mathcal{R}_{csf}$ limit the flow of information from LTM to QTM. This flow limitation is important, especially in this implementation where there is little selection when the information passes from QTM to STM.

Modus Ponens inferences are performed from STM to QTM:

$$\text{(MP)} \quad \frac{(STM, \epsilon, c_1, i, p_1, R) : \alpha, (STM, \epsilon, c_2, i, p_2, R) : \alpha \rightarrow \beta}{(QTM, \epsilon, min(c_1, c_2), i, 0, 0) : \beta} \quad \text{Modus Ponens}$$

$$\text{(EMP)} \quad \frac{\begin{array}{l}(STM, \epsilon, c_1, i, p_1, R) : P_1 a \\ ... \\ (STM, \epsilon, c_n, i, p_n, R) : P_n a \\ (STM, \epsilon, c_{n+1}, i, p_{n+1}, R) : (\forall x)[(P_1 x \wedge ... \wedge P_n x) \rightarrow Qx]\end{array}}{(QTM, \epsilon, min(c_1, ..., c_{n+1}), i, 0, 0) : Qa} \quad \text{Extended Modus Ponens}$$

where function *min* is defined over the set $\{U, C\}$ of certainty levels, with the natural ordering $U < C$. Of course in the implementation of the theorem prover, we will define the certainty levels with numbers, so that we will not have to define a special minimum function only for the certainty levels. The idea behind the certainty levels is that the status of a consequence should not be stronger than any of its premises.

The next rule allows the reasoner to introspect its lack of knowledge:

$$\text{(NI)} \quad \frac{\alpha \in f_{csf}(S_{STM}(i)), \alpha \notin f_{formulae}(S_{STM}(i))}{(QTM, \epsilon, C, i, 0, 0) : \neg K(i, \alpha)} \quad \text{Negative Introspection}$$

For this rule, we need to define some new concepts:

- $S_{theorems}$ is the set of all conclusions that can be drawn in the submitted *SL_n-theory*. We begin with the set of axioms then we look for all the conclusions that can be derived from it using the inference rules:

  $$S_{theorems} \stackrel{df}{=} \{\boxed{(l, t, c, j, p, r) : \alpha} \in S_{du} | S_{axioms} \vdash \boxed{(l, t, c, j, p, r) : \alpha}\}$$

  with $S_{du} \stackrel{df}{=} S_{labels} \times S_{wff}$
  where S_{wff} is the set of all well-formed formulae of SL_7.

- S_{STM} and its relatives are defined as follows:

  $$S_{QTM}(i) \stackrel{df}{=} \{\boxed{(l, t, c, j, p, r) : \alpha} \in S_{theorems} | (j = i) \wedge (l = QTM)\}$$

  $$S_{STM}(i) \stackrel{df}{=} \{\boxed{(l, t, c, j, p, r) : \alpha} \in S_{theorems} | (j = i) \wedge (l = STM)\}$$

  $$S_{new\text{-}STM}(i) \stackrel{df}{=} \{\boxed{(l, t, c, j, p, r) : \alpha} \in S_{STM}(i) | p = i\}$$

  $$S_{RTM}(i) \stackrel{df}{=} \{\boxed{(l, t, c, j, p, r) : \alpha} \in S_{theorems} | (j = i) \wedge ((l = STM) \vee ((l = ITM) \wedge (r > 0)))\}$$

- $f_{formulae}(S)$ extracts all the unlabelled formulae from the pairs *label* : *formula* of the set S:

$$S_{formulae}(S) \stackrel{df}{=} \{\alpha \in S_{wff} | (\exists label \in S_{labels})(\boxed{label : \alpha} \in S)\}$$

- $f_{csf}(S)$ extracts all the closed sub-formulae contained in the unlabelled formulae of S:

$$S_{csf}(S) \stackrel{df}{=} \{\alpha \in S_{wff} | (\exists \boxed{label : \beta} \in S)(\alpha \ \mathcal{R}_{csf} \ \beta)\}$$

The memory model as it was described in [JDP] uses and the *loses* predicate (depending on the level of certainty) instead of the *Contra* predicate used in step logics. Mikael Asker included both methods in his approach:

(CD1) $$\frac{(STM, \epsilon, c, i, p_1, R) : \alpha \quad (STM, \epsilon, c, i, p_2, R) : \neg\alpha}{(QTM, \epsilon, C, i, 0, 0) : Contra(i, \alpha, \neg\alpha)}$$ Contradiction Detection, same certainty

(CD2A) $$\frac{(STM, \epsilon, c_1, i, p_1, R) : \alpha \quad (STM, \epsilon, c_2, i, p_2, R) : \neg\alpha \quad c_1 < c_2}{(QTM, \epsilon, C, i, 0, 0) : loses(\alpha)}$$ Contradiction Detection, different certainties

(CD2B) $$\frac{(STM, \epsilon, c_1, i, p_1, R) : \alpha \quad (STM, \epsilon, c_2, i, p_2, R) : \neg\alpha \quad c_1 > c_2}{(QTM, \epsilon, C, i, 0, 0) : loses(\neg\alpha)}$$ Contradiction Detection, different certainties

The next group of rules handles inheritance, i.e. governs the time a particular formula stays in a memory bank or is moved to another one. The first inheritance rule says that eveything in LTM stays in LTM forever:

(IL) $$\frac{(LTM, \alpha, c, i, 0, 0) : \beta}{(LTM, \alpha, c, i + 1, 0, 0) : \beta}$$ Inheritance in LTM

The following rule concerns inheritance from QTM to STM. Remember that the conclusions of all the new inferences go to QTM. Like we pointed out when we described the memory model, QTM is a technical device for

buffering the next cycle's STM content. In this way we can retain control over the inheritance, like avoiding multiple copies of the same formula in STM or avoiding rework on formulae that have already been contradicted:

$$\text{(IQS)}\quad \frac{\begin{array}{l}(QTM, \epsilon, c, i, 0, 0) : \alpha \\ \alpha \notin f_{formulae}(S_{STM}(i)) \\ loses(\alpha) \notin f_{formulae}(S_{RTM}(i))\end{array}}{(STM, \epsilon, c, i+1, i+1, R) : \alpha} \quad \text{Inheritance QTM} \rightarrow \text{STM}$$

For the last inheritance rules the concept of focus of attention needs to be concretely developed. The STM is implemented as a FIFO queue of *sets* of declarative units rather than as a FIFO queue of declarative units. Each declarative unit from STM is part of one of these sets according to the value of its *position* argument. One problem with this "lazy" STM implementation is that limiting the number of non-empty sets in the STM does not necessarily limit the number of formulae in the STM at the same time. The flow from QTM to STM must be controlled to keep the amount of computation in realistic levels.

The constant S represents the size of STM. We can not just estimate that the declarative units that can stay in the STM are the ones in which the *position* argument has a value situated in $max(0, i+1-S)..i$. That would have had the effect that formulae would "time out" from STM into ITM even when no new formulae would have entered into STM. That is not the FIFO behavior described in [JDP]. In order to get a real FIFO behavior we let STM contain all the declarative units in which the position value is situated in $f_{min\text{-}STM\text{-}pos}(i)..i$, where $f_{min\text{-}STM\text{-}pos}(i)$ is calculated considering that S is actually the number of non-empty sets in the FIFO queue. The function $f_{min\text{-}STM\text{-}pos}(i)$ is defined in terms of another function:

$$f_{min\text{-}STM\text{-}pos}(i) \stackrel{df}{=} f_{STM\text{-}pos}(i, i, S)$$

where

$$f_{STM\text{-}pos}(i,p,s) \stackrel{df}{=} \begin{cases} p & \text{if } p = 0 \\ p+1 & \text{if } s = 0 \\ f_{STM\text{-}pos}(i, p-1, s-1) & \text{if } (\exists \boxed{(l, \alpha, c, i', p', r) : \beta} \in S_{STM}(i))(p' = p) \\ f_{STM\text{-}pos}(i, p-1, s) & \text{otherwise} \end{cases}$$

$f_{STM\text{-}pos} : \mathbb{N}^3 \rightarrow \mathbb{N}$ is a recursive function calculating the position of the s^{th} non-empty subset counting backwards from position p at time i.

This brings us to the inheritance from STM into STM and from STM into ITM. When new formulae are entered into STM from QTM, some old formulae must be pushed out of STM into ITM, to get a FIFO behavior and to limit the STM size. This is done by the (IS) and (ISI) rules:

$$\text{(IS)}\quad \frac{\begin{array}{l}(STM, \epsilon, c, i, p, R) : \alpha \\ (p > f_{min\text{-}STM\text{-}pos}(i)) \vee (S_{new\text{-}STM}(i+1) = \emptyset) \\ Contra(i-1, \alpha, \beta) \notin f_{formulae}(S_{STM}(i)) \\ Contra(i-1, \beta, \alpha) \notin f_{formulae}(S_{STM}(i)) \\ \alpha \neq Now(i) \\ (\alpha \neq \neg K(i-1, \beta)) \vee (\neg K(i, \beta) \notin S_{QTM}(i)) \\ (\alpha \neq Contra(i-1, \beta, \gamma)) \vee (Contra(i, \beta, \gamma) \notin S_{QTM}(i))\end{array}}{(STM, \epsilon, c, i+1, p, R) : \alpha}$$ Inheritance in STM

$$\text{(ISI)}\quad \frac{\begin{array}{l}(STM, \epsilon, c, i, p, R) : \alpha \\ (p = f_{min\text{-}STM\text{-}pos}(i)) \wedge (S_{new\text{-}STM}(i+1) \neq \emptyset) \\ Contra(i-1, \alpha, \beta) \notin f_{formulae}(S_{STM}(i)) \\ Contra(i-1, \beta, \alpha) \notin f_{formulae}(S_{STM}(i)) \\ (\alpha \neq \neg K(i-1, \beta)) \vee (\neg K(i, \beta) \notin S_{QTM}(i)) \\ (\alpha \neq Contra(i-1, \beta, \gamma)) \vee (Contra(i, \beta, \gamma) \notin S_{QTM}(i))\end{array}}{(ITM, \epsilon, c, i+1, p, R) : \alpha}$$ Inheritance STM → ITM

$$\text{(II)}\quad \frac{(ITM, \epsilon, c, i, p, r) : \alpha}{ITM, \epsilon, c, i+1, p, max(0, r-1)) : \alpha}$$ Inheritance in ITM

We can now define the LDS which is intended to be a formalization of the memory model:

$$\mathbb{L}_{mm} \stackrel{df}{=} (S_{labels}, \mathbf{L}, \mathbb{R}_{mm})$$

where $\mathbb{R}_{mm}$ is built around (SR), (MP), (EMP), (NI), (CD1), (CD2A), (CD2B), (IL), (IQS), (IS), (ISI) and (II).

Chapter 3

The automatic theorem prover

The source code is entirely contained in the file "ActiveLogics.pl". The documentation of the code is written in Appendix A. All the arguments are introduced to the prover via the file "arguments.txt". The source code and the different sets of arguments corresponding to the tests described in Appendixes B and C are available on the following web page:
http://ai.cs.lth.se/education.shtml.

3.1 Introduction

The best way to study how powerful is a methodology is to implement it. $\mathbb{L}_{mm}$ has only been tested on simple cases in which the focus of attention is never even full. We would like to see how the "engine" could manage realistically large problems.

The purpose of this work is not only to implement SL_7 and $\mathbb{L}_{mm}$ as they were described in the previous chapter, but allow the user to customize his own LDS system. He should be able to change the following settings:

- the number of arguments into the label policy
- the properties of the arguments in the label policy (words, symbols, numbers, first-order logical formulae, simple mathematical operations, ...)
- the inference rules

Such an implementation involves a very delicate complexity to manage. A lot of elements can be sources of errors. We will see that in the next sections.

