Addressing Constraint Failures in Distributed Dialogue Protocols

Nardine Z. Osman

Master of Science
School of Informatics
University of Edinburgh

2003

Abstract

Early agent communication languages lacked the means to coordinate the interaction between agents. The importance of coordination was better appreciated when open systems became in favor. Some solutions have been proposed; however, these normally rely on centralized gatekeepers. Distributed Dialogues, on the other hand, provide both communication and coordination protocols for multi-agent systems. They do that without the need to re-program agents every time they need to adapt to a new dialogue system and without the need for centralized gatekeepers for coordination.

However, these dialogue protocols still aren't tolerant to failure. When an agent does not respond appropriately, the dialogue simply stops. This usually happens when constraints are broken. This dissertation addresses these kinds of failure by offering 'induced backtracking' to explore other parts of the dialogue. It also proposes a negotiation protocol that could be used when some failed constraints are better dealt with by negotiation.

Acknowledgements

Many thanks to my supervisor, Dr. Dave Robertson, for his support and advice and for making my dissertation work pleasant and interesting. Our meetings and discussions have been the most important source of information for this dissertation; his suggestions and ideas have been invaluable for my work.

I owe special gratitude to my father for his tremendous emotional and financial support, without him my studies would not have been possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nardine Z. Osman*)

To Rida . . .

Table of Contents

Chapter 1

Introduction

Language and communication has been a major philosophical issue since the very early ages of humanity. Things are not different in the computer science world. Communication has again been the central issue that helps define how systems might interact with one another. However, communication got more complicated in concurrent systems where the synchronisation of processes became critical. This led to the introduction of the CCS process calculus and CTL temporal logic theories. Lately, with the demand of large scale multi-agent systems that require agents to automatically solve problems collaboratively, synchronisation and coordination again became a critical issue of agent interaction.

In what follows, we will give a brief introduction to the history of communication in multi-agent systems that will lead us to introduce Robertson's *distributed dialogue* method for communication/coordination.

1.1 Communication in Multi-Agent Systems

To better understand communication in multi-agent systems, let us first take a look at communication in object-oriented programming. In the latter case, communication takes the shape of method invocation. Let's consider the following example from [Woo02]. Object *o1* can communicate with object *o2* by invoking the public available method *m2* of object *o2*. Taking a closer look at this example, we notice that the decision of execution of *m2* lies entirely in the hands of *o1* rather than *o2*! Such actions will be unacceptable in the agent community.

In general, agents are known to be autonomous. An agent would have its own beliefs and goals, and it will act in accordance with its beliefs in order to pursue its goals. This implies that agents cannot control other agent's actions or beliefs. What they *could* do, is attempt to influence each other's beliefs (hence decisions and actions). This leads to the introduction of a new concept of actions based on the theories of *speech acts*. The following section elaborates on this issue.

1.1.1 Performative Languages

Agent communication highlighted the need to perform actions different from the *usual* actions. These actions were 'communicative actions'. Agent Communication Languages that followed were based on Austin's theories of speech acts [Aus62]. Austin has defined some utterances to be 'speech acts' since they were capable of changing the state of the world surrounding them. Speech acts were based on performative verbs like *request*, *inform*, *promise*, etc. Austin's theory was then modified by John Searle, [Sea79], who identified the different types of speech acts and added some properties that need to hold for speech acts to succeed (like normal i/o conditions, sincerity conditions, etc.).

In the early 1990's, Knowledge Query and Manipulation Language (KQML) was delivered along with Knowledge Interchange Format(KIF) as a method for knowledge sharing [Woo02]. KIF described the representation of knowledge of a certain domain. The KQML, on the other hand, was a language for agent communication. It defined a general format for messages and wasn't concerned about the content of the messages (as in KIF). Messages were formed of performatives (e.g.*tell*, *ask-about*, *discard*, *deny*, etc.) and its parameters.

KQML was criticised on many aspects, some of which are the weakly defined semantics, the missing *commissives* class of performatives (the class that deals with commitments), the excessively long performative set (made up of 41 performatives), etc. These criticisms led to the development of FIPA–ACL. However, FIPA–ACL was similar to KQML. It didn't define any specific language for message content. Messages were very similar to those of KQML, the only major difference was the number of performatives provided: 20 performatives. Table 1.1 introduces and categorises the performatives provided by FIPA–ACL [FIP97]. The information passing category allows sharing of knowledge either by imparting information or by confirming/disconfirming the accuracy of information. Information requesting allows the sender to query the receiver about some something. Negotiation actions allow requesting, making, accepting or rejecting proposals. Performing actions allow the sender to agree or refuse to perform actions as well as request others to perform actions. Error handling actions allow informing the receiving agent about misunderstandings and failures.

Performative	Passing Information	Requesting Information	Negotiation	Performing Actions	Error Handling
accept-proposal			×		
agree				×	
cancel		×		×	
cfp			×		
confirm	×				
disconfirm	×				
failure					×
inform	×				
inform-if	×				
inform-ref	×				
not-understood					×
propagate				×	
propose			×		
proxy				×	
query-if		×			
query-ref		×			
refuse				×	
reject-proposal			×		
request				×	
request-when				×	
request-whenever				×	
subscribe		×			

Table 1.1: the performatives provided by FIPA–ACL

1.1.2 The Coordination Issue

Let's consider the following scenario of an English auction system [Rob]. The bidder agents are only allowed to bid when invited by the auctioneer, and their bidding value should be higher than the current leading bid. These constraints are critical constraints, that if unsatisfied, might break down the system or result in unwanted behaviour. 'For instance, an agent operating as if our English auction were a Dutch auction might wait forever for the leading bid announced by the auctioneer to fall (as it does in Dutch auctions) when in fact the (English) auctioneer will assure that the leading bid always rises'. Hence, the need for further coordination rises.

The communication protocols described earlier define how agents can communicate. However, for further coordination, we need to deal with *when* are agents allowed to communicate. This imposes further constraints on agents' communication. These constraints could either be on the sequence of messages, on the content of messages, or even on the status of the dialogue.

A common solution to this problem is to have agents pre-engineered to deal with the dialogues they are known to encounter. But what happens in open systems? The problem in such systems is that the agents will have to be re-engineered for every new dialogue scenario they might have to adapt to. With the emerging need for such large-scale systems, a more reasonable solution needs to be investigated. The concept of Electronic Institutions addresses this problem by defining *agent protocols*, as opposed to *communication protocols such as KQML and FIPA–ACL*, that would deal with the coordination issues. The agents will still use the communication protocols to communicate; however, an upper layer of protocols – the *agent protocols* – will deal with the coordination issues [WR02].

1.1.3 Electronic Institutions

The idea behind these electronic institutions is that human interaction, in general, is always accompanied with 'social conventions'. For instance, when people are bidding in an English auction, it is automatically assumed that people abide with the English auction's rules or conventions. Similarly, agent's interaction should also be accompanied with 'social conventions' in order to insure coordination. Each electronic institution will have its own set of 'social conventions'. For instance, an English auction system could be represented by one electronic institution and a Dutch auction system by another. The representation is very similar to that of the CCS process algebra. State transition diagrams are used to express the acceptable actions each agent can take at a certain time. Actions in these cases are *usually* communicative actions.

Yet another problem arises with these *agent protocols*. In practice, these protocols are implemented by making use of a centralised gatekeeper. All messages would be directed through this gatekeeper which would ensure message sequencing and provide the coordination needed.

However, this centralised gatekeeper might itself become a bottleneck. Moreover, these gatekeepers do not deal with all coordination constraints issues (e.g. the constraints imposed on message content). The next section introduces yet another method for communication / coordination: *distributed dialogues*. These dialogues are an attempt to solve the communication/ coordination problems discussed above.

1.2 Distributed Dialogues

Distributed dialogues do not require agents to be re-programmed each time they need to adapt to a new system of dialogue, nor to adapt their knowledge or beliefs solely for the purposes of dialogues. They preserve the autonomy of agents and offer dialogue coordination without the need for centralised gatekeepers. Distributes dialogues are a form of both communication and coordination protocols. In other words, they replace communication languages such as FIPA–ACL and coordination protocols such as electronic institutions.

1.2.1 The Concept

"A distributed dialogue is a conversation among a group of agents which can be described as a collection of dialogue sequences between agents" [Rob]. Hence, a distributed dialogue is a specification of the allowed dialogue sequence. This could be thought of as a code for the transition diagram of a certain scenario's dialogue.

How does it work?
The dialogue starts when one agent takes a certain permissible action. It marks that action as closed, and sends the dialogue protocol to the other agent it is communicating with. The other agent then looks up the received dialogue protocol for permissible actions it can take, marks the completed action as closed, and send the protocol to the appropriate agent. The process iterates until the dialogue completes successfully.

1.2.2 Failure Issues

The problem with these dialogue protocols is that they are not tolerant to failure. Hence, when an agent does not respond appropriately, the dialogue simply stops. This usually happens when constraints are broken. This dissertation addresses these kinds of failure. It provides a mechanism to backtrack and explore other parts of the dialogue when one part fails. However, upon failure, some constraints might better be solved by negotiation instead of directly marking them as failed. So the second part of the dissertation will provide a sample negotiation protocol that might be useful in certain cases of constraint failure.

To better understand our work, we will start with Chapter 2 with some background on the dialogue protocols and how they actually work. After that, Chapter 3 will introduce our 'induced backtracking' as a solution to constraints failure. Chapter 4 will then provide the sample negotiation protocol.

Chapter 2

Background

In order to better understand our work in chapters 3 and 4, one needs an idea of how do these dialogues work. This chapter provides a detailed description of the distributed dialogue protocols. It starts with a description of the representation used, then explains how these protocols actually work, and finally offers an overview of what these dialogues achieve.

2.1 Dialogue Representation

2.1.1 Dialogue Clauses

Dialogue protocols provide coordination in dialogue systems. They achieve this by restricting agent's actions to those that apply to the system's 'social conventions'. Conventions are implemented by defining rules for the participating parties. These rules, however, do not encode individual agents but the roles those agents take (for instance, rules for *bidder* agents). Hence, agents in the distributed dialogues are expressed by *a(role, id)*, where *role* is the role of the agent in that dialogue (e.g. bidder) and *id* is the agent's unique identifier.

Rules in a dialogue system are implemented by giving *definitions* to agents' roles, i.e. defining the actions a certain agent can take. These could either be taking another role, a message passing action, a do nothing action (or *null* action), or a combination of other actions (or *definitions*). The logical connectives used are the *then*, *or* and *par*. Definitions of the form *'A then B'* imply that action *A* is followed by action *B*. Where as *'A or B'* implies that either action A or action B can take place. Finally, *'A par B'* (or A in parallel with B) implies that both actions *'A'* and *'B'* need to be fulfilled but the order of doing this does not matter.

As for message passing actions, these could either be of the form of receiving a message ($M \Leftarrow A$) or of sending one ($M \Rightarrow A$). As mentioned in the introduction, coordination issues might not be restricted to the dialogue sequence. Hence, further constraints could be added to the message passing actions. [Rob] distinguishes two types of constraints: *proaction constraints* and *reaction constraints*. The *proaction constraints* 'define the circumstances under which a message allowed by the dialogue framework is allowed to be sent'. These constraints are of the form $M \Rightarrow A \leftarrow C$, where C is the condition for sending M to A. The *reaction constraints*, on the other hand, 'define what should be true in an agent following receipt of a message allowed by the dialogue framework'. These constraints are of the form $C \leftarrow M \Leftarrow A$, where C is the reaction upon receiving M from A. The idea behind *proaction* and *reaction constraints* is to allow further testing of the state of the agent or its beliefs. However, constraints could also be used to further test dialogue state, message content, etc. Hence, constraints could also be added to other kinds of *definitions*. For example, when the next action is to take a different role.

2.1.2 Dialogue Framework

We have seen how to give *definitions* to different roles in a dialogue. However, a dialogue system (like the *English auction system*) is made up of several interacting roles (like the *auctioneer*, *bidder*, etc.). Hence, we will need a set of *dialogue clauses*. This set will be known as the *dialogue framework*. As a result, the syntax of the *dialogue framework* will be defined as follows [Rob]:

$Framework := \{Clause, \ldots\}$

$Clause := Agent :: Def$

$Agent := a(Role, Id)$

$Def := \textbf{null} \mid Agent \mid Message \mid Def\ then\ Def \mid Def\ or\ Def \mid Def\ par\ Def$

$Message := M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid C \leftarrow M \Leftarrow Agent$

$C := \textbf{Term} \mid C \wedge C \mid C \vee C$

$Role := \textbf{Term}$

$Id := \textbf{Constant}$

$M := \textbf{Term}$

where *Term* is a structured term and *Constant* is a constant.

2.1.3 Dialogue Protocol

We have been talking about agents communicating by means of sending and receiving the dialogue protocol itself. But what does this dialogue protocol exactly consist of? The dialogue protocol is a collection of the *dialogue state*, the *dialogue framework* and the *common knowledge*. The *dialogue framework*, which we saw above in section 2.1.2 and which consists of sets of dialogue clauses, is the generic protocol. This protocol is preserved throughout the dialogue.

However, during communication, the protocol might need to be modified in order to keep track of the current state of the protocol. Instances of the generic *dialogue clauses* will always get updated in the *dialogue state* to help agents determine the current protocol state, and hence their next action to be taken. This approach frees the agents from the need of keeping track of all the dialogues they're engaged in as well as their state in each dialogue. In the following section, we will demonstrate how this is achieved.

Finally, the dialogue protocol also consists of the *common knowledge*. The *common knowledge* contains the knowledge that might be needed solely for the given dialogue protocol. Hence, this also frees agents from the need to adapt their own knowledge or beliefs only for the sake of some dialogue system. Moreover, throughout the course of the dialogue, agents will also be capable of modifying that knowledge as needed by asserting to and retracting from the *common knowledge*. Knowledge is expressed in the form of *known(Agent, Knowledge)*, where *Knowledge* 'could be expressed in any form, as long as it can be processed reliably by all appropriate agents' [Rob].