Prolog is a well-suited programming language for LDS formalism and logics in general, as it easily manipulates lists. Logical programming, as functional programming, are the best choices for this kind of implementation. Making it in a classical language like C or Java would have unnecessarily grown the amount of work.

3.2 Structure of the arguments

3.2.1 Inference rules

There is a little difference between the modus operandi of step logics, like defined in 2.2, and the one of active logics in general. Indeed, the prover does not take into account all the history of beliefs to apply the inference function on it. It only takes the results of the last application of all the inference rules. So does the prover implemented in this word. We can see that this constraint is well-suited for the inference function INF_b of step logics, because they infer all the beliefs of time $i+1$ form beliefs of time i.

In revenge, two sets of inference rules are needed to suit the LDS approach of the memory model. This is due to the fact that the inference function of this approach is not applied to the belief set in one single step, but in fact uses an intermediate step. Indeed, some of the inference rules start from beliefs at the time step i to bring conclusions concerning the next time step $i+1$. The other inference rules bring conclusions concerning the same time step i than their premises. So the first kind of rules are applied to the belief set at the first time and the conclusions form therefore an intermediate belief set, then the second kind of rules are applied to this intermediate set. The new belief set consists of the regroupment of those last conclusions and the intermediate belief set. The procedure is represented in Figure 3.1. The structures of the inference sets are the following:

```
Rules1 = [Rule_1,
          ...,
          Rule_m]

Rules2 = [Rule_m+1,
          ...,
          Rule_n]
```

This separation of the beliefs is not needed in step logics. `Rules2` is left empty.

3.2.2 Axioms

The axioms are defined as the initial beliefs. The structure of the set of axioms is the following:

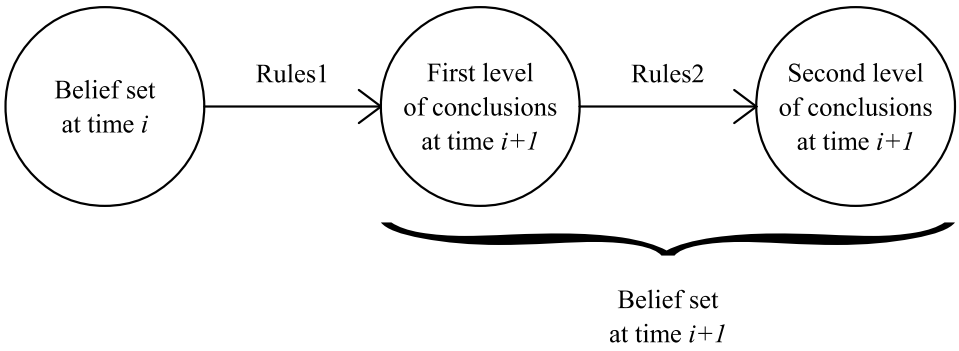

Figure 3.1: The two steps of inference

```
Axioms = [Axiom_1,
          ...,
          Axiom_p]
```

3.2.3 Observations

The set of observations has the same structure than the set of axioms. So we have:

```
Obs = [Observation_1,
       ...,
       Observation_q]
```

3.3 Syntax of the rules and beliefs

The key word of this topic is clearly: LISTS. We use lists to represent the labelled structures, as well as the formulae themselves.

3.3.1 Syntax of the beliefs

We added a little change to the initial formalism of pair $\boxed{label : formula}$. Since the arguments of a label can be formulae, we replaced the pair above by the simple expression *label*, or rather *belief*. So the syntax of a labelled belief would have the form `[Arg_1,Arg_2,...,Arg_n]`.

So if we apply this formalism to $\mathbb{L}_{mm}$, we will obtain the structure:

```
[location,trigger,certainty,time,position,time-left-in-rtm,formula,rule]
```

The last element is just an indicator of which rule has been used to obtain the belief.

For the software, every element of the belief is considered as a formula. The structure of a formula is based on nested lists. An argument of the

label that is not a formula is seen by the software as a formula with only one element. Each element of those nested lists representing a formula can be:

- a word or a symbol. Warning! If you attend to put in a belief a word beginning with a capital letter, you will need to put this word between quotes ('...') or the program will see it as one of its variables.
- a predicate written under the form `pred(Name,Arguments)` where `Name` is the name of the predicate and `Arguments` is the list of all the arguments of the predicate.
- a universal variable with the syntax `any(Name)`[1].
- a subformulae with the same structure.

For example the formula $(\forall x)[(bird(x) \wedge \neg ostrich(x)) \rightarrow flight(x)]$ could be inserted into the prover under the form:

```
['->',[and,pred(bird,[any(x)]),[not,pred(ostrich,[any(x)])]],
      pred(flight,[any(x)])]
```

or under the form[2]:

```
[[pred(bird,[any(x)]),and,[not,pred(ostrich,[any(x)])]],
 '->',pred(flight,[any(x)])]
```

3.3.2 Syntax of the inference rules

Basically, the structure of an inference rule is quite simple. It has the form `rule(Premises,Conclusion)` where `Premises` is a list of premises having a belief pattern or being special rules. The conclusion always has a pattern of belief.

Note that this is an inference device, not an axiom device: each inference rule needs at least one premise. Fortunately it is easy to translate an axiom rule into an inference rule, like the Clock axiom that just becomes the upgrading of the *Now* predicate.

It is obvious that the lists representing patterns of beliefs in the inference rules must have the same size than the beliefs in the belief set.

Because of the fact that the premises and the conclusion of a rule are not beliefs but *patterns* of beliefs, we need to introduce a new element: the variables. They are expressed like this: `var(Name)` where `Name` is the arbitrary name of the variable.

[1] The formulae do not have any prefixes. They are all considered to be the scopes of formulae in prenex form, and the universal variables are marked by the use of *any*.

[2] For assuring compatibility with some special functions of the prover, the first form is required.

The prover is able to perform some very simple mathematical operations: addition, substraction, maximum and minimum. The syntax for those operations is rigid and allows only one typing: `[operation,Arg_1,...,Arg_n]`. The operations can be nested. Remember that if any variable or non-numerical value remains in the nested lists of operations, the result will not be calculated.

The Extended Modus Ponens rule with its undefined number of premises calls for another improvement in the belief patterns. The special element `all(in,Name1,SubFormula1)` in a belief refers to a list of formulae where each one is equal to `SubFormula1` in which the `free(Name2)` elements have been replaced by some new variables. In response, each belief containing the element `all(out,Name1,SubFormula2)` actually refers to a set of beliefs where each occurrence of `all(out,Name1,SubFormula2)` have been replaced by a formula equal to `SubFormula2` in which each occurrence of `free(Name2)` is replaced by the corresponding variable. This tool is hard to understand like this but it will be easier in the next section where the prover's mechanisms are explained in more details.

Now that we have defined the premises under the form of beliefs, we can talk about the special premises. Indeed, there are some inference rules that need some premises that are more complex than just patterns of beliefs. This is the case for most of the inheritance rules. The way to write those premises is still using lists, but they do not have the same length than the beliefs and their first element has always the proper syntax `spec(SpecialFunction)`, in which `SpecialFunction` is replaced by the name of the special function, so that the syntax analyzer will not get confused.

The most important special premise is probably the one allowing access to the observation function. About the observations, their syntax is the same than the syntax of the beliefs. They manage the labelled structures with formulae, predicates, words and symbols, and the mathematical operations. The only difference is that their labelling policy can be different from the one of the beliefs. For example in the memory model, the labelling policy is simply $[i, formula]$ where i is the time step in which *formula* is observed. The call to this special premise is made this way:

```
[spec(obs),ObservationPattern]
```

where `ObservationPattern` has the same label policy than the observations and in which the variables have been introduced under the form `var(Name)` just like in the belief patterns.

The expression $\alpha \in f_{csf}(S)$ is not implemented in the prover as a verification function, but as a census function returning all the shortest closed sub-formulae of the set S, ie the positive and negative predicates. It does not return all the sub-formulae for computational matters. The syntax in the inference rules is the following:

```
[spec(f_csf),Pattern,Alpha]
```

where `Alpha` will be replaced by a variable that can be used in the following premises and where `Pattern` is a belief pattern used to constrain the set of beliefs from which the sub-formulae are taken. For example $\alpha \in f_{csf}(S_{STM}(3))$ could be translated by:

```
[spec(f_csf),
 [stm,0,var(c),3,var(p),var(r),var(formula),var(rule)],
 var(alpha)]
```

Note that this tool works as a verifier too: if we replace `Apha` by any predicate value, the program will check if it is a sub-formula of the set.

The four next functions are conditional functions. They can be used with negation.

The first one implements $\alpha \in f_{formulae}(S)$ or $\alpha \notin f_{formulae}(S)$, respectively:

```
[spec(f_forms),Pattern]
[spec(not,f_forms),Pattern]
```

where `Pattern` is a pattern of belief entailing at the same time all the constraints of the verification. For example

$$Contra(5, \alpha, \neg\alpha) \in f_{formulae}(S_{STM}(6))$$

could be translated by:

```
[spec(f_forms),
 [stm,0,var(c),6,var(p),var(r),
  pred(contra,[5,var(alpha),[not,var(alpha)]]),
  var(rule)]]
```

This device has no interest in its positive form because it could be replaced simply by its `Pattern` argument. The conditional test would be equivalent. But used with prefix `not`, it brings a new functionality.

The other conditional functions are very simple:

$\alpha = \beta$	`[spec(=),Alpha,Beta]`
$\alpha \neq \beta$	`[spec(not,=),Alpha,Beta]`
$x < y$	`[spec(<),X,Y]`
$x \geq y$	`[spec(not,<),X,Y]`
$\alpha\ \mathcal{R}_{csf}\ \beta$	`[spec(r_csf),Alpha,Beta]`
$\alpha\ \not\mathcal{R}_{csf}\ \beta$	`[spec(not,r_csf),Alpha,Beta]`

The special function `f_min_stm_pos` implementing the recursive function $f_{min\text{-}STM\text{-}pos}(i)$ is the only one that is totally devoted to $\mathbb{L}_{mm}$ LDS. The prover has not the ambition to be able to implement any given inference rule

without adding some functions into the programming code. The possibilities of special axioms are too wide to envisage covering them all. For the syntax of `f_min_stm_pos`, it is written this way:

```
[spec(f_lin_stm_pos),I,S,var(Name)]
```

where `I` is the current time step, `S` is the number of non-empty sets in STM (= the size of STM) and `var(Name)` is the variable in which the result will be written.

The special operator $\vee$ (OR) and the set delimiter $S_{new\text{-}STM}(i)$ have not been implemented, because they can be avoided in this case. We can rewrite the rules without using $\vee$, by just splitting one rule in two or more complementary rules. Concerning the second one, its implementation in the rules (IS) and (ISI) is very delicate, namely $S_{new\text{-}STM}(i+1)$ based on the beliefs at time i, because it calls for a window on future conclusions. We can easily get rid of it without important consequences because it only affects the size of STM of one single unit. We could just add one unit to the size of STM and say, about the computational time we lose at time i because of this unit, that we recover it from not implementing $S_{new\text{-}STM}(i+1)$.

In Section 3.5, one can see how $\mathbb{L}_{mm}$ is translated into the prover's formalism.

3.4 Functioning of the prover

In this section we will describe in details how the prover works, ie how it creates a new set of beliefs on the basis of an existing set of beliefs and a set of inference rules. It is easy to understand that each inference rule of the set is applied to the existing set of beliefs, drawing all the possible conclusions, then those conclusions are added to the conclusions drawn by the other inference rules to form the new set of beliefs. So we will focus on how the prover manages one single inference rule applied on the existing set of beliefs.

First we will see how the prover makes the link between the premises of an inference rule that have the pattern of beliefs and the beliefs themselves, by some kind of correlation analysis. In this topic will will also see the management of some particular patterns of beliefs. After that we will describe how the special premises that we talked about in the previous section are implemented. Some complexity issues will be addressed as well as how we tried to solve them.