2.2 Implementing Coordination

We have seen the format of the dialogue protocol that is sent along with the messages during communication. In what follows, we will introduce what happens upon the receipt of a message, how do agents retrieve the dialogue state and decide upon their next action to be taken.

Upon the receipt of a certain message, the agent first needs to lookup for the appropriate *dialogue clause* to deal with. If the agent is already in the middle of the dialogue, then a modified copy of the generic *dialogue clause* should be available in the *dialogue state* section of the protocol received. The appropriate clause is retrieved by checking for the appropriate role. However, if the agent is starting a new role within this dialogue, then it won't be able to find an appropriate *dialogue clause* copy from the *dialogue state*, and will then fetch a generic copy from the *dialogue framework*.

After obtaining the correct *dialogue clause* to work with, the agent will need to find its next action to be taken, complete the action, and modify the *dialogue clause* appropriately. This is achieved by applying rewrite rules to expand the *dialogue state*. Rewrite rules often result in having messages to be sent to certain agents. The rewrite rules are applied repeatedly either until no change occurs in the result or until no rewrite rules match anymore. At this point, the agent should send the appropriate messages to their corresponding agents attaching the new *dialogue state* to these sent messages.

The following table defines the rewrite rules used [Rob]:

$\mathrm{A} \xrightarrow{M_i,M_o,P,O} \mathrm{A :: B}$	if	clause(P, A::B)
$\mathrm{A :: B} \xrightarrow{M_i,M_o,P,O} \mathrm{A :: E}$	if	$\mathrm{B} \xrightarrow{M_i,M_o,P,O} \mathrm{E}$
$\mathrm{A_1 \text{ or } A_2} \xrightarrow{M_i,M_o,P,O} \mathrm{E}$	if	$(\mathrm{A_1} \xrightarrow{M_i,M_o,P,O} \mathrm{E}) \vee (\mathrm{A_2} \xrightarrow{M_i,M_o,P,O} \mathrm{E})$
$\mathrm{A_1 \text{ par } A_2} \xrightarrow{M_i,M_o,P,O_1 \cup O_2} \mathrm{E_1 \text{ par } E_2}$	if	$((\mathrm{A_1} \xrightarrow{M_i,M_n,P,O_1} \mathrm{E_1}) \wedge (\mathrm{A_2} \xrightarrow{M_n,M_o,P,O_2} \mathrm{E_2}))$ $\vee ((\mathrm{A_2} \xrightarrow{M_i,M_n,P,O_1} \mathrm{E_2}) \wedge (\mathrm{A_1} \xrightarrow{M_n,M_o,P,O_2} \mathrm{E_1}))$
$\mathrm{A_1 \text{ then } A_2} \xrightarrow{M_i,M_o,P,O} \mathrm{E \text{ then } A_2}$	if	$\mathrm{A_1} \xrightarrow{M_i,M_o,P,O} \mathrm{E}$
$\mathrm{A_1 \text{ then } A_2} \xrightarrow{M_i,M_o,P,O} \mathrm{A_1 \text{ then } E}$	if	$(\mathrm{closed(A_1)}) \wedge (\mathrm{A_2} \xrightarrow{M_i,M_o,P,O} \mathrm{E})$
$\mathrm{M \Leftarrow A_s} \xrightarrow{M_i,M_i-\{M \Leftarrow A_s\},P,\emptyset} \mathrm{c(M \Leftarrow A_s)}$	if	$((\mathrm{M \Leftarrow A_s}) \in \mathrm{M}_i) \wedge \mathrm{receive(P,M)}$
$\mathrm{M \Rightarrow A} \xrightarrow{M_i,M_i,P,\{M \Rightarrow A\}} \mathrm{c(M \Rightarrow A)}$	if	$\mathrm{send(P}_n\mathrm{,M)}$
$\mathrm{null} \xrightarrow{M_i,M_i,P,\emptyset} \mathrm{c(null)}$		
$\mathrm{A \leftarrow C} \xrightarrow{M_i,M_o,P,O} \mathrm{A :: B}$	if	$\mathrm{clause(P, A::B)} \wedge \mathrm{satisfy(C)}$
$\mathrm{C \leftarrow M \Leftarrow A_s} \xrightarrow{M_i,M_i-\{M \Leftarrow A_s\},P,\emptyset} \mathrm{c(M \Leftarrow A_s)}$	if	$((\mathrm{M \Leftarrow A_s}) \in \mathrm{M}_i) \wedge \mathrm{receive(P,M)} \wedge \mathrm{satisfy(C)}$
$\mathrm{M \Rightarrow A \leftarrow C} \xrightarrow{M_i,M_i,P,\{M \Rightarrow A\}} \mathrm{c(M \Rightarrow A)}$	if	$\mathrm{send(P}_n\mathrm{,M)} \wedge \mathrm{satisfy(C)}$
$\mathrm{null \leftarrow C} \xrightarrow{M_i,M_i,P,\emptyset} \mathrm{c(null)}$	if	satisfy(C)

Table 2.1: the rewrite rules used for expanding *dialogue clauses*

M_i above represents the set of incoming messages (the received messages), M_o represents the remaining set of incoming messages, P represents the *dialogue protocol* and O the set of output messages (the messages to be sent).

The rewrite rules of Table 2.1, expressed informally, say that an agent A may be rewritten to $A::B$ if the clause $A::B$ was part of the protocol P. The *dialogue clause* $A::B$ may be rewritten to $A::E$ if B may be rewritten to E. A_1 *or* A_2 may be rewritten to E if either A_1 or A_2 may be rewritten to E. A_1 *par* A_2 may be rewritten to E_1 *par* E_2 if both A_1 may be rewritten to E_1 and A_2 may be rewritten to E_2 (in whatever order). A_1 *then* A_2 may either be rewritten to E *then* A_2 if A_1 may be rewritten to E, or it may be rewritten to A_1 *then* E if the actions in A_1 have already been completed and A_2 may be rewritten to E. This introduces the *'closed'* concept. When actions are completed, they are marked as *closed* to denote their completion. But how to test when a *definition* is closed? The following are guidelines to be used in testing for *closed* definitions:

– $A :: B$ is closed if B is closed
– A *or* B is closed if either A or B is closed
– A *par* B is closed if both A and B are closed
– A *then* B is closed if A is closed and B is closed
– $c(T)$ resembles a closed term T

And when are terms marked as *closed*?
$M \Leftarrow A_s$ is rewritten to $c(M \Leftarrow A_s)$ if the message M is received along with the protocol P and $M \Leftarrow A_s$ is an element of the incoming messages M_i. After marking $M \Leftarrow A_s$ as closed, the new set of incoming messages M_o will now be M_i - $\{M \Leftarrow A_s\}$ and the set of outgoing messages O will be $\emptyset$. Rewriting $M \Rightarrow A$ to $c(M \Rightarrow A)$ is much more straight forward. The term is directly closed, the set of outgoing messages is $\emptyset$, and the message M is sent along with the new protocol P_n. As for the empty term *null*, it is automatically closed.

The last four rewrite rules are a repetition of four of the above mentioned rules. The difference here is the addition of constraints. Hence, in order to be able to rewrite those terms the constraint C first needs to be satisfied.

2.3 Accomplishments

We have seen how these dialogues work. As promised, the distributed dialogues do offer distributed coordination (as opposed to the centralised gatekeepers solution). They also free agents from the need to keep track of the dialogues they're engaged in as well as their state in each dialogue. This is now possible, with the methodology proposed above, because with each message an agent receives, the protocol is also attached. From the protocol, the agent will be able to compute it's current state in the dialogue and the actions it needs to take. Moreover, knowledge exclusive to a specific dialogue is embedded in its protocol. Hence, this also frees agents from the need to adapt their knowledge to the dialogue's knowledge. As a result, these distributed dialogues remove the need of standardising the engineering of individual agents. They preserve the availability of truly autonomous agents that are capable of getting engaged in several dialogues simultaneously in large scale open systems, and remove the need for global controllers for coordination. Distributed dialogues accomplish these tasks by requiring two engineering commitments on the agent design level. The first is that the constraints must be comprehensible for the agents, i.e. they 'must be in an ontology recognised (possibly via translation) by the agent' [Rob]. The other is that the agents should be able to encode and decode the dialogue protocols (as illustrated in the sections above) in order to be able to extract the protocol state and act accordingly.

The background given in this chapter is sufficient for understanding this dissertation. However, for further readings, [Rob] offers much more interesting information on this subject, such as a comparison to performative languages, possible implementations, possible human interaction with the protocol, etc.

Chapter 3

Failure Recovery: Induced Backtracking

In this chapter we address the problem of failure recovery in dialogues when constraints are not satisfied. As we saw earlier, distributed dialogues offer dialogue coordination; the protocol controls the path the dialogue between agents takes by specifying the acceptable message sequences. But what happens when a certain path fails? In this case, the dialogue simply stops!

In order to better explain the problem, consider the following example: an e-commerce scenario where an agent A wants to buy a piece of furniture from agent B. Say agent A is badly in need of a desk for the office as soon as possible. It will then initiate a dialogue with agent B. Now the dialogue can take different paths. For instance, agent A may either end up buying a readily available desk if a suitable one is found, or it may buy one that is packed flat for home assembly. However, if agent B neither has a readily available one nor one for home assembly, then the dialogue should fail in the sense that the goal of the dialogue is impossible to achieve.

However, what actually happens is that when a certain path is selected, agents cannot backtrack later during communication to select a different path. So let's say the first path was selected and agent A asked agent B for its available desks. After a discussion with agent B, agent A notices that none of the readily available desks are suitable. At this point, agent A should be able to backtrack and select another acceptable dialogue path, which is in this case to inquire about the packed desk for home assembly. Unfortunately, this was not possible with the original protocol mechanism. This chapter discusses the method used to induce backtracking for this kind of failure – failure of constraints.

3.1 Inducing Backtracking

As we saw in the example above, the dialogue sometimes fails before exploring all the possible approaches it can follow to complete successfully. The solution to that is to induce backtracking to help implement a depth first search. Hence, whenever one path fails, the dialogue backtracks until it finds another path it could take. Moreover, a depth first search guarantees that the whole space will be explored, as long as there are no infinite paths or loops in the search tree. (The end of Section 3.2.2 better elaborates on the issue of infinite paths or loops in such dialogues.)

Chapter 2 explained how the protocol actually works and the expansion concept in detail. In this section we explain the changes that were made in order to mark failed paths and properly guide the dialogue.

3.1.1 Marking Failed Atomic Terms

The first step to be made is to mark atomic terms as failed whenever their constraints are not satisfied. An atomic term represents an action to be taken by an agent. These actions are of the following forms:

– no action to be taken: *null*
– the agent should take a different role: *a(Role, Id)*
– send a message to another agent: $M \Rightarrow A$
– receive a message from another agent: $M \Leftarrow A$

Along with these actions, proaction and reaction constraints are added as needed. Hence, the expansion of these atomic terms should not be altered, except when a corresponding constraint is not satisfied. In the latter case, the atomic term should be marked as *failed* instead of *closed*.

The only exception to the above is the case of receiving a message. In this case, three different scenarios can occur:

CASE 1: The message is received and the constraint, if any, is satis£ed:

In this case, the atomic term is marked as closed.

CASE 2: The message is received and the constraint is not satis£ed:

In this case, the atomic term is **not** marked as failed. However, a message is sent back to the sender to inform it of the failure of the sent message. This is the only case when the agent forces the other agent to do the backtracking. This is necessary because if the agent does mark that term as failed and does the backtracking itself, then in most cases it will end up waiting for another message and the other agent wouldn't even know about the failure that occurred. Hence we will end up with two agents each expecting to receive a message from the other. (Note that dealing with the receipt of failed messages is discussed later in section 3.2.1)

CASE 3: The message received is not the message the agent is expecting:

This case takes place when one of the agents backtracks and sends a different message than that the other agent is expecting. Hence, the receiving agent does not delete the received message from the list of incoming messages and it marks the atomic term at which it is standing as failed. This will then force it to backtrack searching for the other path in order to synchronise with the sending agent.

3.1.2 Guiding Dialogue Backtracking

Section 3.1.1 discusses how atomic terms are marked as failed, and this is what induces backtracking. However, we now need to guide the backtracking process. To do that, the expansion rules (or rewrite rules), described in Chapter 2 and concerning the different connectives (*then*, *or* and *par*), were modified as follows:

The *then* Connective:

To understand this case better, let's think of a dialogue tree. A path in that tree is a sequence of *A then B* terms. If one of the nodes on that path is marked as failed, then the whole path should be marked as failed. To do that, we need to move backwards and mark all previously closed terms on that path as failed until we hit a disjunction of events and jump to another path.

Hence, as before, if *A* is closed, then *B* should then be expanded. But if *A* is closed and *B* is marked as a failed term, then *A* should be marked as failed too. However, if *A* is marked as failed, then there is no need to expand *B* and the term *A then B* is marked as failed. Otherwise, if *A* is neither marked as closed nor as failed, then it is expanded.

The *or* Connective:

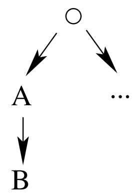

Figure 3.1: the *'A then B'* term in dialogue trees

When we reach a node on the tree where two paths may be selected, first the left-most path is chosen. If that path fails, then the other may be selected. The node is marked as failed only after all paths fail. Hence, if a disjunction of terms of the form *A or B* is reached then the first thing to do is to test whether both *A* and *B* have failed. If they did, then the *A or B* term is marked as failed too. Otherwise, A is expanded first. If that path fails, then B is expanded.

The *par* Connective:

The term *A par B* implies that both *A* and *B* should take place; however, there is no preference concerning the order of the actions. Hence, either *A* is expanded and then *B*, or the other way around. If any of them fails, then the term *A par B* is marked as failed.

3.1.3 Additional Modi£cations

At the beginning of our implementation, testing was done by using examples that would test all possible structures of terms. This shed light on some problems of these dialogue protocols. As we will discuss shortly, with some modifications, many of those problems were solved. Nevertheless, other tougher problems were uncovered. We discuss these later.

The sections above introduced the modifications made to induce and guide backtracking. Those modifications were not enough to ensure that the protocol works correctly. For instance, let's consider dealing with terms of the form *(A or B) then C*. If *A* is first selected and later on *C* fails, then *C* is marked as failed and so is *A*. The other path should then be selected and *B* is expanded. After expanding *B*, *C* should be re-expanded next; however, it is marked as failed! In order not to fall in the trap of deciding when is it permissible to open a previously failed term and re-expand it, every generic protocol is modified at the very beginning of the protocol run in order to convert all terms of the form *(A or B) then C* into (A then C) or (B then C) (illustrated in Figure 3.2 below).

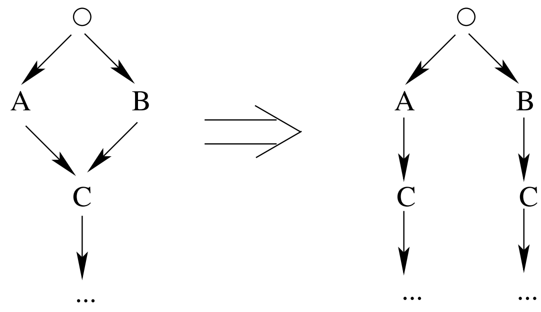

Figure 3.2: converting terms of the form *'(A or B) then C'* into *'(A then C) or (B then C)'*

It is true that this will introduce many redundant terms; however, it preserves a very neat solution to our problem. Moreover, when testing many fairly complicated protocols full of terms of the form *(A or B) then C*, no effect was noted on the performance. Hence, without degrading performance, this method offers a solution which (with the automatic translation) is also transparent to the users – the protocol engineers.

Another faced problem was that of reaching a deadlock by having each communicating party waiting to receive a message from the other. This could happen under several circumstances, one of which is the case of failing to satisfy a reaction constraint (this is the case when a message is received and the constraint is not satisfied, discussed earlier in section 3.1.2). Another case is illustrated in the figure below. It is the case when an agent backtracks and reaches a disjunction of terms where the next action to take is to receive a message.

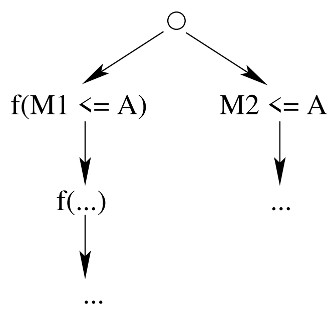

Figure 3.3: backtracking to a disjunction of *'$M \Leftarrow A$'* terms

To solve this problem, modifications where made to the expansion protocol of the *'or'* connective discussed earlier, and the following new case was added:

– When a disjunction of terms is reached and one of the terms has already failed, then the first atomic term in each path is examined. If all are of the form $M \Leftarrow A$, then the expansion stops and a message is sent back to the sender of the failed term forcing it to do the backtracking.

To understand this case better, let's look at the example in the figure above. When the left-most path fails, eventually $M1 \Leftarrow A$ is marked as failed. Now since we reach a disjunction of terms of the form $M \Leftarrow A$ and one of them has failed, the expansion stops here and a message indicating the failure of receipt of *M1* is sent to agent *A*.

But how do agents deal with an incoming failed message notice?
When an agent receives a message accompanied by a protocol, the first thing it does is to check whether the message is a notice of an earlier failed sending message action (i.e. of the form $f(M \Rightarrow B)$). If it is, then it marks all previously closed terms matching the failed message as failed. Otherwise, nothing happens. Only after that the protocol expansion process proceeds.

Now we move on to discuss yet another complication. This complication arises when an agent whose protocol has already completed successfully receives an incoming message. Most of the times this happens at the very last step of a dialogue. One agent sends the last message in the dialogue; however, the receiving agent's constraints fail to be satisfied. Hence, the first agent, which has already completed its part in a dialogue successfully, eventually receives a failed message notice. Now whenever a message is received and the protocol is considered closed, the protocol should be opened again. In this case, it is automatically opened after marking that previously closed term(s) matching the incoming failed message as failed. Hence, when expansion is called, backtracking automatically takes place.

Nevertheless, it is not always the case that the received message is a failed message notice. If we consider the case in which multiple agents are communicating, then it could very easily happen that one of those agents fulfils its part in that dialogue, then another agent backtracks and sends back another message. The following figure better explain this situation.

Let us consider the case when agent *C* is sending its last message *M5*. At this point it is clear that agent *B* has already completed its part in the dialogue. However, if *C* now fails to satisfy a proaction constraint accompanied with sending *M5*, then *C* will backtrack and send *M3* to agent *B*.

What can agent *B* do then?
We noted above that whenever a message is received and the dialogue is marked as *closed*, this implies that the protocol shouldn't be closed and backtracking should take place. This is automatically done in the case when the received message is a failure message notice (as noted

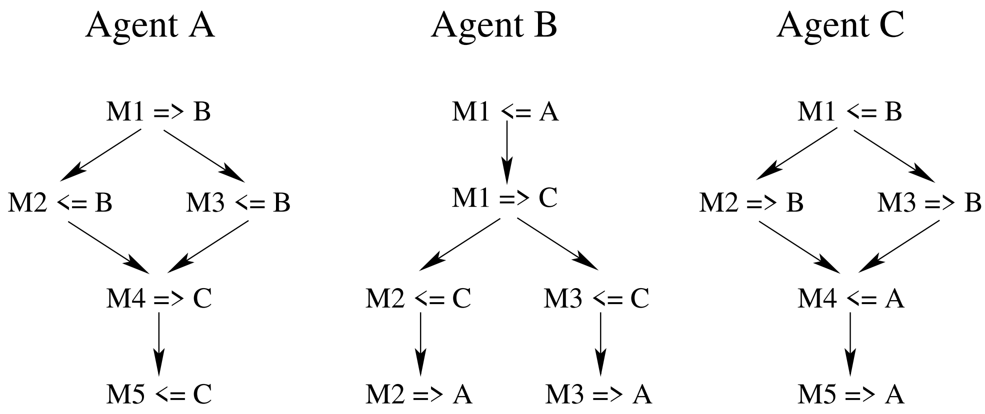

Figure 3.4: an example of multiple agents communicating

above). But what if the message is a normal message (i.e. not a failure message notice)? In this case, the dialogue needs to re-open the protocol by marking all leaf nodes (i.e. nodes that have no children) that are already closed as failed. Hence, in agent B's tree above, the only closed leaf node would be the node $M2 \Rightarrow A$. Hence, it will be marked as failed which will force agent B to backtrack until it reaches the node $M3 \Leftarrow C$. From there the protocol will proceed successfully.

3.2 Limitations

We have seen the theory behind inducing backtracking in order to be able to search all possible scenarios a dialogue can take. We also discussed the implementation and the difficulties that were resolved. However, the method discussed in this chapter has its limitations. In this section we address those limitations in order to reach a better understanding of them and of why weren't those problems resolved.

The first limitation we encounter in this design is that of resolving all possible deadlock situations. Deadlocks will arise during backtracking when reaching a disjunction of terms. If the next atomic term to be selected was not of the form $M \Leftarrow A$, then there is no problem. The problem occurs when the agent backtracks and reaches a point where it is waiting for a message (the term is of the form $M \Leftarrow A$). Section 3.2.1 presented a partial solution to this problem. The solution solves the problem when all nodes following that disjunction are of the form $M \Leftarrow A$. Hence, if one of them has already failed, the agent can send a failure message notice back to the sender. However, if the nodes are a collection of terms of the form $M \Leftarrow A$ and $M \Rightarrow A$, then it could become extremely complicated to know what failed message to send back to the sender. The complexity of this issue depends on the complexity of the sub-tree of the failed path.

Nevertheless, this limitation isn't a critical one since protocol engineers can overcome this limitation by adding slight variations to the protocol. The simplest way is to engineer the protocol in such a way that the message actions at each tree level are of one form, either $M \Leftarrow A$ or $M \Rightarrow A$. Another similar engineering limitation is the order of the disjunction of terms. The search technique used to search the dialogue tree is a depth-first search. Hence, this should be kept in mind when writing protocols, especially to ensure that the terms in a disjunction are written in parallel.

Overcoming the above limitations is a comparatively easy task for engineers. However, a bit more challenging limitation is faced when common knowledge might need to be changed when backtracking occurs. As described in chapter 2, common knowledge is the knowledge needed for a specific protocol. Along the run of the protocol, common knowledge might be changed with the use of constraints. Clauses defining common knowledge may be asserted or retracted. When we backtrack, we do not revert all changes that already took place in the common knowledge. Although this can be implemented, it will result in a messy solution. Moreover, when backtracking, not all knowledge might need to be reverted; on the contrary, in some cases, knowledge *should* be preserved. And if we did want to revert knowledge, we might face the problem of constraints not being saved when a term is marked as closed. Tackling the latter issue, we notice that constraints might get very large as in the next example in section 3.3. Saving them might in most cases be a waste of space. In order to keep our solution neat and tidy, and in order not to face the problem of when and which constraints should be saved and which part of the common knowledge should be reverted, an alternative option would be to construct the protocol in a way that backtracking shouldn't affect a change in the common knowledge. This could be achieved by distinguishing between the used knowledge of different paths wherever needed. An example on that is presented in the following section (Section 3.3).

All of these engineering limitations could be overcome, as we will see from the example in the next section. However, there remains a performance issue concerning the type of search that is implemented. The depth-first search is definitely not a very efficient search technique and, as we saw earlier, it also imposes some limitations. But the traditional depth-first disadvantages, such as having an infinite path or loop, is not possible in such dialogue protocols unless explicitly defined by the protocol. That is because the expansion of a protocol is made step by step. And since our protocols are dialogue protocols, the expansion will eventually have to stop until it receives some additional info from the agent it is communicating with. The only way expansion can loop forever in a certain path is if it was explicitly defined by the protocol that an agent should loop forever doing a certain action. For example, to loop forever sending a message M to a certain agent A, or to loop forever doing *null*. But since we're dealing with communication protocols, infinite loops in these protocols become meaningless, since they imply that an agent is performing some actions independently without communicating with others. Moreover, as in any other language, if one knows the rules of the language, one might be able then to predict the results of the code.

This topic triggers the efficiency issue of the depth-first search technique. Paths will have to be tested from the left-most path towards the right-most path. This means that the search will be mostly inefficient in cases when the successful path is the right-most one. This raises the question: Can a different and more efficient search technique be implemented?

In such search trees which represent a dialogue between agents, each node represents an action to be taken by the agent. In most cases, this action is either to send or to receive a message. Hence, if other more efficient search techniques, like the breadth-first technique, were to be implemented, this will result in a massive increase in the number of messages sent back and forth between agents and the expansion will be a considerably complicated task. Unfortunately, this will be outside the scope of this dissertation.

We may conclude that despite the above mentioned limitations, this method offers a neat and formal solution for backtracking when constraints failure occur. The limitations are not critical, since they can be overcome by the protocol engineers, and the methodology is a simple and clear one.

3.3 Example and Analysis

The following is a practical dialogue protocol example that was written for further testing and illustrating this methodology. The scenario is a typical e-commerce scenario where agent B, the buyer, is interested in buying an item I from the seller S.

```
a(buyer(I,S),B) ::=
    inquiry(I) ⇒ a(seller,S) then
    a(buyerSt2(I),B).

a(buyerSt2(I),B) ::=
    (
        inquiring_preference(T) ⇐ a(sellerSt2(B,I),S) then
        (
            preference(T,P) ⇒ a(sellerSt2(B,I),S) ← preference(I,T,P)
            or
            preference(T,null) ⇒ a(sellerSt2(B,I),S)
        ) then
        a(buyerSt2(I),B)
    )
    or
    a(buyerSt3,B).

a(buyerSt3,B) ::=
    suitable_items([I1|_]) ⇐ a(sellerSt3(B,I),S) then
    reserve(I1,B) ⇒ a(sellerSt3(B,I),S) then
    waiting_for_payment(I1) ⇐ a(sellerSt3(B,I),S).




a(seller,S) ::=
    seller_of(I) ← inquiry(I) ⇐ a(buyer(I,S), B) then
    null ← item_topics(I,AllTopics) and
        assert(topicsToInquire(AllTopics)) and
        assert(preferences([])) then
    null ← assertz((compute_results([], I, _, AllItems) :-
            acceptable(I, AllItems))) and
        assertz((compute_results([[T,P]|[]], I, null, NewResults) :-
            acceptable_title(I, List), acceptable(I, AllItems),
```

```
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, AllItems), NewResults))) and
        assertz((compute_results([[T,P]|Tail], I, null, NewResults) :-
            acceptable_title(I, List), acceptable(I, AllItems),
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, AllItems), OldResults2),
            compute_results(Tail, OldResults2, NewResults))) and
        assertz((compute_results([[T,P]|[]], I, OldResults, NewResults) :-
            acceptable_title(I, List), \+ OldResults = null,
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, OldResults), NewResults))) and
        assertz((compute_results([[T,P]|Tail], I, OldResults, NewResults) :-
            acceptable_title(I, List), \+ OldResults = null,
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, OldResults), OldResults2),
            compute_results(Tail, OldResults2, NewResults))) and
        assertz((set_search(_, [], []))) and
        assertz((set_search(V, [V|T], [V|R]) :- set_search(V, T, R))) and
        assertz((set_search(V, [X|T], [_|R]) :- \+ X=V, set_search(V, T, R))) then
    a(sellerSt2(B,I),S).

a(sellerSt2(B,I),S) ::=
    (
        inquiring_preference(T) ⇒ a(buyerSt2(I),B)
            ← retract(topicsToInquire([T|RT])) and assert(topicsToInquire(RT)) then
        preference(T,P) ⇐ a(buyerSt2(I),B) then
        (
            null ← (\+ P=null) and retract(preferences(OldPref)) and
                assert(preferences([[T,P] | OldPref]))
            or
            null ← retract(preferences(Old_list)) and
                select([T,_],Old_list,New_list) and assert(preferences(New_list))
        ) then
```

```
        a(sellerSt2(B,I),S)
    )
    or
    (
        null ← topicsToInquire([]) then
        a(sellerSt3(B,I),S)
    ).

a(sellerSt3(B,I),S) ::=
    suitable_items(Results) ⇒ a(buyerSt3,B) ← preferences(Preferences) and
        compute_results(Preferences, I, null, Results) then
    reserve(I1,B) ⇐ a(buyerSt3,B) then
    waiting_for_payment(I1) ⇒ a(buyerSt3,B).
```

The protocol is initiated by the buyer B. B initiates the protocol run whenever it has a certain item I and a seller S in mind. It then sends an inquiry on that item to the seller S (*inquiry(I)* ⇒ *a(seller,S)*). After that step the buyer takes another role, buyer_state1.