3.4.1 Syntax analyzer

The most important part of the inference engine concerns processing of formulae and beliefs in general. Although some special premises are used a

lot, the only premises that are used for sure are the ones that are patterns of beliefs and with which the prover has to find correlations with elements of the belief set. The most important rules are often based only on patterns of beliefs, like the unavoidable Modus Ponens rule. In the ideal case that kind of rules would be the only ones needed, but the attempts to formalize the human reasoning calls for more complicated rules. Anyway the conclusion of any rule has the pattern of a belief because new beliefs are what we want at the end.

The fundamental way of processing is that the prover analyzes the premises one by one, in their initial order from top to bottom, and uses the correlation between the analyzed premise and a belief to "upgrade" all the following premises and the conclusion. Each upgrading of the premises adds constraints on them so it tightens the number of beliefs that could be correlated with those premises, and by the same it transforms little by little the conclusion into a real belief. If no correlation is found for some upgraded (or not upgraded) premise, the prover considers that the set of conclusions from this upgraded premise is empty. But the operation is repeated for each belief that can be correlated with the upstream premise so the prover stopping on an uncorrelated upgraded belief does not mean that the rule will not have any valid conclusion.

Now let us have a look on the way the correlations are found. Variables play a very important role in this part. And there are not only variables in the premises but also in some beliefs. Indeed, before applying an inference rule to a belief set, the program transforms all the universal variables (`any(Name1)`) of the belief set into prover-processed variables (`var(Name2)`), the same than in the inference rules. The program ensures that none of the names (that are actually numbers) of the transformed variables are already used in the inference rule. The newly-created variables have different names in each belief, even if they had the same name when they were universal variables, because they refer to independent formulae. In the same vein, after the analysis of all the premises, if some found conclusions still have variables, those variables will be transformed in turn into universal variables.

When the prover is looking for a correlation between a premise and a belief, that happens in two steps: possible correlation finding and confirmation of the correlation. Of course the correlation finding can be given up after the first step. This first step analyzes the separated correlations for each element of the labelled structure. So the belief and the premise must have the same number of elements to envisage a correlation. Two corresponding elements, one from the belief and the other from the premise, are designated for a potential correlation if they fulfill one of the following conditions (remember that the prover considers every element as a formula):

- They are exactly the same.
- At least one of them is a variable. In this case the prover will keep

track of the element that is not a variable (actually it could also be a variable) as the value of the variable.

- They are both formulae in which all the corresponding sub-formulae fulfill one of these conditions.
- One of them is a simple operation of nested additions and substractions and containing at least one variable. To keep track of a value for this variable, the prover will reverse the operation between the two elements.

The tests are made in the same order than this listing above.

The treatment in the case of the fourth condition is a source of complexity and potential errors, because the *min* and *max* operations can not be reversed. Anyway there is almost no reason to use such operators outside the conclusion. It is the responsibility of the user to make sure that all the variables amongst the arguments of a *min* or *max* operation, which can itself be used as an argument in an addition or a substraction, will be replaced by numbers before the analysis of the premise they belong to.

Since the tests are made independently for each element separately, the prover needs to apply a second step checking that there is no conflict between the possible different values given to a same variable. For that the prover has got a list regrouping all the identified variables and their respective values. Some variables, identified or not, are part of the "values" of other identified variables, for example we could have the value of `var(alpha)` being:

```
['->',pred(intrusion,[var(room)]),pred(call,[police])]
```

We can keep the basis of this example to explain how the confirmation step works. Of course if the variable `var(alpha)` is defined more than once with each time the same value, the prover will validate this variable and just keep one occurrence of its definition. But in the other case, the fact that the different definitions are not exactly the same does not mean that the correlation can not be validate. they can still be compatible. Another definition of `var(alpha)` could be:

```
['->',pred(intrusion,[livingroom]),pred(call,[police])]
```

So the variable `var(alpha)` would be constrained to the definition just above and we would have a new value for the already or not yet identified variable `var(room)` that would be `livingroom`. If `var(room)` was already identified, its new definition will have to be compatible will the existing ones. There is unfortunately a part of this step we can not really retain control on: when a variable is part of its own definition. In such a case the prover simply deletes the definition. Most of the time this is the right thing to do, because the deleted definition was something like:

```
[+,1,[-,var(x),1]]
```

Such a definition is impossible to solve for the prover on its current state (at least in the way it solves that kind of operations), because all the nested arguments are not numbers. Sometimes deleting the definition can also lead to errors. The nested operators clearly constitute the major source of uncertainties and errors. It has to be used very carefully when designing a system, the simpler the better. One of the purposes of future work concerning this theorem prover could be a device able to simplify the nested operations even if some of the arguments are unreplaced variables, in order to increase the robustness of the analyzer. But for now there is a simple formalism that the user should adopt to limit the source of errors: always keep the same structure form incrementing or decrementing a variable, ie avoid writing `[+,var(x),1]` and `[+,1,var(x)]` in the same system. This is due to the fact that during the possible correlation finding step the third test is applied before the fourth, therefore it is possible to avoid useless reversing of operations.

The upgrading of the next premises will be processed using this 'clean' set of variable values. The principle is simple, each occurrence of a variable present in the set is replaced by its definition. And each *min*, *max*, *addition*, *substraction* operation is solved when possible.

Now that we have described the basic mechanisms of the inference engine, we should illustrate them by an example. The Modus Ponens rule of the step logic approach lends itself to this example judiciously:

```
rule([[var(i),var(alpha)],
      [var(i),['->',var(alpha),var(beta)]]],
     [[+,var(i),1],var(beta)])
```

We imagine a belief set containing the two following beliefs:

```
[5,pred(intrusion,[livingroom])]
[5,['->',pred(intrusion,[any(room)]),pred(call,[police])]]
```

First of all, the beliefs are translated in order to transform universal variables into prover-processed variables. The names (actually the numbers) of the new variables only depend on the ones that are already present in the inference rule and in the other beliefs:

```
[5,pred(intrusion,[livingroom])]
[5,['->',pred(intrusion,[var(1)]),pred(call,[police])]]
```

Both of those beliefs are compatible with the first premise, but the second belief would lead to an upgrading of the second premise that would probably be incompatible with all the belief set. In revenge, if we correlate the first belief with the first premise and upgrade the other premises and the conclusion, the rule will become:

```
rule([[5,['->',pred(intrusion,[livingroom]),var(beta)]]],
     [6,var(beta)])
```

Then the second belief is still compatible with the premise, leading to the conclusion:

```
[6,pred(call,[police])]
```

A last point to be addressed in this topic is the management of an undefined number of premises, ie the `all(...)` term. When the prover detects the presence of an `all(...)` predicate in a premise, its behavior becomes different:

- If the premise contains at least one predicate `all(out,...,...)` without containing a predicate `all(in,...,...)`, then the premise is simply moved to the end of the premises set. The reason is that all the `all(out,Name,...)` predicates depend on the `all(in,Name,...)` predicate. If it was not the case this mechanism would be computationally unacceptable. It would be like planning an itinerary without knowing the destination and then see if the itinerary matches the destination.

- If the premise contains at least one predicate `all(in,Name,SubForm)`, the prover will first try to match some very tolerant correlation between the premise and the belief. Actually it only checks the correlation between the sub-formula pattern `SubForm` and the corresponding ensemble of sub-formulae (if they exist) in the belief. Then it will remodel the inference rule, including the analyzed premise, according to the number of those corresponding sub-formulae. Each predicate `all(in,Name,SubForm2)` contained in a premise or in the conclusion will be replaced by a list of Sub-formulae based on the pattern SubForm2 in which the `free(...)` elements will be replaced by new variables in accordance with the replacements of the `free(...)` elements in the other beliefs. This is why the prover requires that the syntax of the formulae was prefixed[1] (see 3.3.1), like in HP calculators, so that the prover does not have to separate the different arguments of an operation by the shared operator (like $\wedge$ or $\vee$). The premises containing the predicate `all(out,Name,SubForm3)` will be replaced by a group of same premises in which this `all` predicate will be replaced by a sub-formula based on the pattern SubForm3 where the `free(...)` elements are replaced by variables with the same accordance than before.

As an example we analyze the Extended Modus Ponens rule of the step logics approach:

[1]Except for that, the logic used by the user can be quite flexible and is implicitly defined by the inference rules, the axioms and the observations.

```
rule([[var(i),all(out,emp,free(form))]],
      [var(i),['->',[and|all(in,emp,free(form))],var(q)]],
     [[+,var(i),1],var(q)])
```

where the formalism `[El|Ls]` means that the element El is implanted as the first element of the list Ls. First the prover detects `all(out,emp,free(form))` in the first premise without detecting any `all(in,...,...)` so it will move the premise to the end of the set of premises:

```
rule([[var(i),['->',[and|all(in,emp,free(form))],var(impl)]],
      [var(i),all(out,emp,free(form))]],
     [[+,var(i),1],var(impl)])
```

In this case the prover detects in the first premise the presence of the predicate `all(in,emp,free(form))`. Thus for each belief that could be correlated with the premise, the inference rule will be remodeled. For example if we consider a belief set, after translating the universal variables into prover-processed variables, containing the following belief:

```
[5,['->',[and,pred(bird,[var(12)]),[not,pred(ostrich,[var(12)])]],
         pred(flight,[var(12)])]]
```

Then the only correlation that the prover will check is the one between `[all(in,emp,free(form))]` and:

```
[pred(bird,[var(12)]),[not,pred(ostrich,[var(12)])]]
```

This correlation is verified and the only information that the prover will withdraw from it is the number of elements substituting the `all` predicate, 2 in this case. After that the rule will be rewritten like this:

```
rule([[var(i),['->',[and,var(17),var(18)],var(impl)]],
      [var(i),var(17)]],
      [var(i),var(18)]],
     [[+,var(i),1],var(impl)])
```

Then this remodeled rule can be applied to the belief set like any other inference rule. We recognize that this tool is not very efficient, since because of its very tolerant analysis for correlation it can remodel the rule in the same way a big amount of times. Of course the identical conclusions will be cleaned to keep only one of them at the end, but the algorithm could surely be computationally improved.

Moreover, we can see here a source of complexity in the programming activity. The mathematical operations (addition, maximum, ...) and the `all(...)` function are both special tools expressed *inside* the belief patterns. It would have been convenient to store them in a same part of the programming code in order to maintain a certain organization in it. But

this is unfeasible because the first one just replaces some nested lists inside the beliefs without any external help while the second one concerns relations between a group of beliefs. For the first one the dedicated function of the prover just needs to have in argument the part of the belief where the operation appears. But for the second one it needs to have all the concerned premises of the inference rule.

3.4.2 Special devices

In a lot of inference rules the LDS approach of the memory model uses some different premises than the simple patterns of beliefs. Therefore new devices have to be implemented. Each one of them has a unique purpose and there is no general procedure to implement them. This is the need for that kind of special devices that makes the prover not totally flexible. However, those implementations are often easy for a programmer who has a sufficient knowledge of the code. In this subsection we will talk about how those devices are implemented by the prover. See 3.3.2 for the syntax of these special premises.

The observation device `spec(obs)` simply takes all the tools initially destined to the classical premises, like described in the previous subsection, and applies them to the pattern of observation that it has in argument. In this case of course the correlations are searched in the set of observations instead of the set of beliefs.

The conditional test $\alpha \in f_{formulae}(S)$ (`spec(f_forms)`) uses the correlation device applied to its argument but does not upgrade the next premises after this correlation research. The operation just returns *Yes* or *No*. For the negation use of the device (`spec(not,f_forms)`), the result is simply reversed.

The three other conditional tests ($\alpha = \beta$, $\alpha < \beta$, $\alpha \mathcal{R}_{csf} \beta$) are too obvious to be explained.