At that point, this is all the buyer can do. On the other side of the dialogue, agent S, the seller, receives B's inquiry on item I. However, there is a constraint on that inquiry. The agent can only accept it if it is truly the seller of such items. This becomes the first possible node of failure. If this constraint is satisfied, then the seller proceeds to make sure that it has all the knowledge needed in that domain and asserts some additional knowledge that will be needed for this specific protocol run. Then the seller will take the role seller_state1. This will move the dialogue into the next state.

The first part of the dialogue was concerned with making sure that the seller is the correct seller and that it has all the knowledge needed to assist the buyer in buying its item. The second state of the dialogue is when the seller tries to know the buyer's preferences in some areas in order to check if it does have the suitable item needed. Hence, in this state of the dialogue, both the buyer and the seller will enter a loop that results in the seller knowing the buyer's preferences. At the beginning of the loop, the seller sends a question inquiring about the buyer's preferences concerning a certain area. For instance, if the seller was questioning the buyer's budget, it would send it *inquiring_preference(budget)*. The seller then stops, since it cannot complete its loop before receiving a message from the buyer. The buyer receives the seller's inquiry and replies back with the appropriate answer, either *preference(budget,something)* or *no_preference(budget)*, depending on its beliefs.

When the seller receives a buyer's preference, it either loops back again to inquire about some other topic or it exits the loop by computing the result of acceptable available items. The latter case takes place when the list of preferences to inquire about is empty. Similarly, the buyer exits its loop when it receives a list of available items instead of a message inquiring about some other preference.

The last stage in the dialogue is the *reserve(X,B)* and *waiting_for_payment(X,P)* messages that confirm and end the transaction.

3.3.1 Testing

The context used for testing the above protocol was the selling of 'cars'. For that the seller had to have some knowledge about cars. The following is an example of the beliefs the seller had:

```
seller_of(cars).
item_topics(cars, [budget,manufacturer,color]).
acceptable_title(cars, [budget, manufacturer, color]).
acceptable(cars, [ [10000, bmw, black],
                   [9000, mer, red],
                   [7000, bmw, grey],
                   [7000, bmw, yellow] ]).
```

Similarly, the buyer also needed some knowledge of its own preferences:

```
preference(cars, budget, 1000).
preference(cars, manufacturer, bmw).
preference(cars, color, black).
```

The next page illustrates what the dialogue tree looks like for the above special case. Note that constraints are not added to this picture for the sake of simplification. However, nodes with proaction or reaction constraints are underlined.

Now let's look at the possible failures that could occur in this protocol. If failure occurred during the very first steps of the protocol, i.e. if agent S is not the appropriate seller agent or if it fails to acquire the needed knowledge in order to sell cars, then the dialogue fails. Since the failed term is of the form $M \Leftarrow A$, then the seller sends the failed message back to the buyer.

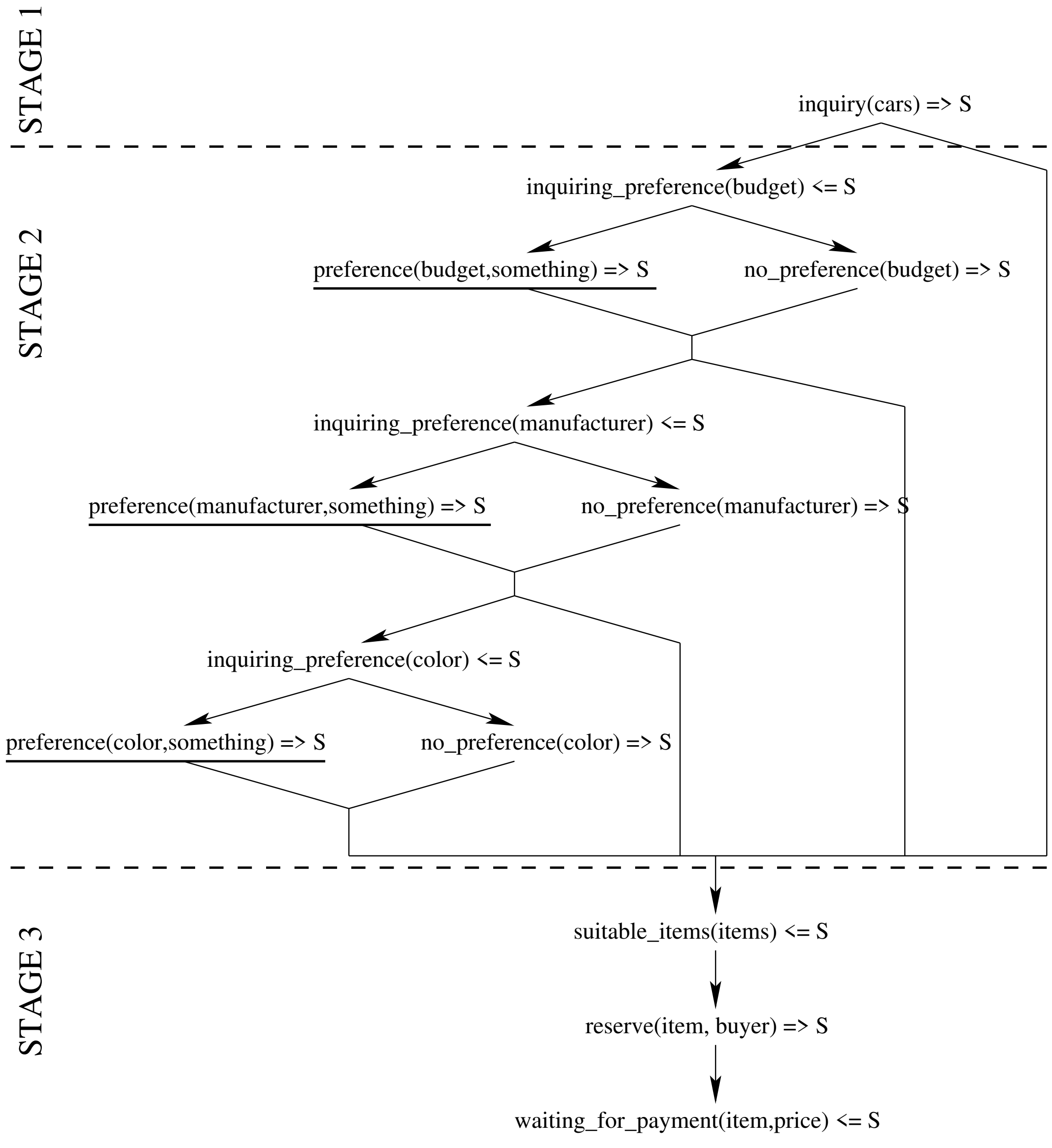

Figure 3.5: the buyer's dialogue tree

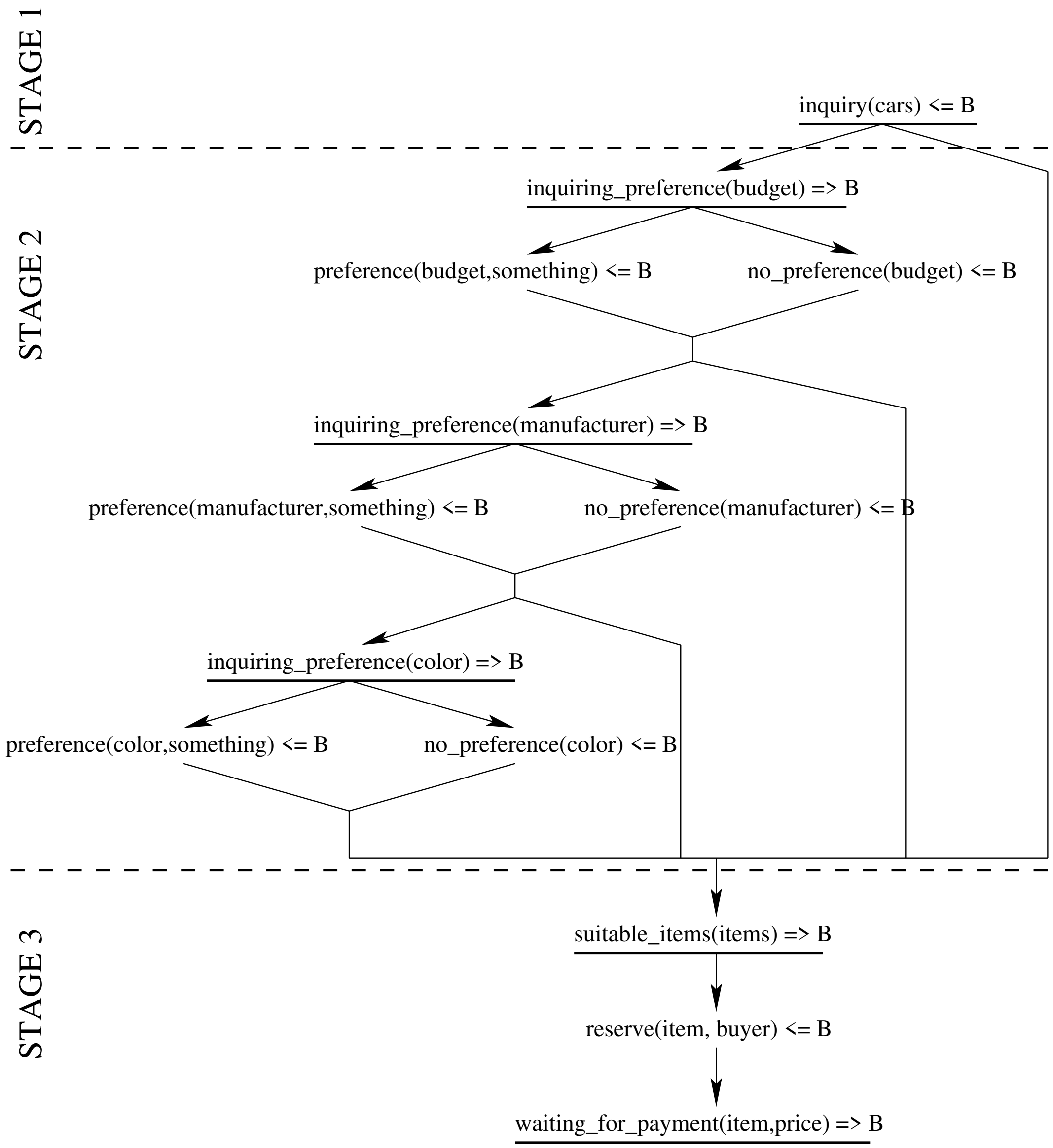

Figure 3.6: the seller's dialogue tree

The buyer tries to backtrack, but this dialogue protocol doesn't give the buyer any options if it selected the wrong seller. Hence, the dialogue fails.

The second possible failure could occur at the beginning and throughout stage 2 of the protocol. This is whenever the seller needs to inquire about a certain preference. If there are no more preferences to inquire about, then the seller backtracks and selects the path that computes the results to be sent to the buyer.

On the other hand, failure also occurs when the buyer doesn't have a preference for the seller's inquiry. In this case, the buyer sends the *no_preference(P)* back to the seller.

Another failure that might occur at the beginning of stage 3 of the protocol is when the seller fails to compute a non empty list of suitable items. Hence, if no suitable cars were found, the dialogue will backtrack. This backtracking will result in sending a failure message notice to the buyer informing it that its last mentioned preference failed. In the above example, it will inform the buyer that its preference for 'color' has failed. This will then force the buyer to backtrack and send a *no_preference(color)*.

Note that if again no suitable cars were found, this will force the seller to backtrack and send the buyer a failure notice of the preference sent concerning the 'manufacturer'.

This scenario is repeated until a suitable car is found or the dialogue fails. However, in our specific example above, eventually the dialogue will always succeed. That is because whenever a preference is not suitable, a *no_preference* message is sent. Hence, in the worst case scenario, the list of suitable cars will be the list of all available cars.

3.3.2 Observations

It is important to note that the protocol described above is a general purpose protocol. The protocol is written in a way that it suits any item, whether cars or not. And for each item, each different seller might have his own questions to ask the buyer. For instance, one seller might inquire about the color of the car while another might not. All of those details are specific to a certain item and/or seller since they are extracted from the seller's own beliefs.

Moreover, room is made for additional modifications of the protocol. For instance, the buyer now selects the first car from the list of suitable cars. Changes could be made to select the car according to preferences.

Another modification that could easily be appended to the following version is adding a constraint to sending a *no_preference* message. For instance, the buyer might want his budget constraint to be a strict constraint. Hence, if no cars were found within that budget, the buyer prefers not to buy. And so, many other modification can easily be added.