The function $\alpha \in f_{csf}(S)$ (`spec(f_csf)`) is implemented like this: the prover first applies the correlation device on each belief of the belief set in order to filter the beliefs that match the `Pattern` argument in the special premise (see in 3.3.2). Then it identifies all the positive and negative predicates of the filtered beliefs. Finally it upgrades, for each result, all the occurrences of the argument `var(alpha)` that are in the next premises.

Finally, the implementation of the very specific function $f_{min\text{-}STM\text{-}pos}(i)$ has a recursive use of same device than the one used by (`spec(f_forms)`). Keeping in memory values for *S-temp* and P that are initially the value of S and the current time respectively, the following premise is recursively simulated:

```
[spec(f_forms),[stm,0,var(i),P,var(r),var(form),var(rule)]]
```

If the test is verified, *S-temp* becomes *S-temp* -1, if not the value of *S-temp* stays the same. Then the simulated premise becomes:

```
[spec(f_forms),[stm,0,var(i),[-,P,1],var(r),var(form),var(rule)]]
```

The recursion stops when $P = 0$, in which case all the occurrences of the argument `var(Name)` (see 3.3.2) in the next premises are replaced by 0, or when *S-temp* $= 0$, in which case those occurrences are replaced by $P + 1$.

3.5 Application of the prover to LDS

In this section we will see how the inference rules and the axioms of $\mathbb{L}_{mm}$ LDS can be translated as the arguments of the prover. Those arguments are contained in the file '*arguments.txt*'. Sometimes this translating part requires a little bit of "tinkering". For example, since the prover continuously replaces the belief set instead of growing a 'static' belief set, like explained in 3.2.1, the inference rules are separated in two different sets of rules. The reason for that is that the conclusions of some rules are at the same time step than the premises and the conclusions of the other rules are at the next time step. Since the beliefs of a set at any given time step i have all the same time indicator i, then to obtain the belief set at time $i + 1$ we have to apply first the rules leading to conclusions with $i + 1$ as time indicator then apply to those new conclusions the other set of rules and finally put all the conclusions together. This is not needed with SL_7 LDS since all the inference rules take premises at time i to infer conclusions at time $i + 1$. In the following translations of the rules we will not make the distinction between the two kinds of inference rules.

For the prover, the axiom set is considered as the initial set of beliefs. So according to the purpose of translating the Clock axiom into an inference rule, the belief

```
[stm,0,1,0,0,0,pred(now,[0]),axiom]
```

is required into the set of axioms, assuming that the initial time is 0. We can now define the Clock (A1) 'inference rule':

```
rule([[stm,0,1,var(i),var(i),0,pred(now,[var(i)]),var(rule)]],
     [stm,0,1,[+,var(i),1],[+,var(i),1],0,pred(now, [[+,var(i),1]]),clock])
```

The second axiom, Obs (A2), is translated into an inference rule as well:

```
rule([[stm,0,1,var(i),var(i),0,pred(now,[var(i)]),var(rule)],
      [spec(obs),[var(i),var(form)]]],
     [qtm,0,1,var(i),0,0,var(form),obs])
```

The last axiom, LTM (A3), only concerns the first set of beliefs. So each formula specified by this axiom must be in the axiom set under the form:

```
[ltm,Alpha,Certainty,0,0,0,Beta,axiom]
```

Then comes the original inference rules. The first one is Semantic Retrieval (SR):

```
rule([[stm,0,var(c1),var(i),var(p),var(r),var(alpha),var(rule1)],
      [ltm,var(beta),var(c2),var(i),0,0,var(gamma),var(rule2)],
      [spec(r_csf),[var(alpha),var(beta)]]],
     [qtm,0,var(c2),var(i),0,0,var(gamma),sr])
```

Modus Ponens (MP):

```
rule([[stm,0,var(c1),var(i),var(p1),var(r1),var(alpha),var(rule1)],
      [stm,0,var(c2),var(i),var(p2),var(r2),
                          ['->',var(alpha),var(beta)],var(rule2)]],
     [qtm,0,[min,var(c1),var(c2)],var(i),0,0,var(beta),mp])
```

Extended Modus Ponens (EMP):

```
rule([[stm,0,var(c),var(i),var(p),var(r),
               ['->',[and|all(in,emp,free(form))],var(impl)],var(rule)],
      [stm,0,all(out,emp,free(c)),var(i),all(out,emp,free(p)),
                     all(out,emp,free(r)),all(out,emp,free(form)),
                     all(out,emp,free(rule))]],
     [qtm,0,[min,var(c)|all(in,emp,free(c))],var(i),0,0,var(impl),emp])
```

The use of the special function `all` could seem not very easy. The symbol '|' replaces the symbol ',' to stipulate that the elements that are in the same list than '|' and precede it will be integrated in the list just after '|'. So if we consider for example that `all(in,emp,free(c))` will be replaced by the list `[var(1),var(2),var(3)]`, then `[min,var(c)|all(in,emp,free(c))]` will be replaced by the list `[min,var(c),var(1),var(2),var(3)]` instead of `[min,var(c),[var(1),var(2),var(3)]]`.

Negative Introspection (NI):

```
rule([[spec(f_csf),[stm,0,var(c1),var(i1),var(i2),var(r1),
                                    var(form1),var(rule1)],var(form2)],
      [spec(not,f_forms),[stm,0,var(c2),var(i3),var(i4),var(r2),
                                              var(form2),var(rule2)]],
      [stm,0,1,var(i),var(i),0,pred(now, [var(i)]),var(rule3)]],
     [qtm,0,1,var(i),0,0,[not,pred(k,[var(i),var(form2)])],ni])
```

The third premise has been added to know which time step must be attributed to the conclusion. Indeed, the `f_form` condition is just a verifier function and the `f_csf` function just returns a positive or negative predicate. So they do not keep track of some particular time argument.

Contradiction Detection (CD1), (CD2A) and (CD2B):

```
rule([[stm,0,var(c),var(i),var(p1),var(r1),var(alpha),var(rule1)],
      [stm,0,var(c),var(i),var(p2),var(r2),[not,var(alpha)],var(rule2)]],
     [qtm,0,1,var(i),0,0,
                 pred(contra,[var(i),var(alpha),[not,var(alpha)]]),cd1])

rule([[stm,0,0,var(i),var(p1),var(r1),var(alpha),var(rule1)],
      [stm,0,1,var(i),var(p2),var(r2),[not,var(alpha)],var(rule2)]],
     [qtm,0,1,var(i),0,0,pred(loses,[var(alpha)]),cd2a])

rule([[stm,0,1,var(i),var(p1),var(r1),var(alpha),var(rule1)],
      [stm,0,0,var(i),var(p2),var(r2),[not,var(alpha)],var(rule2)]],
     [qtm,0,1,var(i),0,0,pred(loses,[[not,var(alpha)]]),cd2b])
```

Because of the fact that there are only two different certainty levels, the $c1 < c2$ and $c1 > c2$ premises were not necessary. It was easier to directly give the explicit values of the certainty levels of the rules (CD2A) and (CD2B).

Inheritance in LTM (IL):

```
rule([[ltm,var(alpha),var(c),var(i),0,0,var(beta),var(rule)]],
     [ltm,var(alpha),var(c),[+,var(i),1],0,0,var(beta),il])
```

For the next rule, Inheritance QTM → STM (IQS), the implementation takes into account that RTM is implicitly defined through STM and ITM. We need two complementary rules to specify that a certain formula is not already in RTM:

```
rule([[qtm,0,var(c1),var(i),0,0,var(alpha),var(rule1)],
      [spec(not,f_forms),[stm,0,var(c2),var(i),var(p2),var(r2),
                          var(alpha),var(rule2)]],
      [spec(not,f_forms),[stm,0,var(c3),var(i),var(p3),var(r3),
                          pred(loses,[var(alpha)]),var(rule3)]],
      [spec(not,f_forms),[itm,0,var(c4),var(i),var(p4),var(r4),
                          pred(loses,[var(alpha)]),var(rule4)]]],
     [stm,0,var(c1),[+,var(i),1],[+,var(i),1],20,var(alpha),iqs])

rule([[qtm,0,var(c1),var(i),0,0,var(alpha),var(rule1)],
      [spec(not,f_forms),[stm,0,var(c2),var(i),var(p2),var(r2),
                          var(alpha),var(rule2)]],
      [spec(not,f_forms),[stm,0,var(c3),var(i),var(p3),var(r3),
                          pred(loses,[var(alpha)]),var(rule3)]],
      [spec(f_forms),[itm,0,var(c4),var(i),var(p4),0,
                      pred(loses,[var(alpha)]),var(rule4)]]],
     [stm,0,var(c1),[+,var(i),1],[+,var(i),1],20,var(alpha),iqs])
```

In other words, if we look at the fourth premise in each rule, it means that the formula *loses*(α) is not in ITM, or is in ITM but the value of the argument R is 0.

The two next rules of inheritance, that are very similar, are the less obvious to implement because we get rid of the set delimiter $S_{new\text{-}STM}(i+1)$, like we said in Section 3.3.2, and we need to implement three complementary rules each time to substitute the $\vee$ operator.

Inheritance in STM (IS):

```
rule([[stm,0,var(c1),var(i),var(p1),var(r1),var(alpha),var(rule1)],
      [spec(f_min_stm_pos),var(i),20,var(pos)],
      [spec(<),[-,var(pos),1],var(p1)],
      [spec(not,f_forms),[stm,0,var(c2),var(i),var(p2),var(r2),
                          pred(contra,[[-,var(i),1],var(alpha),var(beta1)]),
                          var(rule2)]],
      [spec(not,f_forms),[stm,0,var(c3),var(i),var(p3),var(r3),
                          pred(contra,[[-,var(i),1],var(beta2),var(alpha)]),
                          var(rule3)]],
      [spec(not,f_forms),[stm,0,var(c4),var(i),var(p4),var(r4),
                          pred(loses,[var(alpha)]),var(rule4)]],
      [spec(not,=),[var(alpha),pred(now,[var(i)])]],
      [spec(not,=),[var(alpha),[not,pred(k,[[-,var(i),1],var(beta3)])]]],
      [spec(not,=),[var(alpha),
                    pred(contra,[[-,var(i),1],var(beta4),var(beta5)])]]],
     [stm,0,var(c1),[+,var(i),1],var(p1),var(r1),var(alpha),is])

rule([[stm,0,var(c1),var(i),var(p),var(r),
             [not,pred(k,[[-,var(i),1],var(alpha)])],var(rule1)],
      [spec(f_min_stm_pos),var(i),20,var(pos)],
      [spec(<),[-,var(pos),1],var(p)],
      [spec(not,f_forms),[qtm,0,var(c2),var(i),0,0,
                          [not,pred(k,[var(i),var(alpha)])],
                          var(rule2)]]],
     [stm,0,var(c1),[+,var(i),1],var(p),var(r),
                    [not,pred(k,[[-,var(i),1],var(alpha)])],is])

rule([[stm,0,var(c1),var(i),var(p),var(r),
             pred(contra,[[-,var(i),1],var(alpha),var(beta)]),var(rule1)],
      [spec(f_min_stm_pos),var(i),20,var(pos)],
      [spec(<),[-,var(pos),1],var(p)],
      [spec(not,f_forms),[qtm,0,var(c2),var(i),0,0,
                          pred(contra,[var(i),var(alpha),var(beta)]),
                          var(rule2)]]],
```

```
[stm,0,var(c1),[+,var(i),1],var(p),var(r),
                pred(contra,[[-,var(i),1],var(alpha),var(beta)]),is])
```

Inheritance STM → ITM (ISI): This rule is the same than (IS) where, in each one of the three complementary rules, `[spec(<),[-,var(pos),1],var(Name)]` is replaced by `[spec(not,<),[-,var(pos),1],var(Name)]` and `stm` in the conclusion is replaced by `itm`.

Inheritance in ITM (II):

```
rule([[itm,0,var(c),var(i),var(p),var(r),var(alpha),var(rule)]],
     [itm,0,var(c),[+,var(i),1],var(p),[max,0,[-,var(r),1]],var(alpha),ii])
```

This implementation has been tested on Mikael Asker's example. We obtained the expected results, see Appendix C.

Chapter 4

Conclusion

4.1 Active logics and LDS

Active logics seem like a good choice in terms of efficient reasoning. The reasoning is situated in time, which solves the omniscience problem and allows the agent to reason about the reasoning process itself. Active logics also allow non-monotonic reasoning and contradiction handling. Everything seems to indicate that active logics are a good basis to formalize the memory model based on cognitive psychology defined in [JDP]. The first attempt to do it were the step logics, which are an oversimplification of the memory model. In particular, the focus of attention was not taken into account, leading to computational issues for large problems.

The purpose of Mikael Asker's thesis was to extend this formalization using Gabbay's *Labelled Deductive Systems* (LDS). The inference rules used in step logics were kept and extended with labels to respect the different parts of the memory and how the information travels between them.

The LDS approach seems to formalize well the memory model but has only been tested on one simple problem in which the use of focus of attention is not needed. It would be interesting to test the approach on realistically large problems, to see if the concept of focus of attention really works like expected. The memory model LDS has already been implemented once, by Sonia Fabre in Common Lisp (see [Fab04]). But this prover was non-extensible, the inference function could not be modified.

4.2 The automatic theorem prover

The purpose of this work was to implement a theorem prover able to implement not only the memory model LDS, but any kind of LDS, allowing a control over the inference function as flexible as possible. The result is quite satisfactory, and a wide range of different inference rules can be encoded. But still there are some limitations due to the fact that some inference rules

need special functions. Sometimes a little reprogramming of the prover is inevitable. The user also needs to be careful when designing a system in which complex use of the operations "(min,max,+,-)" is made. For now, the prover has only been tested, with success, on the Three-wise-men Problem (via step logics) and on the Tweety Problem (via LDS). The descriptions of these problems as well as the results obtained are presented in Appendixes B and C, respectively.

We are aware that the biggest weakness of our prover resides in its lack of efficiency when it comes to larger problems[1], so the notion of focus of attention is very important.

4.3 Future work

Concerning the software itself, the prover is not perfect and some of its parts can be improved:

- The biggest source of errors is clearly the nested operations. All the nested arguments must be numbers and not unknown variables in order to expect the prover to resolve it. A good improvement could be to implement a device able to simplify those operations when some of the arguments are still variables. Another way to increase the robustness would be to improve the correlation device so that it could find the correlation between two equivalent nested operations but where the arguments are placed differently.

- The mechanism looking for correlation between beliefs and premises containing the element `all(in,...,...)` are too tolerant. So this part of the prover is inefficient. We still do not see how to decrease this excessive tolerance without increasing the risk of errors.

- When the value of a specific element of a premise does not matter, we designate it with a variable unused anywhere else. We should introduce a new symbol especially for those non-important elements, for clarity and for avoiding useless upgrading attempts in the inference rules.

We did not try to determine how powerful the prover is:

- It could be useful to test it on some famous problems, like we did with the Three-wise-men Problem, to see if it leads to the right conclusions.

- This prover should be compared with the others present in the market on some famous probles, in order to compare their flexibility as well as their robustness.

[1] Note that we first made a non-flexible prover for $\mathbb{L}_{mm}$ LDS, and it was much faster than this one.

Finally, if the prover is determined to work well on larger problems, we could use it to serve different formalisms like the memory model LDS[2], as it was its initial purpose.

[2]Considering the lack of efficiency of our prover, the implementation of the focus of attention is primary for modelling the realistically-large problems.

Bibliography

[Ask03] Mikael Asker. Logical reasoning with temporal constraints. Master's thesis, Department of Computer Science, Lund University, August 2003.

[ED88] Jennifer J. Elgot-Drapkin. *Step-logic: Reasoning Situated in Time*. PhD thesis, University of Maryland, 1988.

[Fab04] Sonia Fabre. Automatic theorem proving in labelled deduction systems. Master's thesis, Department of Computer Science, Lund University, 2004.

[Gab96] Dov M. Gabbay. Labelled deductive systems, volume 1. *Oxford University Press*, 1996.

[JDP] Michael Miller Jennifer Drapkin and Donal Perlis. A memory model for real-time commonsense reasoning. Technical report, Systems Research Center, University of Maryland, College Park, Maryland.

[JEDPa] Michael Miller Jennifer Elgot-Drapkin and Donal Perlis. Reasoning situated in time i: Basic concepts. Technical report, Department of Computer Science, University of Maryland, College Park, Maryland.

[JEDPb] Michael Miller Madhura Nirkhe Jennifer Elgot-Drapkin, Sarit Kraus and Donal Perlis. Active logics: A unified formal approach to episodic reasoning. Technical report, Department of Computer Science, University of Maryland, College Park, Maryland.

Appendix A

Documentation of the code

All the program holds in one single file, written in Prolog: "ActiveLogics.pl".

All the concepts of beliefs, inference rules, premises, special functions... used in the following specifications respect the descriptions given in 3.2 and 3.3.

apply_all_rules(+Rules,+Beliefs,+Obs,−Concls) is true if Rules is a set of inference rules, Beliefs is a set of beliefs, Obs is the set of observations that can be used by the inference rules if necessary, and Concls is the set of all the conclusions (without doubles) that can be drawn by the application of all the inference rules present in Rules to the belief set Beliefs.

all_groups(+Beliefs1,+Beliefs2,+Obs,+N,+Rule,−Group) is true if:

- Rule is an inference rule.
- Obs is the set of observation that can be used by the inference rule Rule if necessary.
- N is the highest number used as a name of variable (var(N)) in Rule and in the belief sets Beliefs1 and Beliefs2.
- Group is the set of all the different good substitutions of beliefs that can be done to the premises of Rule, with each time the inferred conclusion. So the elements of Group are under the form: rule([Belief1,Belief2,...],InferredConcl). The beliefs substituting the first premise of Rule come from Beliefs1, and all the beliefs substituting the other premises come from Beliefs2.

correlation(+X,+Y,−Ls) is true if the two formulae X and Y are potentially correlated[1] and if the list Ls regroups all the values attributed to the variables following this correlation (under the form corr(var(Name),Value)).

multi_correl(+Xs,+Ys,−Ls) is true if all the respective elements of the lists of formulae Xs and Ys are potentially correlated[1] and if the list

Ls regroups all the values attributed to the variables of those formulae following those correlations (under the form corr(var(Name),Value)).

replace(+X,+Vars,−Y) is true if Vars is a list of distinct variables with associated values, under the form corr(var(Name),Value), and if Y is a formula equivalent to the formula X in which all the occurrences of any variable present in Vars is replaced by its associated value and where the simple mathematical operations (addition, substraction, minimum and maximum) are solved when possible (ie when all their arguments can be reduced to numbers).

multi_replace(+Xs,+Vars,−Ys) is true if Vars is a list of distinct variables with associated values, under the form corr(var(Name),Value), and if Ys is a list of formulae equivalent to the list of formulae Xs in which all the occurrences of any variable present in Vars is replaced by its associated value and where the simple mathematical operations (addition, substraction, minimum and maximum) are solved when possible (ie when all their arguments can be reduced to numbers).

rule_replace(+Rule1,+Vars,−Rule2) is true if Vars is a list of distinct variables with associated values, under the form corr(var(Name),Value), and if Rule2 is an inference rule equivalent to the inference rule Rule1 in which all the occurrences of any variable present in Vars is replaced by its associated value and where the simple mathematical operations (addition, substraction, minimum and maximum) are solved when possible (ie when all their arguments can be reduced to numbers).

map_prem(+Prem,+Rules1,−Rules2) is true if Rules2 is a list of inference rules corresponding to the list of inference rules Rules1 in which the element Prem has been added as the first premise of each rule.

addition(+Xs,−N) is true if Xs is a list of numbers and if N is a number corresponding to the addition of all the elements of Xs.

[1]Two formulae are designated for a potential correlation if they fulfill one of the following conditions:

- They are exactly the same.
- At least one of them is a variable. In this case the prover will keep track of the other formula (actually it could also be a variable) as the value of the variable.
- They are both formulae in which all the corresponding sub-formulae fulfill one of these conditions.
- One of them is a simple operation of nested additions and substractions and containing at least one variable. To keep track of a value for this variable, the prover will reverse the operation between the two elements.

substraction(+Xs,−N) is true if Xs is a list of numbers and if N is a number corresponding to the substraction from the first element of Xs of all the other elements of Xs.

minimum(+Xs,−N) is true if Xs is a list of numbers and if N is a number corresponding to the minimum of all the elements of Xs.

maximum(+Xs,−N) is true if Xs is a list of numbers and if N is a number corresponding to the maximum of all the elements of Xs.

not_numbers(+Xs) is true if the list Xs contains at least one element which is not a number.

var_max(+X,−N) is true if X is a formula and if N is the highest number corresponding to the name of a variable in X. If there is no element var(Name) in X for which Name is a number, then N = 0.

multi_var_max(+Xs,−N) is true if Xs is a list of formulae and if N is the highest number corresponding to the name of a variable in Xs. If there is no element var(Name) in Xs for which Name is a number, then N = 0.

rule_var_max(+Rule,−N) is true if Rule is an inference rule and if N is the highest number corresponding to the name of a variable in Rule. If there is no element var(Name) in Rule for which Name is a number, then N = 0.

uni_vars(+X,−Ls) is true if X is a formula in prenex form and if Ls is a list containing all the universal variables (in the form any(Name)) of X.

multi_uni_vars(+Xs,−Ls) is true if Xs is a list of formulae in prenex form and if Ls is a list containing all the universal variables (in the form any(Name)) of Xs.

del_doubles(+Xs,−Ys) is true if the list Ys corresponds to the list Xs in which all the repetitions have been removed.

repl_one_uni(+X,+any(Name1),+var(Name2),−Y) is true if the formula Y corresponds to the formula X in which all the occurrences of any(Name1) have been replaced by var(Name2).

multi_repl_one_uni(+Xs,+any(Name1),+var(Name2),−Ys) is true if the list of formulae Ys corresponds to the list of formulae Xs in which all the occurrences of any(Name1) have been replaced by var(Name2).

multi_repl_all_uni(+Xs,+UniVars,+N,−Ys) is true if UniVars is a list of universal variables (in the form any(Name)) and if the list of formulae Ys corresponds to the list of formulae Xs after the application of this treatment:

- All the occurrences of the first element of UniVars is replaced by var(N+1).
- All the occurrences of the second element of UniVars is replaced by var(N+2).
- All the occurrences of the third element of UniVars is replaced by var(N+3).
- ... And so on until UniVars is empty.

uni_belief_set(+Beliefs1,+Rule,−Beliefs2,−N2) is true if:

- Rule is an inference rule.
- N1 is the highest number corresponding to the name of a variable in Rule.
- The set of beliefs Beliefs2 corresponds to the set of beliefs Beliefs1 where all the universal variables (in the form any(Name)) have been replaced by new variables var(Number) with Number = N1+1, N1+2, ..., N2 so that two independent universal variables will be replaced by distinct variables.

var_in_formula(+X,−Ls) is true if Ls is a list regrouping all the variables (in the form var(Name)) present in the formula X.

var_in_belief(+Xs,−Ls) is true if Ls is a list regrouping all the variables (in the form var(Name)) present in the list of formulae Xs.

rename_vars(+N1,+Vars,−Corrs,−N2) is true if Vars is a list of variables, N1 and N2 are numbers and the list Corrs corresponds to the list Vars after the application of this treatment:

- The first element var(Name1) is replaced by corr(var(Name1),var(N1+1)).
- The second element var(Name2) is replaced by corr(var(Name2),var(N1+2)).
- ...
- The last element var(NameLast) is replaced by corr(var(NameLast),var(N2)).