However, it should be noted that the ontology used in this protocol needs to be the same as that used in both the seller and buyer's beliefs.

3.3.3 Limitations Revisited

Reconsidering the limitations mentioned in section 3.2, it is worthy to note that the limitations mentioned there were easily resolved. The result was similar to backtracking in other logic programming languages, such as Prolog. Keeping in mind how the language works and its limitations, one should be able to construct a dialogue protocol that does not end up in a deadlock situation when some constraint fails.

As for the limitation of being unable to revert changes in common knowledge, it is clear from the example above that this is again possible through careful engineering. The trick in such problems is not to change clauses when backtracking might require to revert the change. Instead, make use of the possibility of asserting new predicate definitions. Then engineer your knowledge in a way that makes use of those asserted predicates to catch changes instead of making a change permanent. For instance, in the example above, instead of re-computing the list of suitable cars every time a new preference is added, the computation takes place at the very end. If something fails and backtracking is considered, the protocol is engineered carefully to easily obtain the old knowledge needed.

3.3.4 Final Remark

What we have defined in this chapter is a formal method to overcome constraint satisfaction failure in distributed dialogues by searching for other possible dialogue scenarios. This was possible by inducing backtracking. We notice there was no need for agents to communicate failed terms on a regular basis for synchronisation. The agent which encounters such a failure automatically backtracks. The other agents involved in the dialogue will automatically readjust their positions accordingly. The only case when failure is reported, is when backtracking will result in a deadlock situation where each agent is waiting to hear from the other. To overcome these situations, agents need to communicate failure messages in order to force the other agent to backtrack.

This methodology has its limitations. However, with careful engineering of the protocol, it appears to be possible to overcome these limitations.

Chapter 4

Constraint Failure and Negotiation

Chapter 3 proposes a solution to constraint failure in agent's dialogues: backtracking to investigate further possible dialogue solutions. The e-commerce dialogue example in chapter 3 illustrates the possible use of such a solution. However, considering the same example, we notice that a more realistic solution could still be implemented to better address such failure. Let's consider the case when the seller couldn't find the car with the customer's specified constraints. In scenarios similar to this, negotiation might be necessary. For instance, the buyer might decide to soften its constraints and give other acceptable values instead of simply deciding whether it can either accept *any* other value or *none*!

This chapter discusses the possibility of constraints negotiation. It starts with some background on negotiation theories in general. Then moves to discuss the possibility of implementing negotiation in the case of constraints failure of distributed dialogue protocols. Details of the implementation are presented and the example in the previous chapter is used again to test the negotiation protocol.

4.1 Overview: Negotiation Theories

One of the most important properties of agents is *autonomy*. This property implies that an agent "can act without the direct intervention of others and has control over its own actions and internal state" [KJ99]. We know that agents have their own goals and that they are expected to take initiatives in order to meet their goals. However, in multi-agent systems, goals are achieved by the help of other agents. At many times, the actions an agent needs to take in order to fulfil its goal includes asking other agents to take other actions.

But what happens when the means of one agent to achieve its goals con¤icts with the beliefs of another involved agent? “Because the agents are autonomous and cannot be assumed to be benevolent, agents must in¤uence others to convince them to act in certain ways, and negotiation is thus critical for managing such inter-agent dependencies” [BdL$^+$99]. Agents will have to negotiate over the issue in question until they could persuade each other to change their beliefs and reach an acceptable solution.

Negotiation could be defined as a “distributed search through a space of potential agreements” [JFL$^+$01]. The following figure better explains this idea. The grey areas are the agents’ initial region of acceptability. The areas with rigid boundaries are the current regions of acceptability. The element marked ‘O’ is the current offer, while the ‘Xs’ are previous offers.

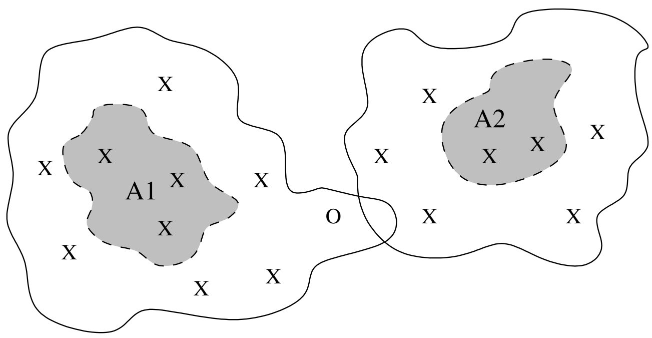

Figure 4.1: the space of negotiated agreements (a modi£ed version of [JFL$^+$01])

Hence, in theory, agents will communicate trying to change other agent’s beliefs that would result in changing their regions of acceptability in the direction of reaching a common point in the space of potential agreements.

In practice, how can we define this communication (negotiation)?
The minimum requirements for negotiation is to be able to propose acceptable parts of the search space and to respond to such proposals by either acceptance or rejection. However, with a simple accept or reject reply, the proposer has no idea in which direction of the search space should it move. This could not only be time consuming and inefficient, but it might also lead to an infinite loop. The agent that responds to a proposal will need to include more information in its response in order to help direct the proposer.

To do that, we can define two forms of replies to proposals could take two different forms: *critiques* and *counter-proposals* [PSJ98]. Critiques could be comments on those parts of the proposal that need further modifications and might also indicate the direction of modification. For example, consider the following dialogue between agents A and B:

A: I propose that I will provide you with service Y if you provide me with service X.
B: I'm not interested in service Y.

On the other side, a critique could simply be an acceptance or rejection with further details:

A: I propose that you provide me with service X.
B: I don't have the privilege to provide service X.

Critiques usually cause the proposer to send another modified proposal. It is important to note that the more information a critique holds the better the probability is for reaching an agreement faster.

In addition to critiques, agents might also reply to proposals with another proposal. These are called counter-proposals. Counter proposals might extend the initial proposal, as shown in the following example:

A: I propose that you provide me with service X.
B: I propose that I will provide you with service X if you provide me with service Y.

Or, counter proposals might only amend part of the initial proposal:

A: I propose that I will provide you with service Y if you provide me with service X.
B: I propose that I will provide you with service Y if you provide me with service Z.

In addition to *critiques* and *counter-proposals*, *meta-information* could also be sent. Meta-information is usually used to justify the response or proposal. Meta-information could also help persuade the other agent to change it's beliefs. For instance it could offer a reward, indicate a threat or appeal [SJNP97].

With the above defined negotiation's generic model, the figure next page best describes the resulting negotiation protocol. The process begins when an agent '*a*' proposes '*X*' to an agent 'b' (*proposal(a,b,X)*). This automatically moves the protocol to state 1. At this point, either agent '*a*' makes a second proposal (protocol stays at state 1), or agent '*b*' makes either a counter proposal *proposal(b,a,X)* (protocol moves to state 2), or a critique of the initial proposal (protocol move to state 3). The process iterates until one of the agents sends an accept or reject. These are represented by the grey final states 4 and 5, respectively.

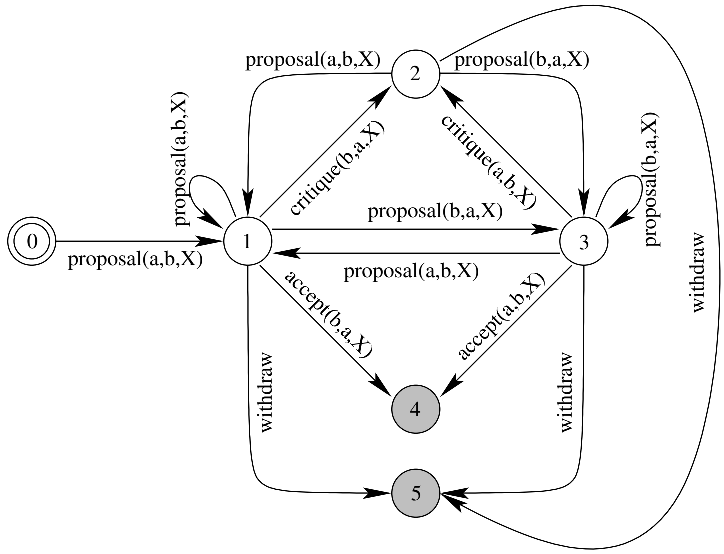

Figure 4.2: the negotiation protocol [PSJ98]

Negotiation could get very complex. Different negotiation rules might be needed for different scenarios. However, the above tries to generalise the negotiation protocol by giving general permissible states and state transitions for negotiation. In what follows, we will introduce the negotiation protocol that we built for our distributed dialogue system.

4.2 Constraints Failure and Negotiation

How could a dialogue recover from failure when constraints are not satisfied? Negotiation might be a solution to many cases. The agent might decide to negotiate constraints that failed with other agents in order to permit the dialogue to proceed and complete successfully.

In our negotiation model, critiques and counter-proposals do not exist. Since we are negotiating constraints, messages could either hold a set of constraints, a set of meta-information, or both. To better understand why we did this, let us consider the critique example above. Agent B’s responses could be considered as a meta-information itself, since it is informing the proposer the reasons behind the failure of its proposal. The counter-proposal examples, on the other hand, could be embedded in our dialogue protocol as follows:

First Example,
A sends B: *B_provides_A* = X
B sends A: *B_provides_A* = X and *A_provides_B* = Y

Second Example,
A sends B: *B_provides_A* = X and *A_provides_B* = Y
B sends A: *B_provides_A* = X and *A_provides_B* = Z

where *B_provides_A* and *A_provides_B* are the constraints being negotiated.

Now the question is how to implement the negotiation protocol?
The first thing to bear in mind is that not all constraints in a dialogue protocol are negotiable. Hence, the negotiation protocol should not be implemented automatically whenever failure occurs. It should be specifically called when certain constraints need negotiation. Secondly, Figure 4.2 above defines the protocol as a set of states that the negotiators pass through when performing the negotiation actions, where these actions are usually of the form of message passing. With our modifications to the negotiation model above, the resulting negotiation protocol will look similar to that in Figure 4.2, except that state 3 is now deleted. Hence, it looks a perfect candidate to be implemented with a dialogue protocol. With the use of backtracking, the protocol engineer would decide when should a constraint be negotiated by simply backtracking to the negotiation protocol as needed (a more practical example is given in section 4.3).

But how general could the protocol be? As we mentioned before, negotiation may get extremely complex. Different protocols might be needed for different scenarios. Section 4.1 tries to categorise the huge range of negotiation dialogues. Some of these may include cases where the issues (or constraints in the case of our dialogue protocols) negotiated might change. For instance, consider the case when agent A proposes that agent B would provide it with service 'X', and agent B replied by proposing that it will provide service 'X' if A provides it with service 'Y'. In such cases, constraints being negotiated might themselves be subject to change by either being deleted or introduced. Our negotiation protocol assumes that the set of constraints being negotiated is a strict *un-negotiable* set. Values of these constraints are the only thing being negotiated. Moreover, if there is any need for change within the set of negotiated constraints, then this will have to take place outside our defined protocol.

Another main assumption our negotiation protocol takes is that only two agents are involved in any one negotiation dialogue at a certain time. Nevertheless, agents may still be engaged in several dialogues simultaneously.

The next section gives an exact definition of our negotiation protocol. Following that are some implementation details and a practical example.

4.3 The Negotiation Protocol

Figures 4.3 and 4.4 illustrate the negotiation protocol implemented. One agent initiates the protocol by asking the other whether it agrees to negotiate (*soften_constraints(Error)* $\Rightarrow$ *B*). If it does receive an agreement (*negotiation_status(accepted)* $\Leftarrow$ *B*), then it inquires about the topic's ranking and the acceptable values for the topic being negotiated (*inquiring_topics_ranking (AllTopics)* $\Rightarrow$ *B* and *inquiring_topic_preference(T, data_cat, preference)* $\Rightarrow$ *B*). After receiving answers for its inquiries, it moves to search for applicable results (Stage 3 of figure 4.3). It exits stage 3 either when applicable results are found, or when failure occurs. In the latter case, it proposes other acceptable results (*only_acceptable_values(OldNResults)* $\Rightarrow$ *B*) which may or may not be accepted by the other agent (the negotiant).