upgrade_vars(+N1,+Belief,+Beliefs1,−Beliefs2,−N2) is true if the set of beliefs Beliefs2 corresponds to the set of beliefs Beliefs1 in which all the variables (in the form var(Name)) present in the belief Belief have been replaced by new variables var(Number) with Number = N1+1, N1+2, ..., N2 so that two independent variables will be replaced by distinct variables.

is_in(+X,+Belief) is true if at least one occurrence of the element X is present in the belief Belief.

invert_op(+X,+Xs,+Ys,−Zs) is true if:

- Xs ans Ys are mathematical operations using only nested additions and substractions.
- X is an element of Xs.
- Following the hypothesis that Xs = Ys, then X = Zs.

belief_flex_corr(+Name,+Xs,+Ys,−L) is true if:

- Xs is a belief containing somewhere in at least one of its formulae the element all(in,Name,X), where X is a formula.
- Ys is a belief containing a list of elements Ls respectively at the exact same position than (one of) the element(s) all(in,Name,X) in Xs.
- Each element of the list Ls can be correlated with the formula X.
- L is the length of the list Ls.

flex_vars(+Name,+N1,+L,+Corrs1,+Belief,−N2,−Corrs2) is true if:

- Corrs1 is a list of elements that are each in the form corr(free(Y),[var(Z+1),var(Z+2),...,var(Z+L)]).
- Corrs2 is a list corresponding to Corrs1 where a new element is added for each element free(...) not already present in the elements of Corrs2 and contained in the formula X of any element all(_,Name,X) itself contained in the belief Belief.
- All those added elements have together the following form:
  corr(free(Y1),[var(N1+1)...var(N1+L)]),
  corr(free(Y2),[var(N1+L+1)...var(N1+2L)]),
  ...
  corr(free(Ylast),[var(N2-L+1)...var(N2)]).

repl_flex(+Corrs,+X,−Y) is true if Corrs is a list of distinct free elements with associated variables, under the form corr(free(Name1),var(Name2)), and if Y is a formula equivalent to the formula X in which all the occurrences of any free element present in Corrs is replaced by its associated variable.

repl_flex_in(+all(S,Name,_),+L,+Corrs,+Form1,−Form2) is true if:

- S = in.
- Corrs is a list of elements that are each in the form corr(free(Y),[var(Z+1),var(Z+2),...,var(Z+L)]).

- Form2 is a formula corresponding to the formula Form1 in which all the occurrences of all(in,Name,Form3) are replaced by a list of L elements where each i^{th} element is equivalent to the formula Form3 in which all the free(...) elements have been replaced by the i^{th} variable of the corresponding element in Corrs.

or if:

- S = out.
- Corrs is a list of elements that are each in the form corr(free(Y),var(Z)).
- Form2 is a formula corresponding to the formula Form1 in which all the occurrences of all(out,Name,Form3) are replaced by the formula Form3 in which all the free(...) elements have been replaced by the variable of the corresponding element in Corrs.

repl_flex_out(+all(_,Name,_),+L,+Corrs,+Prem,−Prems) is true if:

- Corrs is a list of elements that are each in the form corr(free(Y),[var(Z+1),var(Z+2),...,var(Z+L)]).
- Prems is a list of L premises, each i^{th} premise corresponds to the premise Prem in which all the occurrences of all(out,Name,Form) have been replaced by the formula Form where all the free(...) elements have been replaced by the i^{th} variable of the corresponding element in Corrs.

repl_all_flex(+all(in,Name,_),+N1,+L,+Corrs,+Rule1,−Rule2,−N2) is true if Rule2 is an inference rule corresponding to the inference rule Rule1 in which:

- Corrs is a list of elements that each have the form corr(free(Y),[var(Z+1),var(Z+2),...,var(Z+L)]).
- The list Corrs1 is equivalent to Corrs completed by all the elements corresponding to the free(...) elements that are in Rule1 but not censed in Corrs. L new variables are attached to each added free(...) element. The new variables have the form var(N1+1), var(N1+2), ..., var(N2+L) such that distinct free(...) elements have distinct corresponding variables.
- All the occurrences of all(in,Name,Form1) in the premises and the conclusion are replaced by a list of L elements where each i^{th} element is equivalent to the formula Form1 in which all the free(...) elements have been replaced by the i^{th} variable of the corresponding element in Corrs2.
- All the premises having some occurrences of all(out,Name,Form2) are replaced by a list of L premises where each i^{th} premise keeps

the initial form, except for the all(out,Name,Form2) element that is replaced by the formula Form2 in which all the free(...) elements have been replaced by the i^{th} variable of the corresponding element in Corrs2.

clean_vars(+Corrs1,−Corrs2) is true if Corrs1 is a list of elements that each have the form corr(var(X),Formula), in which all the variables explicitly or implicitly defined[1] more than once have compatible definitions[2], and Corrs2 is a list corresponding to Corrs1 after a "cleaning" step, which means:

- All the implicit definitions become explicit.
- All the definitions containing a variable that is defined somewhere else have this variable replaced by its definition.
- All the formulae defining a same variable are combined to form one only definition, since they are compatible by hypothesis.
- All the formulae defining a variable and containing this same variable are simply deleted.

var_reduce(+X,+Y,−Z,−Corrs) is true if:

- X and Y are compatible[1] formulae.
- The list Corrs contains all the implicit[2] definitions following the fact that X and Y are compatible.
- Z is equivalent to the combination of X and Y, compatible with both of them, so that the subformulae are privileged compared to the variables.

condition(+Cond,+Beliefs,+Args) is true if:

- Cond = 'r_csf', Args = [Alpha,Beta] with Alpha and Beta being formulae, and Alpha $\mathcal{R}_{csf}$ Beta.
- Cond = '=', Args = [Alpha,Beta] with Alpha and Beta being formulae, and Alpha = Beta.
- Cond = '<', Args = [Alpha,Beta] with Alpha and Beta being formulae, and Alpha < Beta.

[1] An explicit definition is a formula Formula explicitly associated to a variable var(X) by the element corr(var(X),Formula). An implicit definition appears when, considering that we have two compatible formulae defining a same variable, there is a variable in one of them corresponding to a subformula in the other. Then the second one defines the first one.

[2] Two formulae are compatible if they are exactly the same except that a subformula of one of them can be replaced in the other by a variable (under the form var(...)) and vice versa.

- Cond = 'f_forms', Beliefs is a set of beliefs and Args is a premise that can be correlated with at least one of the beliefs of Beliefs.

belief_filter(+Beliefs1,+Belief,−Beliefs2) is true if Beliefs1 is a list of beliefs, Belief is a belief and Beliefs2 is the list of all the beliefs of Beliefs1 than can be correlated with Belief.

all_csf(+Beliefs,−Ls) is true if Beliefs is a list of beliefs and Ls is the list of all the positive and negative predicates (pred(...,...) or [not,pred(...,...)]) that are in Beliefs and that are not 'k', 'contra' or 'loses' predicates.

uni_vars_back(+X,−Vars) is true if Vars is a list containing all the variables (under the form var(...)) present in the formula X.

multi_uni_vars_back(+Xs,−Vars) is true if Vars is a list containing all the variables (under the form var(...)) present in the list of formulae Xs.

any2var(+N,+Vars,−Corrs) is true if:

- N is an integer.
- Vars is a list of L variables [var(X1), var(X2), ..., var(XL)].
- Corrs is a list associating universal variables to each variable present in Vars: [corr(var(X1),any(N+1)), corr(var(X2),any(N+2)), ..., corr(var(XL),any(N+L))].

repl_concl(+Concl1,−Concl2) is true if the belief Concl2 corresponds to the belief Concl1 in which all the variable (under the form var(...)) have been replaced by universal variables (under the form any(...)). Two occurrences of the same variable are replaced by the same universal variable. Two different variables are replaced by different universal variables. For the rest, the name of the universal variables is randomly chosen.

repl_all_concl(+Rules,−Concls) is true if Rules is a set of inference rules and if Concls is the set of all the conclusions of those inference rules, after replacing all the variables (under the form var(...)) by universal variables (under the form any(...)). Two occurrences of the same variable in the same conclusion are replaced by the same universal variable. Two different variables in the same conclusion are replaced by different universal variables. For the rest, the name of the universal variables is randomly chosen.

Appendix B

Step logic: The Wise-men problem(s)

B.1 Statement of the problem

This problem comes from Jennifer Elgot-Drapkin's PhD thesis, see [ED88] for more details.

A king wishes to know whether his three advisors are as wise as they claim to be. Three chairs are lined up, all facing the same direction, with one behind the other. The wise men are instructed to sit down. The wise man in the back (wise man #3) can see the backs of the other two men. The man in the middle (wise man #2) can only see the one wise man in front of him (wise man #1); and the wise man in front (wise man #1) can see neither wise man #3 nor wise man #2. The king informs the wise men that he has three cards, all of which are either black or white, at least one of which is white. He places one card, face up, behind each of the three wise men, explaining that each wise man must determine the color of his own card. Each wise man must announce the color of his own card as soon as he knows what it is. The room is silent, then, after several minutes, wise man #1 says "My card is white!".

We assume in this puzzle that the wise men do not lie, that they all have the same reasoning capabilities, and that they can all think at the same speed.

B.2 The Two-wise-men Problem

B.2.1 Statement of the problem

In this puzzle the king has just two wise men and two cards, at least one of which is white. The reasoning involved in this version of the puzzle is much simpler than in the three-wise-men version. Wise man #2 can see the color

of wise man #1's card. If it were black, then wise man #2 would know, since there is at least one white card, that his card was white. Wise man #1 knows this. Wise man #2 says nothing. Therefore, wise man #1's card must not be black, but rather white.

The problem is modelled from wise man #1's point of view. The inference function used to model this problem is defined in Figure B.1 below, in which:

- $K_j(i, x)$ is intended to mean "wise man j knows x at step i."
- $U(i, x)$ expresses the fact that an utterance of x is made at step i.
- $s(i)$ is the successor function (where $s^k(0)$ is used as an abbreviation for $\underbrace{s(s(...(s(}_{k}0))...)))$.

Rule 1 :	$\dfrac{i : \ldots}{i+1 : \ldots, \alpha}$	if $\alpha \in OBS(i+1)$
Rule 2 :	$\dfrac{i : \ldots, \alpha, (\alpha \rightarrow \beta)}{i+1 : \ldots, \beta}$	Modus ponens
Rule 3 :	$\dfrac{i : \ldots, P_1\overline{a}, \ldots, P_n\overline{a}, (\forall \overline{x})[(P_1\overline{x} \wedge \ldots \wedge P_n\overline{x}) \rightarrow Q\overline{x}]}{i+1 : \ldots, Q\overline{a}}$	Extended modus ponens
Rule 4 :	$\dfrac{i : \ldots, \neg\beta, (\alpha \rightarrow \beta)}{i+1 : \ldots, \neg\alpha}$	
Rule 5 :	$\dfrac{i : \ldots}{i+1 : \ldots, \neg K_1(s^i(0), U(s^{i-1}(0), W_2))}$	if $U(s^{i-1}(0), W_2) \notin \vdash_i$, $i > 1$
Rule 6 :	$\dfrac{i : \ldots, \alpha}{i+1 : \ldots, \alpha}$	Inheritance

Figure B.1: INF_{W_2} for the Two-wise-men Problem

The observation function, defined in Figure B.2, contains all the axioms that wise man #1 needs to solve the problem. W_i and B_i expresses the facts that i's card is white, and i's card is black, respectively.

The solution is given in Figures B.3 and B.4. Only the relevant beliefs for the next steps appear on each step of these figures.

For more explanations about the observation function and the different steps of the solution, see [ED88].

B.2.2 Implementation

We used our prover to model this problem with its step logic approach. We just made some imperceptible changes compared to the description given above:

- According to the nature of the "observations" in the observation function, we chose to put them in the set of axioms instead. Indeed, those formulae appears more like initial knowledge than new observations.
- We dropped the concept of successor function, by considering $s(0) = 1$ and $s(i) = i + 1$.

$$OBS(i) = \begin{cases} \left\{ \begin{array}{l} (\forall i)(\forall x)(\forall y)[K_2(i, x \rightarrow y) \rightarrow (K_2(i, x) \rightarrow K_2(s(i), y))] \\ K_2(s(0), B_1 \rightarrow W_2) \\ (B_1 \rightarrow K_2(s(0), B_1)) \\ (\neg B_1 \rightarrow W_1) \\ (\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)] \\ (\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)] \end{array} \right\} & \text{if } i = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

Figure B.2: OBS_{W_2} for the Two-wise-men Problem

We can show now how the arguments of the prover must look to model this problem:

```
Rules1 = [
  rule([[var(i),var(form)],
        [spec(obs),[var(i),var(alpha)]]],
       [[+,var(i),1],var(alpha)]),
  rule([[var(i),var(alpha)],
        [var(i),['->',var(alpha),var(beta)]]],
       [[+,var(i),1],var(beta)]),
  rule([[var(i),['->',[and|all(in,emp,free(p))],var(q)]],
        [var(i),all(out,emp,free(p))]],
       [[+,var(i),1],var(q)]),
  rule([[var(i),[not,var(beta)]],
        [var(i),['->',var(alpha),var(beta)]]],
       [[+,var(i),1],[not,var(alpha)]]),
  rule([[var(i),var(formula)],
        [spec(not,f_forms),[var(i),pred(u,[[-,var(i),1],pred(w,[2])])]],
        [spec('<'),[1,var(i)]]],
       [[+,var(i),1],[not,pred(k,[1,var(i),
                                  pred(u,[[-,var(i),1],pred(w,[2])])])]]),
  rule([[var(i),var(alpha)]],
       [[+,var(i),1],var(alpha)])
],

Rules2 = [],

Axioms = [
```

```
  [1,['->',pred(k,[2,any(i),['->',any(x),any(y)]]),
           ['->',pred(k,[2,any(i),any(x)]),
                 pred(k,[2,[+,any(i),1],any(y)])]]],
  [1,pred(k,[2,1,['->',pred(b,[1]),pred(w,[2])]])],
  [1,['->',pred(b,[1]),pred(k,[2,1,pred(b,[1])])]],
  [1,['->',[not,pred(b,[1])],pred(w,[1])]],
  [1,['->',[not,pred(u,[[+,any(i),1],pred(w,[2])])],
           [not,pred(k,[2,any(i),pred(w,[2])])]]],
  [1,['->',[not,pred(k,[1,[+,any(i),1],pred(u,[any(i),pred(w,[2])])])],
           [not,pred(u,[any(i),pred(w,[2])])]]]
],

Obs = [].
```

The prover works as expected, and solves this simple problem in less than 2 seconds.

B.3 The Three-wise-men Problem

B.3.1 Statement of the problem

Now that the Two-wise-men Problem has been modelled, the mechanics of that problem can be brought to bear on the three-wise-men version.

We can postulate that the following reasoning took place. Each wise man knows there is at least one white card. If the cards of wise man #2 and wise man #1 were black, then wise man #3 would have been able to announce immediately that his card was white. They all realize this. Since wise man #3 kept silent, either wise man #2's card is white, or wise man #1's is. At this point wise man #2 would be able to determine, if wise man #1's were black, that his card was white. They all realize this. Since wise man #2 also remains silent, wise man #1 knows his card must be white.

The set of observations (that we will still use as the set of axioms in the implementation) is defined in Figure B.5.

INF_{W_3} is the same as INF_{W_2} augmented with the additional rules in Figure B.6 (where Rule 8 replaces Rule 5).

For more explanations about the observation function and the different steps of the solution, see [ED88].

B.3.2 Implementation

For the prover, a universal variable means something like "valid for every existing value of the variable". So if the formula $(\forall \bar{x})(P\bar{x} \rightarrow Q\bar{x})$ is present in the belief set, that notably means for the prover that the formula $(P\bar{a} \rightarrow Q\bar{a}$ is present in the belief set. Then Rule 7 is covered by Rule 4. Furthermore, Rule 9 is completely useless for the prover.

0:		$\emptyset$	
1:	(a)	$(\forall i)(\forall x)(\forall y)[K_2(i, x \rightarrow y) \rightarrow (K_2(i, x) \rightarrow K_2(s(i), y))]$	(R1)
	(b)	$K_2(s(0), B_1 \rightarrow W_2)$	(R1)
	(c)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R1)
	(d)	$(\neg B_1 \rightarrow W_1)$	(R1)
	(e)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R1)
	(f)	$(\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)]$	(R1)
2:	(a)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R3,1a,1b)
	(b)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,1c)
	(c)	$(\neg B_1 \rightarrow W_1)$	(R6,1d)
	(d)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R6,1e)
	(e)	$(\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)]$	(R6,1f)
3:	(a)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R6,2a)
	(b)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,2b)
	(c)	$(\neg B_1 \rightarrow W_1)$	(R6,2c)
	(d)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R6,2d)
	(e)	$(\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)]$	(R6,2e)
4:	(a)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R6,3a)
	(b)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,3b)
	(c)	$(\neg B_1 \rightarrow W_1)$	(R6,3c)
	(d)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R6,3d)
	(e)	$(\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)]$	(R6,3e)

Figure B.3: Solution to the Two-wise-men Problem - Part 1

5:	(a)	$\neg K_1(s^4(0), U(s^3(0), W_2))$	(R5)
	(b)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R6,4a)
	(c)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,4b)
	(d)	$(\neg B_1 \rightarrow W_1)$	(R6,4c)
	(e)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R6,4d)
	(f)	$(\forall i)[\neg K_1(s(i), U(i, W_2)) \rightarrow \neg U(i, W_2)]$	(R6,4e)
6:	(a)	$\neg U(s^3(0), W_2)$	(R3,5a,5f)
	(b)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R6,5b)
	(c)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,5c)
	(d)	$(\neg B_1 \rightarrow W_1)$	(R6,5d)
	(e)	$(\forall i)[\neg U(s(i), W_2) \rightarrow \neg K_2(i, W_2)]$	(R6,5e)
7:	(a)	$\neg K_2(s^2(0), W_2)$	(R3,6a,6e)
	(b)	$(K_2(s(0), B_1) \rightarrow K_2(s^2(0), W_2))$	(R6,6b)
	(c)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,6c)
	(d)	$(\neg B_1 \rightarrow W_1)$	(R6,6d)
8:	(a)	$\neg K_2(s(0), B_1)$	(R4,7a,7b)
	(b)	$(B_1 \rightarrow K_2(s(0), B_1))$	(R6,7c)
	(c)	$(\neg B_1 \rightarrow W_1)$	(R6,7d)
9:	(a)	$\neg B_1$	(R4,8a,8b)
	(b)	$(\neg B_1 \rightarrow W_1)$	(R6,8c)
10:		W_1	(R2,9a,9b)

Figure B.4: Solution to the Two-wise-men Problem - Part 2

$$
OBS_{W_3}(i) = \begin{cases}
\left\{\begin{array}{l}
(\forall j)K_2(j,(\forall i)(\forall x)(\forall y)[K_3(i,x\rightarrow y)\rightarrow \\
\qquad (K_3(i,x)\rightarrow K_3(s(i),y))]) \\
(\forall j)K_2(j,K_3(s(0),(B_1\wedge B_2)\rightarrow W_3)) \\
(\forall j)K_2(j,(B_1\wedge B_2)\rightarrow K_3(s(0),B_1\wedge B_2)) \\
(\forall j)K_2(j,\neg(B_1\wedge B_2)\rightarrow (B_1\rightarrow W_2)) \\
(\forall j)K_2(j,(\forall i)[\neg U(s(i),W_3)\rightarrow \neg K_3(i,W_3)]) \\
(\forall i)(\forall x)[\neg K_1(s(i),U(i,x))\rightarrow \neg U(i,x)] \\
(\forall i)[\neg U(i,W_3)\rightarrow K_2(s(i),\neg U(i,W_3))] \\
(\forall i)(\forall x)(\forall y)[K_2(i,x\rightarrow y)\rightarrow (K_2(i,x)\rightarrow K_2(s(i),y))] \\
(\forall i)(\forall x)(\forall x')(\forall y)(\forall y') \\
\quad [(K_2(i,\neg(x\wedge x')\rightarrow (y\wedge y'))\wedge K_2(i,\neg(x\wedge x')))\rightarrow \\
\quad K_2(s(i),y\wedge y')] \\
(\forall j)(\forall k)(\forall z)(\forall z')(\forall w) \\
\quad [(K_2(j,(\forall i)(\forall x)(\forall y)[K_3(i,x\rightarrow y)\rightarrow \\
\qquad (K_3(i,x)\rightarrow K_3(s(i),y))]\wedge \\
\quad K_2(j,K_3(k,(z\wedge z')\rightarrow w)))\rightarrow \\
\quad K_2(s(j),K_3(k,z\wedge z')\rightarrow K_3(s(k),w))] \\
(\forall j)(\forall k) \\
\quad [(K_2(j,(\forall i)[\neg U(s(i),W_3)\rightarrow \neg K_3(i,W_3)])\wedge \\
\quad K_2(j,\neg U(s(k),W_3)))\rightarrow \\
\quad K_2(s(j),\neg K_3(k,W_3))] \\
(\forall i)(\forall x)(\forall y)[(K_2(i,x\rightarrow y)\wedge K_2(i,\neg y))\rightarrow K_2(s(i),\neg x)] \\
(\forall i)(\forall x)(\forall x')(\forall y) \\
\quad [(K_2(i,(x\wedge x')\rightarrow y)\wedge K_2(i,\neg y))\rightarrow K_2(s(i),\neg(x\wedge x'))] \\
(\forall i)[B_1\rightarrow K_2(i,B_1)] \\
(\neg B_1\rightarrow W_1) \\
(\forall i)[\neg U(s(i),W_2)\rightarrow \neg K_2(i,W_2)]
\end{array}\right. & \text{if } i=1 \\
\emptyset & \text{otherwise}
\end{cases}
$$

Figure B.5: OBS_{W_3} for the Three-wise-men Problem

Rule 7 :
$$\frac{i:\ldots,\neg Q\overline{a},(\forall\overline{x})(P\overline{x}\rightarrow Q\overline{x})}{i+1:\ldots,\neg P\overline{a}}$$

Rule 8 :
$$\frac{i:\ldots}{i+1:\ldots,\neg K_1(s^i(0),U(s^{i-1}(0),W_j))} \quad \text{if } U(s^{i-1}(0),W_j)\notin\vdash_i,\ j=2,3,i>1$$

Rule 9 :
$$\frac{i:\ldots,(\forall j)K_2(j,\alpha)}{i+1:\ldots,K_2(s^i(0),\alpha)}$$

Figure B.6: Completion of INF_{W_2} to form INF_{W_3} for the Three-wise-men Problem

So we just add the next rule in order to extend Rule 5 to Rule 8:

```
rule([[var(i),var(formula)],
      [spec(not,f_forms),[var(i),pred(u,[[-,var(i),1],pred(w,[3])])]],
      [spec('<'),[1,var(i)]]],
     [[+,var(i),1],[not,pred(k,[1,var(i),pred(u,[[-,var(i),1],pred(w,[3])])])]])
```

This time we will skip the writing of OBS_{W_3} in the prover's arguments because it is pointless for the reader.

Sadly, when we run the prover, it gets "Out of global stack" after the 9th step of deduction, even if it worked well until this step. This is due to the inheritance rule (Rule 6), which is a way too permissive.

We verified manually that the the prover would solve the problem in step 17 as expected without this computational limitation.