The stages of figures 4.3 and 4.4 are discussed below. (For further reference, the code of the negotiation protocol is available in Appendix A)

4.3.1 Stage 1: Establishing Negotiation

The negotiation protocol is called when certain constraints need to be negotiated. However, negotiation is triggered after some failure to fulfil certain constraints. Hence, the protocol first starts by examining the type of error that occurred and resulted in the failure. Then, depending on this error, the negotiant may or may not decided to proceed with the negotiation. To further understand the need for this, let's consider the example when an agent seeks the help of another agent to build a PC. The agent should have certain requirements. It would then start a dialogue with an appropriate agent informing it of the requirements to be fulfilled. The other agent then tries to fulfil those requirements. If failure occurs, it asks the first agent whether it's interested in negotiating and informs it of the reason behind the failure. Now let's say that the failure was due to manufacturing constraints. Manufacturing constraints could include component placement, component type, etc. In such cases, the negotiant *should* accept negotiation, because otherwise it will not be able to reach its goal even if it seeks other agents' help. We classify

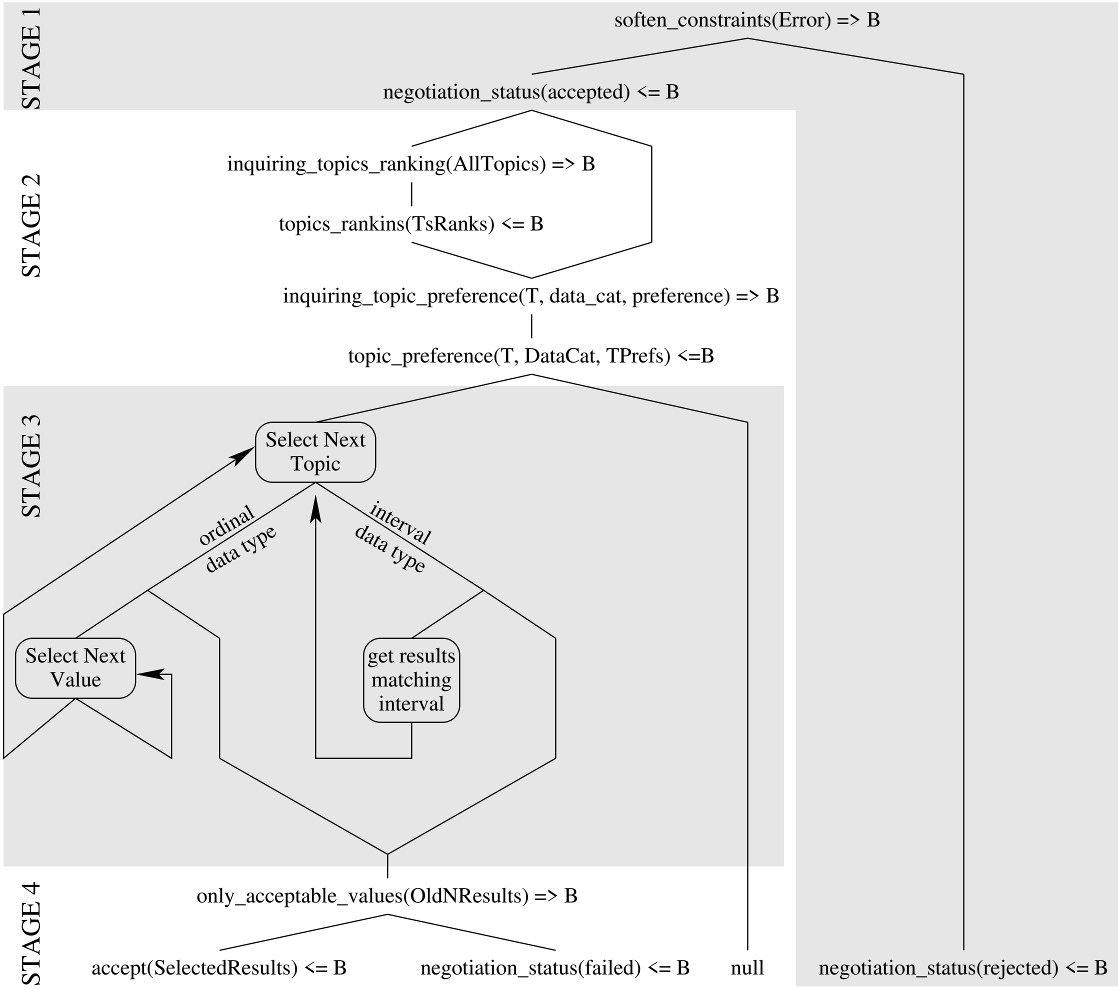

Figure 4.3: the negotiation initiator dialogue tree

those constraints as *hard constraints* that *need* to be negotiated. Another possible reason behind failure might be that a certain item is out of stock, or maybe the agent being contacted doesn't support this specific requirement. We classify these kind of constraints as *soft constraints*, and it is completely up to the negotiant agent to decide whether it will proceed with negotiation or not. For instance, if the error description was *'out of stock'*, then the agent might decide to reject negotiation and prefer to wait for more supplies to arrive. However, in the latter case, it might either decide to negotiate or again end negotiation and try to find another agent that could fulfil its requirements.

The negotiation protocol's very first step is to try to establish negotiation. Now this step could get as complex as the agents want it to get and its complexity has nothing to do with the negoti-

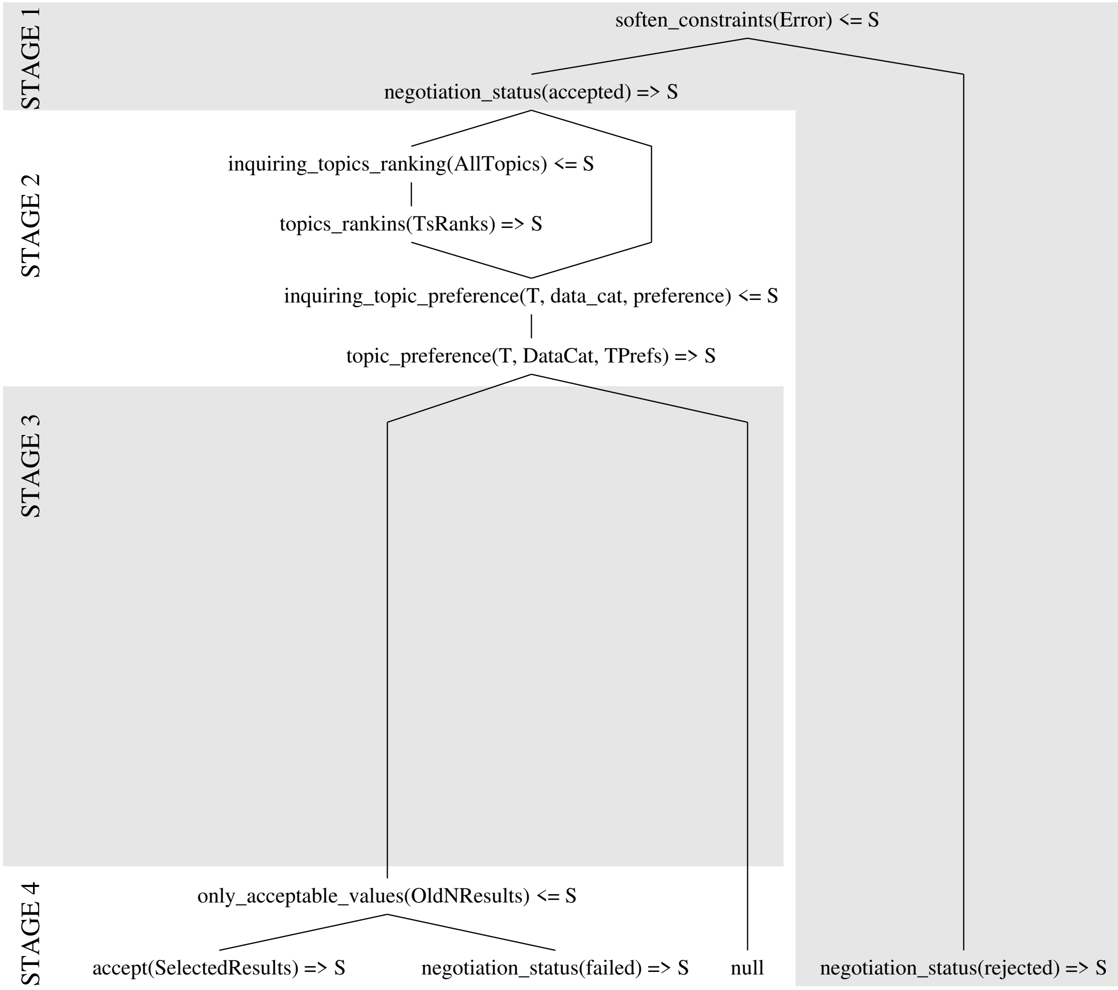

Figure 4.4: the negotiant's dialogue tree

ation protocol itself. In the simplest case, an agent might have no knowledge whatsoever on the reason of failure and hence always result with an *'unknown error'*. In a more complex scenario, the agent might be connected to a complex knowledge base that would give the exact details explaining failure. This would very much help the negotiant to achieve a better decision when accepting or rejecting negotiation. In such cases, the knowledge base should always be up-to-date, and it should offer the smallest detail that could help the negotiant make a better decision. Similarly, on the other side, the negotiant might either decide to always accept negotiation, or it could have a more complex opinion on which errors are acceptable. Moreover, a GUI could be used to display an unrecognised error to the negotiant in order to also reach a better decision (unfortunately, a GUI was not available at the moment of writing this dissertation).

Hence we see that the complexity of the first step of negotiation, establishing negotiation, does not depend on the negotiation protocol itself, but on the agents involved as well as the dialogue scenario and the type of constraints that result in failure. Different dialogue scenarios and different constraints might have different considerations to this first step.

4.3.2 Stage 2: Preference Inquiry

At the very beginning of negotiation, and if several constraints are subject to negotiation, then the agent that initiates negotiation will need to inquire about the preference of constraints. The other agent will need to rank the constraints according to their importance. For instance, if we take a look at the example in section 3.3, the constraints that might need to be negotiated are the *budget*, *manufacturer* and *color*. Color won't be as important to the buyer as much as the budget is. It might prefer to change its color instead of its budget. Therefore, it should inform the seller of the ranking of these constraints (for example, budget=1, manufacturer=2, color=3).

The above takes place only once and at the very beginning of negotiation. However, the protocol is called once for each constraint being negotiated. For every run of the protocol, the agent that initiates the protocol will also need to inquire about the alternative acceptable preferences for a given constraint.

Now alternative preferences might be categorised differently depending on the type of the constraints being negotiated. The alternative preferences for the constraint *budget* might be an *interval* of values. For instance, the buyer might accept any value between £5,000 and £10,000. For other constraints, like *manufacturer*, it could be a list of other acceptable manufacturers with a ranking indicating the order of preference. This data category is known as *ordinal*.

The protocol deals differently with each of these categories (as we will see in the following section). However, for the time being, only those two categories are being dealt with. In the future, if the need for additional categories arises, like the combination of the *interval* and *ordinal*, then that will have to be introduced. Such a task will be a simple task. The protocol part to deal with the new category will have to be written in a separate agent definition. Then a link to this definition will have to be introduced in the '*a(select_topic(T, OldNResults, B),S)*' agent definition of the protocol (check appendix A for reference).

4.3.3 Stage 3: Searching for a Solution

As we have mentioned above, the protocol is called once for each constraint being negotiated. So if we take a look at the example in section 3.3, we notice that if failure occurs, the constraint *color* will be negotiated to find other acceptable results. If this fails, then *manufacturer* will be negotiated, and so on. It's also important to note that constraints that have been given a *no_preference* value from the beginning will not be negotiated.

Now when searching for different values for the constraint *color*, for example, the protocol first makes sure that the un-negotiated constraints do succeed in finding acceptable results. For instance, the un-negotiated *manufacturer* and *budget* do have acceptable results. Because if they don't, then it's useless to search for other acceptable color values, since the protocol will eventually backtrack again to negotiate the *manufacturer* constraint. Hence, only if un-negotiated constraints do have successful results, then the protocol will further search for solutions to the constraints being negotiated. Moreover, it will search for its solutions among the successfully results of the un-negotiated constraints only.

Now the question is how does the protocol search for acceptable results with the given preferences? Let's say that the topics being negotiated so far are the *color* and *manufacturer* of the earlier example. They rank 3 and 2, respectively. Let's also assume the following to be the acceptable values of each:

– Manufacturer: *Mercedes* with rank 1, or *BMW* with rank 2
– Color: *Black* with rank 1, *Dark Blue* with rank 2, or *Grey* with rank 3

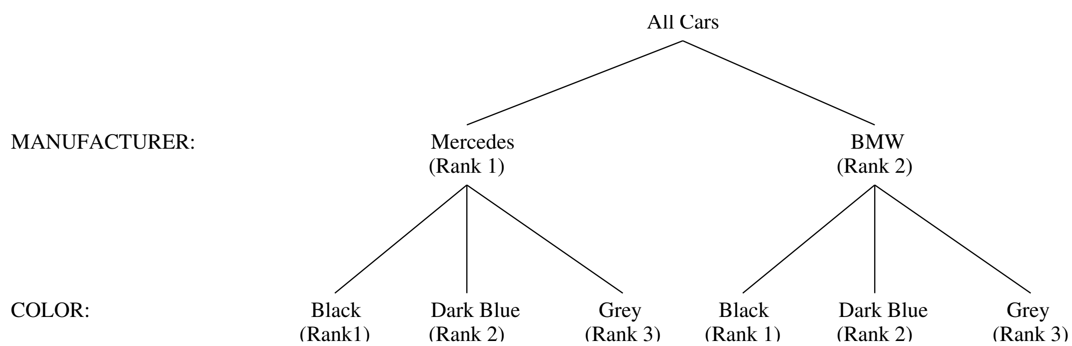

Figure 4.5: searching for a solution for the negotiated constraints *manufacturer* and *color*

The above picture explains the order that values will be tested for. The protocol will first try to check if a Mercedes car is available, since Mercedes is the value with the highest rank to the constraint with the highest rank, the *manufacturer* constraint. If this succeeds, it will proceed down the tree to try and find whether a ‘black Mercedes’ is available, since *black* is the highest ranked value of the *color* constraint. If *black* fails, then *dark blue* is tested and so on. If all colors fail for a Mercedes, then a BMW manufacturer is selected, and colors are again tested one by one, starting with the highest ranked value to the least ranked.

The above example is a simple example. Constraints with *interval* typed values might also be included. However, the concept is still the same. The protocol will try its best to get the highest ranked value for each constraint, giving priorities for constraints with higher ranks. Hence we can look at the above tree as being part of the actual dialogue tree. Moreover, since such branches change with the different constraints being negotiated and their different acceptable values, our protocol’s tree will also be dynamic.

4.3.4 Stage 4: Results

As we have seen above, the protocol moves from one constraint to the other and from one value (for a given constraint) to the other. Now let’s consider the case when the protocol is solving constraints of an interval data type. If this fails, then the protocol will suggest other available solutions, before finalising the failure of negotiation. This is very much useful, especially when available values are very close to the limits of the interval. However, in general, it might still be a good idea to close a failing negotiation dialogue by giving the last possible options. The negotiant might find something appealing among them, and save the negotiation from failure. Hence, even when failure occurs when negotiating constraints of an ordinal data type, it might still be a good idea to conclude with the list of suggested options.

But the question is what will the list of suggested options be?
We definitely wouldn’t want to include everything available! Let’s take the same e-commerce scenario to make things clearer. Only that we’ll now increase the number of constraints being negotiated in order to better illustrate this issue. Let the customer agent’s initial request to be as follows: Budget = £10,000, VehicleType = Sports Car, YearModel= 2000, Manufacturer = Mercedes and Color = Red.

Now let’s say that the protocol is half way through negotiating the *color*, *manufacturer* and *year model* constraints. It finds an acceptable year model, say 1999. However, it fails when it gets to the *manufacturer*. Instead of letting it fail, the protocol would suggest the available cars, if any, with the given *un-negotiated* constraints values (i.e. Budget = £10,000, VehicleType = Sports Car) and the successful *negotiated* constraints values (i.e. YearModel= 1999). Hence, it would be suggesting the available results that are closest to what the negotiant wants. If the negotiant accepts any of the results, then the negotiation is saved from failure.

It is important to note that in the case when several partial *negotiated* constraints succeed, the selected results are those values with the highest rankings.

4.4 Example and Analysis

In this section we discuss the tests and results made on the above proposed method of negotiation. For testing purposes, we have used the same example as that in section 3.3. This has been modified, as shown in the following figure, to make use of negotiation. The figure below only shows the seller's side of the dialogue. Similarly, the buyer's side will be parallel to this. For further reference, the modified code of the e-commerce scenario dialogue protocol is also available in Appendix A.

The added paths are those with the circle in the middle. As we can see in Figure 4.6, the dialogue starts as usual by collecting info on the preferred *budget*, *manufacturer* and *color*. When there are no more topics to inquire about, it tries to compute the results and send them to the buyer within the *'suitable_items(Items)'* message. If this fails, then the dialogue will first backtrack to negotiate the *color* constraint. If this fails too, it will try to negotiate the *manufacturer* constraint as well, and so on, until either a solution is found, or negotiation fails.

4.4.1 Testing and Results

The above have been tested thoroughly to ensure that the design explained in section 4.3 works as intended. In what follows, different cases have been tested to further understand how the negotiation takes place.

First, note that the buyer's requested values for budget, manufacturer, and color are £10,000, Mercedes, and Black, respectively. If negotiation is needed, then the other acceptable values are:

– Budget between £8,000 and £12,000

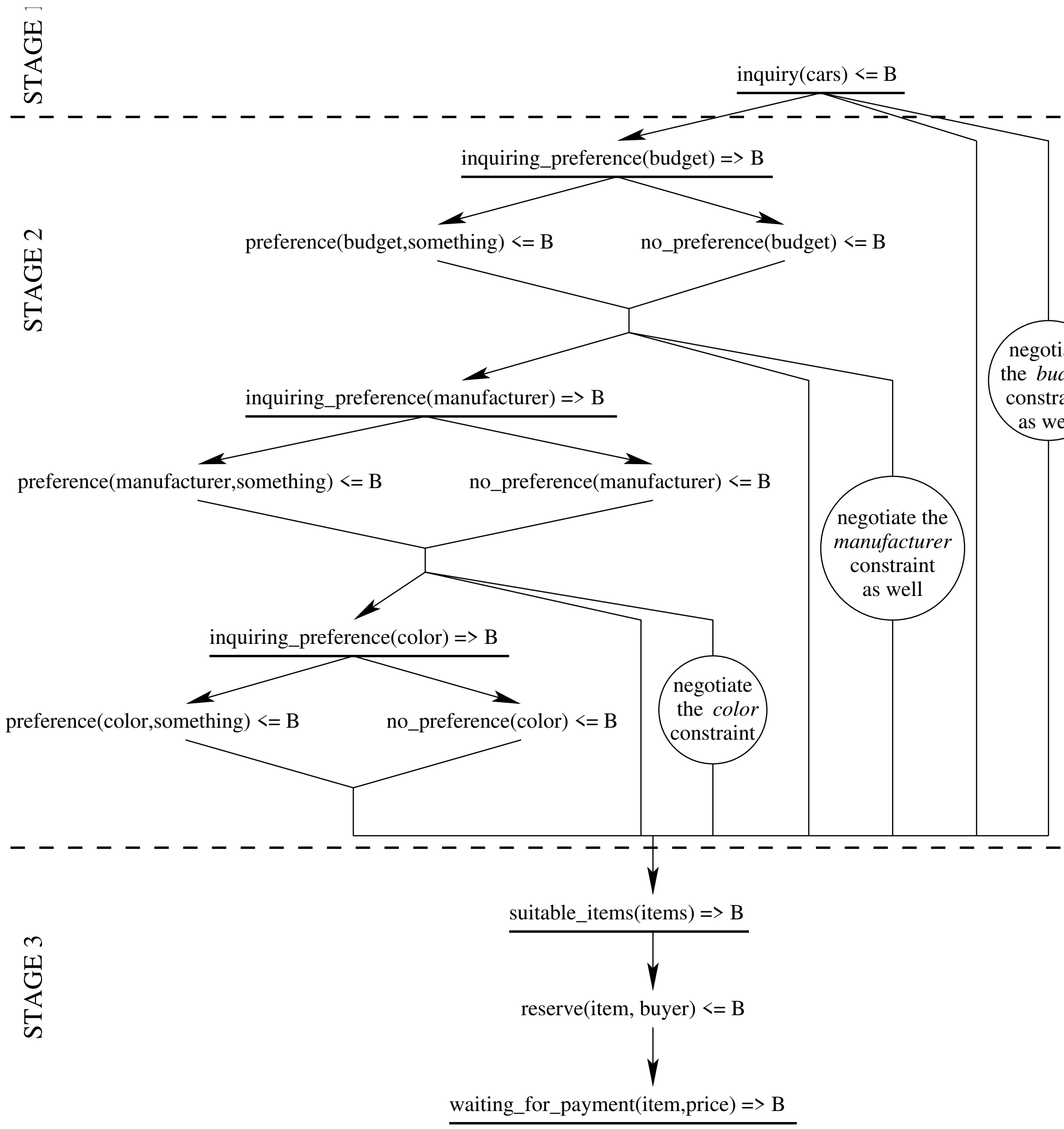

Figure 4.6: he modi£ed version of the seller's dialogue tree

– Manufacturer could either be Mercedes or BMW with ranks 1 and 2 respectively

– Color could either be Black, Dark Blue or Grey with ranks 1, 2 and 3, respectively

Case 0: Let's assume that the list of available cars is:

Car #	Price	Manufacturer	Color
Car 1	£10,000	Mercedes	Black
Car 2	£10,000	Mercedes	Red
Car 3	£10,000	BMW	Grey
Car 4	£10,000	BMW	Yellow

In this case, Car 1 is selected without any need for negotiation.

Case 1: Now, let's assume that the list of available cars is:

Car #	Price	Manufacturer	Color
Car 1	£10,000	Mercedes	Grey
Car 2	£10,000	Mercedes	Red
Car 3	£10,000	BMW	Grey
Car 4	£10,000	BMW	Yellow

In this case, none of the above cars suits the initial requirements of the buyer. The seller backtracks to negotiate the *color* values. It tests whether Mercedes cars priced £10,000 already exist, and it gets 2 results for that, cars 1 and 2. Now it tries to find if any of them will fit the negotiated *color* values. Car 1 succeeds, and it is returned to the buyer as a suitable car.

Case 2: Now, let's assume that the list of available cars is:

Car #	Price	Manufacturer	Color
Car 1	£9,000	Mercedes	Yellow
Car 2	£9,000	Mercedes	Red
Car 3	£10,000	BMW	Grey
Car 4	£10,000	BMW	Yellow

Again, none of the above cars suits the initial requirements of the buyer. The seller backtracks to negotiate the *color* values. It tests whether Mercedes cars priced £10,000 already exist. Since it fails, it automatically backtracks to negotiate the manufacturer constraint.

The seller then tries to find a £10,000 priced cars (i.e. cars 3 and 4). It checks branches of the tree (Figure 4.5) that fit any of the above cars. The search used is a depth first search. That's because it needs to find the results with the highest rankings. In this case, car 3 is the only one that succeeds, and it is returned to the buyer as the only suitable item.

Case 3: Now, let's assume that the list of available cars is:

Car #	Price	Manufacturer	Color
Car 1	£10,000	Mercedes	Yellow
Car 2	£10,000	Mercedes	Red
Car 3	£10,000	BMW	Grey
Car 4	£10,000	BMW	Yellow

Again, none of the above cars suits the initial requirements of the buyer. The seller backtracks to negotiate the *color* values. It again tests whether Mercedes cars priced £10,000 already exist. This time it succeeds, returning cars 1 and 2 as a result. However, when it tries to find acceptable *color* values among those results, it fails!

At this point the protocol doesn't automatically backtrack to negotiate the manufacturer constraint, as above, but it suggests available values (i.e. cars 1 and 2). In Case 2 above, checking for available acceptable *color* values was useless. Hence, it automatically backtracks. However, in this case, negotiating the color constraint has failed since no results were found. Trying to overcome failure, the seller decides to let the buyer know of any available results (cars 1 and 2). The buyer might either accept any of the results and successfully complete the protocol, or it might reject them, forcing the seller to backtrack and continue with the negotiation.

Case 4: Now, let's assume that the list of available cars is:

Car #	Price	Manufacturer	Color
Car 1	£9,000	Mercedes	Black
Car 2	£9,000	Mercedes	Red
Car 3	£9,000	BMW	Grey
Car 4	£9,000	BMW	Yellow

Let's also assume now that at the very beginning of the protocol, the buyer decided that it doesn't have a certain preference for the *manufacturer* constraint.

Hence the initial requirements now are *Budget = £10,000* and *Color = Black*. This fails, so the seller backtracks to negotiate the *color* values. It again tests whether cars priced £10,000 already exist, and it fails.

At this point, the protocol doesn't backtrack to negotiate the *manufacturer* constraint, but backtracks directly to negotiate the *budget*. That's because the buyer already had no preference what so ever for the *manufacturer*.

The seller will now try to find car's whose prices is between £8,000 and £12,000. All of the above cars succeed. So it now tries to get the one with the highest ranking color value, and car 1 is returned to the buyer as the suitable item.

4.4.2 Observations and Remarks

The negotiation protocol discussed above is complex. But it provides results closer to real negotiation.

Let us go back to the search process. The protocol tries to soften constraints one after the other until a solution is found. After softening few constraints, it tries to fulfil them with respect to the constraint's ranking order. Now if this fails, then in order to give the negotiation a final chance to succeed, acceptable results are suggested. Those acceptable results are carefully computed to give the negotiant the closest results to the current given preferences.

This option makes the negotiation dialogue more real. It allows both parties to actually negotiate by permitting each to inform the other of what it has or needs, instead of just having one agent requesting and the other trying to fulfil its requests without actual negotiation.

Another similar case is at the very beginning of the protocol. The negotiant is given the option to accept or reject the negotiation. In this case too, more information is communicated between both parties in order to make the negotiation as realistic as possible, and in order to allow the protocol to reach better results in the shortest time. One agent tries to give as much detailed explanation as possible to the other, to help it better understand the problem that results in making a better decision.

However, as noted in section 4.3.1, this also depends on the agents involved, dialogue scenarios and constraints being negotiated. In cases when such information is given high importance, it is recommended that the agent would be connected to a powerful knowledge base that would provide it with the most updated detailed explanation.

Hence we notice that the proposed negotiation method could become more powerful by making use of other more powerful tools, like a knowledge base. Another important tool that could be used is a graphical user interface. Consider the case when the agent is informing the negotiant of the reasons behind failure. The negotiant receives this data and notices that it is an unrecognised reason. In such cases, a GUI would be perfect to handle these situations, instead of using a default value that would always accept or reject such un-recognised errors.

Yet another important use of a GUI would be at the very end of the negotiation. If negotiation fails and the agent proposes some available results, the GUI might be a better way to deal with those results too.

We have tried to make the protocol as general as possible, and as realistic as possible. The protocol we implemented deals only with cases when two agents are negotiating over a strict set of constraints. One agent would have certain requirements, and the other would try to convince that agent to soften its constraints in order to reach acceptable results. It might also direct the agent by providing it with some advice. However, only with the help of other tools can the protocol reach it's full potential.

4.4.3 Final Remark

In chapter 3, we proposed a formal solution, induced backtracking, to overcome constraints' failure in distributed dialogue protocols. In this chapter we have discussed the importance of negotiation for certain cases of constraints failure. The negotiation protocol proposed employs our 'induced backtracking' method. Without backtracking, the protocol can not backtrack to negotiate other constraints. The potential capabilities of backtracking are especially obvious in the part of the protocol that looks for acceptable results by moving from one constraint to the other and from one constraint value to the other.

Chapter 5

Conclusion

The distributed dialogues proposed did not completely offer what the language presented. When engineers construct definitions of the form *'A or B'*, they expect that if either *A* or *B* *can* be completed successfully, then *'A or B' will* be completed successfully. This is true when *A* and *B* are *atomic terms*. However, with more complex definitions, it becomes hard to backtrack to the appropriate other option.

In this dissertation we have provided a simple methodology to induce backtracking and preserve the full potential of these dialogues. The concept was relatively straight forward. However, the tricky part was to try to cover and resolve the most likely deadlock situations. We have tackled questions like *when* and *how* do we open already closed dialogues? *When* and *how* do we communicate failed terms? These were critical questions needed to provide a smooth coordination. We had to insure that the agents communicating will backtrack together to the same protocol state. This had to be done without the need to send every single term marked as failed. Hence, failed terms were only communicated on crucial occasions: to prevent the dialogue from ending up in a deadlock situation.

However, similar to human dialogues, agent dialogues might also get very complicated. A formal methodology, such as our 'induced backtracking', can not cover all possible dialogue deadlock situations. But engineers could make use of our proposed solution in building dialogues to act as intended. Chapter 3 already discussed the limitations of our solution and how to overcome them.