Even if the prover could have been more efficient (which is hard if we want to keep the flexibility), there would still be some computational limitations. Hence the focus of attention appears as a potential solution deserving to be investigated.

Appendix C

$\mathbb{L}_{mm}$ LDS: The *Tweety* Problem

This small example expresses well the on-going process of reasoning and the contradiction handling.

The observation function is the following:

```
Obs = [[0,pred(bird,[tweety])],
       [4,pred(ostrich,[tweety])]
      ]
```

The agent observes that Tweety is a bird before observing that it is an ostrich.

Here follow the axioms of the agent, ie the knowledge present in its long term memory and the awareness of the time:

```
Axioms = [[ltm,pred(bird,[any(y)]),0,0,0,0,
           ['->',[and,pred(bird, [any(x)]),
                      pred(now, [any(i)]),
                      [not,pred(k,[[-,any(i),1],[not,pred(flies,[any(x)])]])]],
                 pred(flies, [any(x)])],
           axiom],
          [ltm, pred(ostrich,[any(y)]),1,0,0,0,['->',pred(ostrich,[any(x)]),
                                                [not,pred(flies,[any(x)])]],
           axiom],
          [stm,0,1,0,0,0,pred(now,[0]),axiom]
         ]
```

So this means that every bird is supposed to be able to fly except if we know that the bird in question is an ostrich. Applying the inference rules described in the previous chapters, we obtain the consecutive sets of beliefs:

```
[ltm, pred(bird, [any(y)]), 0, 0, 0, 0,
      [(->), [and, pred(bird,[any(x)]),
                   pred(now, [any(i)]),
```

```
                [not, pred(k, [[-, any(i), 1], [not, pred(flies, [any(x)])]])]],
          pred(flies,[any(x)])],
      axiom]
[ltm, pred(ostrich, [any(y)]), 1, 0, 0, 0, [ (->), pred(ostrich, [any(x)]),
                                                  [not, pred(flies, [any(x)])]],
      axiom]
[stm, 0, 1, 0, 0, 0, pred(now, [0]), axiom]
[qtm, 0, 1, 0, 0, 0, pred(bird, [tweety]), obs]
```

Note: The agent observes here that Tweety is a bird.

```
[stm, 0, 1, 1, 1, 0, pred(now, [1]), clock]
[ltm, pred(bird, [any(1)]), 0, 1, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 1, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                  [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 1, 1, 20, pred(bird, [tweety]), iqs]
[qtm, 0, 0, 1, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
```

```
[stm, 0, 1, 2, 2, 0, pred(now, [2]), clock]
[ltm, pred(bird, [any(1)]), 0, 2, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 2, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                  [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 0, 2, 2, 20,
```

```
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      iqs]
[stm, 0, 1, 2, 1, 20, pred(bird, [tweety]), is]
[qtm, 0, 0, 2, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 1, 2, 0, 0, [not, pred(k, [2, [not, pred(flies, [any(1)])]])], ni]
[qtm, 0, 1, 2, 0, 0, [not, pred(k, [2, pred(flies, [any(1)])])], ni]

--------------------------------------------------------------------------------

[stm, 0, 1, 3, 3, 0, pred(now, [3]), clock]
[ltm, pred(bird, [any(1)]), 0, 3, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 3, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                   [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 3, 3, 20, [not, pred(k, [2, [not, pred(flies, [any(1)])]])], iqs]
[stm, 0, 1, 3, 3, 20, [not, pred(k, [2, pred(flies, [any(1)])])], iqs]
[stm, 0, 0, 3, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 3, 1, 20, pred(bird, [tweety]), is]
[qtm, 0, 0, 3, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 0, 3, 0, 0, pred(flies, [tweety]), emp]
[qtm, 0, 1, 3, 0, 0, [not, pred(k, [3, [not, pred(flies, [any(1)])]])], ni]
```

```
[qtm, 0, 1, 3, 0, 0, [not, pred(k, [3, pred(flies, [any(1)])])], ni]
```

```
[stm, 0, 1, 4, 4, 0, pred(now, [4]), clock]
[ltm, pred(bird, [any(1)]), 0, 4, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 4, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                   [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 0, 4, 4, 20, pred(flies, [tweety]), iqs]
[stm, 0, 1, 4, 4, 20, [not, pred(k, [3, [not, pred(flies, [any(1)])]])], iqs]
[stm, 0, 1, 4, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], iqs]
[stm, 0, 0, 4, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 4, 1, 20, pred(bird, [tweety]), is]
[qtm, 0, 1, 4, 0, 0, pred(ostrich, [tweety]), obs]
[qtm, 0, 0, 4, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 0, 4, 0, 0, pred(flies, [tweety]), emp]
[qtm, 0, 1, 4, 0, 0, [not, pred(k, [4, [not, pred(flies, [any(1)])]])], ni]
```

Note: The agent infers that tweety can fly. At the same time, he observes that Tweety is an ostrich.

```
[stm, 0, 1, 5, 5, 0, pred(now, [5]), clock]
[ltm, pred(bird, [any(1)]), 0, 5, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
```

```
[ltm, pred(ostrich, [any(1)]), 1, 5, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                    [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 5, 5, 20, pred(ostrich, [tweety]), iqs]
[stm, 0, 1, 5, 5, 20, [not, pred(k, [4, [not, pred(flies, [any(1)])]])], iqs]
[stm, 0, 0, 5, 4, 20, pred(flies, [tweety]), is]
[stm, 0, 0, 5, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 5, 1, 20, pred(bird, [tweety]), is]
[stm, 0, 1, 5, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], is]
[qtm, 0, 1, 5, 0, 0, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], sr]
[qtm, 0, 0, 5, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 0, 5, 0, 0, pred(flies, [tweety]), emp]
[qtm, 0, 1, 5, 0, 0, [not, pred(k, [5, [not, pred(flies, [any(1)])]])], ni]

--------------------------------------------------------------------------------

[stm, 0, 1, 6, 6, 0, pred(now, [6]), clock]
[ltm, pred(bird, [any(1)]), 0, 6, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 6, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                    [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 6, 6, 20, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], iqs]
[stm, 0, 1, 6, 6, 20, [not, pred(k, [5, [not, pred(flies, [any(1)])]])], iqs]
[stm, 0, 1, 6, 5, 20, pred(ostrich, [tweety]), is]
[stm, 0, 0, 6, 4, 20, pred(flies, [tweety]), is]
[stm, 0, 0, 6, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
```

```
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 6, 1, 20, pred(bird, [tweety]), is]
[stm, 0, 1, 6, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], is]
[qtm, 0, 1, 6, 0, 0, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], sr]
[qtm, 0, 0, 6, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 1, 6, 0, 0, [not, pred(flies, [tweety])], mp]
[qtm, 0, 0, 6, 0, 0, pred(flies, [tweety]), emp]
[qtm, 0, 1, 6, 0, 0, [not, pred(k, [6, [not, pred(flies, [any(1)])]])], ni]

--------------------------------------------------------------------------------

[stm, 0, 1, 7, 7, 0, pred(now, [7]), clock]
[ltm, pred(bird, [any(1)]), 0, 7, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 7, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                   [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 7, 7, 20, [not, pred(flies, [tweety])], iqs]
[stm, 0, 1, 7, 7, 20, [not, pred(k, [6, [not, pred(flies, [any(1)])]])], iqs]
[stm, 0, 1, 7, 6, 20, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], is]
[stm, 0, 1, 7, 5, 20, pred(ostrich, [tweety]), is]
[stm, 0, 0, 7, 4, 20, pred(flies, [tweety]), is]
[stm, 0, 0, 7, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 7, 1, 20, pred(bird, [tweety]), is]
[stm, 0, 1, 7, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], is]
[qtm, 0, 1, 7, 0, 0, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], sr]
[qtm, 0, 0, 7, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
```

```
                  [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
            pred(flies, [any(2)])],
      sr]
[qtm, 0, 1, 7, 0, 0, [not, pred(flies, [tweety])], mp]
[qtm, 0, 0, 7, 0, 0, pred(flies, [tweety]), emp]
[qtm, 0, 1, 7, 0, 0, pred(loses, [pred(flies, [tweety])]), cd2a]
```

Note: The agent deduces that Tweety can not fly (because Tweety is an ostrich). The belief that Tweety can not fly having a bigger degree of certainty that the belief that Tweety can fly, this second belief is lost (predicate "loses").

```
[stm, 0, 1, 8, 8, 0, pred(now, [8]), clock]
[ltm, pred(bird, [any(1)]), 0, 8, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                  pred(now, [any(2)]),
                  [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
            pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 8, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                   [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 8, 8, 20, pred(loses, [pred(flies, [tweety])]), iqs]
[stm, 0, 1, 8, 7, 20, [not, pred(flies, [tweety])], is]
[stm, 0, 1, 8, 6, 20, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], is]
[stm, 0, 1, 8, 5, 20, pred(ostrich, [tweety]), is]
[stm, 0, 0, 8, 4, 20, pred(flies, [tweety]), is]
[stm, 0, 0, 8, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                  pred(now, [any(1)]),
                  [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
            pred(flies, [any(2)])],
      is]
[stm, 0, 1, 8, 1, 20, pred(bird, [tweety]), is]
[stm, 0, 1, 8, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], is]
[stm, 0, 1, 8, 7, 20, [not, pred(k, [6, [not, pred(flies, [any(1)])]])], is]
[qtm, 0, 1, 8, 0, 0, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], sr]
[qtm, 0, 0, 8, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                  pred(now, [any(1)]),
                  [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
            pred(flies, [any(2)])],
      sr]
[qtm, 0, 1, 8, 0, 0, [not, pred(flies, [tweety])], mp]
```

```
[qtm, 0, 1, 8, 0, 0, pred(loses, [pred(flies, [tweety])]), cd2a]

--------------------------------------------------------------------------------

[stm, 0, 1, 9, 9, 0, pred(now, [9]), clock]
[ltm, pred(bird, [any(1)]), 0, 9, 0, 0,
      [ (->), [and, pred(bird, [any(3)]),
                    pred(now, [any(2)]),
                    [not, pred(k, [[-, any(2), 1], [not, pred(flies, [any(3)])]])]],
              pred(flies, [any(3)])],
      il]
[ltm, pred(ostrich, [any(1)]), 1, 9, 0, 0, [ (->), pred(ostrich, [any(2)]),
                                                   [not, pred(flies, [any(2)])]],
      il]
[stm, 0, 1, 9, 8, 20, pred(loses, [pred(flies, [tweety])]), is]
[stm, 0, 1, 9, 7, 20, [not, pred(flies, [tweety])], is]
[stm, 0, 1, 9, 6, 20, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], is]
[stm, 0, 1, 9, 5, 20, pred(ostrich, [tweety]), is]
[stm, 0, 0, 9, 2, 20,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      is]
[stm, 0, 1, 9, 1, 20, pred(bird, [tweety]), is]
[stm, 0, 1, 9, 4, 20, [not, pred(k, [3, pred(flies, [any(1)])])], is]
[stm, 0, 1, 9, 7, 20, [not, pred(k, [6, [not, pred(flies, [any(1)])]])], is]
[qtm, 0, 1, 9, 0, 0, [ (->), pred(ostrich, [any(1)]), [not, pred(flies, [any(1)])]], sr]
[qtm, 0, 0, 9, 0, 0,
      [ (->), [and, pred(bird, [any(2)]),
                    pred(now, [any(1)]),
                    [not, pred(k, [[-, any(1), 1], [not, pred(flies, [any(2)])]])]],
              pred(flies, [any(2)])],
      sr]
[qtm, 0, 1, 9, 0, 0, [not, pred(flies, [tweety])], mp]
[qtm, 0, 1, 9, 0, 0, [not, pred(k, [9, pred(flies, [any(1)])])], ni]
```

Note: The "loses" predicate prevents the next deductions of the belief that Tweety can fly.