The result is that we have introduced a method that could be used to solve constraint failure issues. When one path of the dialogue fails, other paths are investigated for possible success. Further more fancy solutions could also be investigated for such failure. But these will most probably be dependent on the type of failure, the dialogue involved, the specific constraints that failed, etc. For instance, on certain occasions, one might be interested in considering patching the protocols to deal with the specific failure involved. At other occasions, it might be more appropriate to negotiate the failed constraints.

In the second part of our dissertation we constructed a negotiation protocol that makes use of our 'induced backtracking' solution and could be used to deal with constraints failure. Again, we tried to make the protocol as general as possible. The protocol we implemented dealt only with cases when two agents are negotiating over a strict set of constraints. One agent would have certain requirements, and the other would try to convince that agent to soften its constraints in order to reach acceptable results. It might also direct the agent by providing it with some advice.

However, as mentioned earlier, the full capabilities of this protocol could only be uncovered with the use of other tools (such as having a *selling* agent connected to a powerful knowledge base or having a *buyer* agent with a GUI). Hence, It would be interesting to consider these additional issues that could help highlight the full potential of our protocol.

As we have seen, different situations require different actions towards constraint failure. In this dissertation we have fixed the backtracking issue in distributed dialogues. This is not only a solution that addresses the constraints failure issue, but also a basic definition of the behaviour of dialogue protocols. We also provided a negotiation protocol that implements the 'backtracking' solution to negotiate constraints. However, the door is also open for much a wider range of further proposals.

Bibliography

[Aus62] John L. Austin. How to do things with words. *Oxford: Oxford University Press*, 1962.

[BdL+99] Martin Beer, Mark dInverno, Michael Luck, Nick R. Jennings, Chris Preist, and Michael Schroeder. Negotiation in multi-agent systems. *Knowledge Engineering Review*, 14(3):285–289, 1999.

[FIP97] FIPA. Agent communication language specification, October 1997. FIPA 97 Specification, Part 2.

[GHB99] Mark Greaves, Heather Holmback, and Jeffrey Bradshaw. What is a conversation policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, at third annual conference on Autonomous Agents*, May 1999.

[JFL+01] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: Prospects, methods and challenges. *Journal of Logic and Computation*, 10(2):199–215, July 2001.

[KJ99] Susanne Kalenka and Nick R. Jennings. Socially responsible decision making by autonomous agents. In K. Korta, E. Sosa, and X. Arrazola, editors, *Cognition, Agency and Rationality*, pages 135–149. Kluwer, 1999.

[PSJ98] Simon Parsons, Carles Sierra, and Nick R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, June 1998.

[Rob] Dave Robertson. A lightweight method for coordination of agent oriented web services. Preprint, available from the author.

[Sea79] John Searle. Expression and meaning: studies in the theory of speech acts. *Cambridge: Cambridge Universiy Press*, 1979.

[SJNP97] Carles Sierra, Nick R. Jennings, Pablo Noriega, and Simon Parsons. A framework for argumentation-based negotiation. In *Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages*, pages 167–182, July 1997.

[Woo02] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Chichester, England, 2002.

[WR02] Christopher Walton and Dave Robertson. Flexible multi-agent protocols. Informatics Research Report EDI-INF-RR-0164, Centre for Intelligent Systems and their Applications, October 2002.

Appendix A

Dialogue Protocol Examples

A.1 The E-Commerce Protocol

A.1.1 The Buyer's Dialogue Clauses

```
a(buyer(I,S),B) ::=
    inquiry(I) ⇒ a(seller,S) then
    a(buyerSt2(I),B).

a(buyerSt2(I),B) ::=
    (
        inquiring_preference(T) ⇐ a(sellerSt2(B,I),S) then
        (
            preference(T,P) ⇒ a(sellerSt2(B,I),S) ← preference(I,T,P)
            or
            preference(T,null) ⇒ a(sellerSt2(B,I),S)
        ) then
        a(buyerSt2(I),B)
    )
    or
    a(buyerSt3,B).
```

```
a(buyerSt3,B) ::=
    suitable_items([I1|_]) ⇐ a(sellerSt3(B,I),S) then
    reserve(I1,B) ⇒ a(sellerSt3(B,I),S) then
    waiting_for_payment(I1) ⇐ a(sellerSt3(B,I),S).
```

A.1.2 The Seller's Dialogue Clauses

```
a(seller,S) ::=
    seller_of(I) ← inquiry(I) ⇐ a(buyer(I,S), B) then
    null ← item_topics(I,AllTopics) and
        assert(topicsToInquire(AllTopics)) and
        assert(preferences([])) then
    null ← assertz((compute_results([], I, _, AllItems) :-
            acceptable(I, AllItems))) and
        assertz((compute_results([[T,P]|[]], I, null, NewResults) :-
            acceptable_title(I, List), acceptable(I, AllItems),
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, AllItems), NewResults))) and
        assertz((compute_results([[T,P]|Tail], I, null, NewResults) :-
            acceptable_title(I, List), acceptable(I, AllItems),
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, AllItems), OldResults2),
            compute_results(Tail, OldResults2, NewResults))) and
        assertz((compute_results([[T,P]|[]], I, OldResults, NewResults) :-
            acceptable_title(I, List), \+ OldResults = null,
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, OldResults), NewResults))) and
        assertz((compute_results([[T,P]|Tail], I, OldResults, NewResults) :-
            acceptable_title(I, List), \+ OldResults = null,
            substitute(T, List, P, NewList), set_search(P, NewList, FinalList),
            setof(FinalList, member(FinalList, OldResults), OldResults2),
            compute_results(Tail, OldResults2, NewResults))) and
        assertz((set_search(_, [], []))) and
        assertz((set_search(V, [V|T], [V|R]) :- set_search(V, T, R))) and
        assertz((set_search(V, [X|T], [_|R]) :- \+ X=V, set_search(V, T, R))) then
    a(sellerSt2(B,I),S).
```

```
a(sellerSt2(B,I),S) ::=
    (
        inquiring_preference(T) ⇒ a(buyerSt2(I),B)
            ← retract(topicsToInquire([T|RT])) and assert(topicsToInquire(RT)) then
        preference(T,P) ⇐ a(buyerSt2(I),B) then
        (
            null ← (\+ P=null) and retract(preferences(OldPref)) and
                assert(preferences([[T,P] | OldPref]))
            or
            null ← retract(preferences(Old_list)) and
                select([T,_],Old_list,New_list) and assert(preferences(New_list))
        ) then
        a(sellerSt2(B,I),S)
    )
    or
    (
        null ← topicsToInquire([]) then
        a(sellerSt3(B,I),S)
    ).

a(sellerSt3(B,I),S) ::=
    suitable_items(Results) ⇒ a(buyerSt3,B) ← preferences(Preferences) and
        compute_results(Preferences, I, null, Results) then
    reserve(I1,B) ⇐ a(buyerSt3,B) then
    waiting_for_payment(I1) ⇒ a(buyerSt3,B).
```

A.2 The E-Commerce Protocol Implementing Negotiation

A.2.1 The Buyer's Dialogue Clauses

```
a(buyer(I,S),B) ::=
    inquiry(I) ⇒ a(seller,S) then
    a(buyerSt2(I),B).

a(buyerSt2(I),B) ::=
    (
        inquiring_preference(T) ⇐ a(sellerSt2(B,I),S) then
        (
            preference(T,P) ⇒ a(sellerSt2(B,I),S) ← preference(I,T,P)
            or
            preference(T,null) ⇒ a(sellerSt2(B,I),S) ← no_preference(I,T)
        ) then
        a(buyerSt2(I),B)
    )
    or
    (
        a(buyerSt3,B)
    )
    or
    (
        a(negotiant,B) then
        a(buyerSt2(I),B)
    ).

a(buyerSt3,B) ::=
    suitable_items([I1|_]) ⇐ a(sellerSt3(B),S) then
    reserve(I1,B) ⇒ a(sellerSt3(B),S) then
    waiting_for_payment(I1) ⇐ a(sellerSt3(B),S).
```

A.2.2 The Seller's Dialogue Clauses

```
a(seller,S) ::=
    seller_of(I) ← inquiry(I) ⇐ a(buyer(I,S), B) then
    null ← item_topics(I,AllTopics) and
        assert(all_topics(AllTopics)) and
        assert(topicsToInquire(AllTopics)) and
        assert(preferences([])) then
    a(sellerSt2(B,I),S).

a(sellerSt2(B,I),S) ::=
    (
        inquiring_preference(T) ⇒ a(buyerSt2(I),B)
            ← retract(topicsToInquire([T|RT])) and assert(topicsToInquire(RT)) then
        preference(T,P) ⇐ a(buyerSt2(I),B) then
        (
            null ← P=null
            or
            null ← (\+ P=null) and retract(preferences(OldPref)) and
                assert(preferences([[T,P] | OldPref]))
        ) then
        a(sellerSt2(B,I),S)
    )
    or
    (
        null ← topicsToInquire([]) then
        a(sellerSt3(B),S)
    )
    or
    (
        a(negotiation_initiator(I,T,B), S) ← (\+ var(T)) and preferences(Pref)
            and member([T,_], Pref) then
        a(sellerSt3(B),S)
    ).
```

```
a(sellerSt3(B),S) ::=
    (
        suitable_items(Results0) ⇒ a(buyerSt3,B) ← all_topics(AllTopics) and
            preferences(Prefs) and get_results(I, AllTopics, Prefs, Results0)
        or
        suitable_items(NResults) ⇒ a(buyerSt3,B) ← negotiated_list(NResults)
            and (\+ NResults = [])
    ) then
    reserve(I1,B) ⇐ a(buyerSt3,B) then
    waiting_for_payment(I1) ⇒ a(buyerSt3,B).
```

A.3 The Negotiation Protocol

A.3.1 The Negotiation Initiator's Dialogue Clauses

```
a(negotiation_initiator(I, T,B), S) ::=
    (
        null ← unresolved(UnresolvedTs) and retract(unresolved(_))
            and retract(resolved(ResolvedTs))
        or
        null ← not(unresolved(_)) and all_topics(UnresolvedTs)
            and ResolvedTs=[] and assert(negotiated_list([]))
    ) then
    soften_constraints(Error) ⇒ a(negotiant,B)
    ← preferences(Prefs) and error_test(I, Prefs, UnresolvedTs, Error) then
    (
        negotiation_status(rejected) ⇐ a(negotiant,B)
        or
        (
            negotiation_status(accepted) ⇐ a(negotiant,B) then
            (
                null ← topics_rankings(_)
                or
                (
                    inquiring_topics_ranking(AllTopics) ⇒ a(negotiant,B)
                        ← not(topics_rankings(_)) and all_topics(AllTopics) then
                    assert(topics_rankings(TsRanks)) ←
                        topics_ranking(TsRanks) ⇐ a(negotiant,B)
                )
            ) then
            inquiring_topic_preference(T, data_cat, preference)
                ⇒ a(negotiant,B) then
            assert(topic_preference(T, DataCat, TPrefs)) and
                assert(resolved([T | ResolvedTs]))
                ← topic_preference(T, DataCat, TPrefs) ⇐ a(negotiant,B) then
```

```
            null ← all_topics(AllTopics) and resolved(NResolvedTs)
                and subtract(AllTopics, NResolvedTs, NUnresolvedTs)
                and assert(unresolved(NUnresolvedTs)) then
            a(select_topic(I,null,ResultsI,B),S) ← preferences(Prefs)
                and get_results(I, NUnresolvedTs, Prefs, ResultsI)
        )
    ).

a(select_topic(I, T, OldNResults, B),S) ::=
    (
        null ← resolved(ResolvedTs) and topics_rankings(TsRanks) and
            next_topic(ResolvedTs, TsRanks, T, NextT) then
        (
            (
                null ← topic_preference(NextT, ordinal, _) then
                (
                    null ← (T = null) and assert(firstTopic(NextT))
                    or
                    null ← (\+ T = null)
                ) then
                a(select_value(I,NextT, null, OldNResults, B),S)
            )
            or
            (
                null ← topic_preference(NextT, interval, [LB, UB]) then
                a(interval_data(I,NextT, LB, UB, OldNResults, B),S)
            )
        )
    )
    or
    (
        null ← resolved(ResolvedTs) and topics_rankings(TsRanks) and
            (\+ next_topic(ResolvedTs, TsRanks, T, Next_T)) and
```

```
            retract(negotiated_list(_)) and assert(negotiated_list(OldNResults)) then
        (
            null ← firstTopic(_) and retract(firstTopic(_))
            or
            null ← not(firstTopic(_))
        ) then
        (
            null ← savedResults(_) and retract(savedResults(_))
            or
            null ← not(savedResults(_))
        )
    ).

a(select_value(I, T, V, OldNResults, B),S) ::=
    (
        null ← topic_preference(T, ordinal, TPrefs)
            and next_value(TPrefs, V, NextV) then
        a(ordinal_data(I,T, NextV, OldNResults, B),S)
    )
    or
    (
        null ← topic_preference(T, ordinal, TPrefs) and
            (\+ next_value(TPrefs, V, Next_V)) then
        (
            null ← savedResults(SR)
            or
            null ← not(savedResults(_)) and assert(savedResults(OldNResults))
        ) then
        null ← firstTopic(T) and retract(firstTopic(T)) then
        a(suggest_values(OldNResults, B),S)
    ).
```

```
a(ordinal_data(I, T, V, OldNResults, B),S) ::=
    (
        null ← retract(preferences(OldPref)) and select([T,_], OldPref, NewPref)
            and assert(preferences([[T,V] | NewPref])) and
            ordinal_results(I, T, [[T,V] | NewPref], OldNResults, NewNResults) then
        a(select_topic(I,T, NewNResults, B),S)
    )
    or
    (
        null ←
            (\+ ordinal_results(I,T,[[T,V]|NewPref],OldNResults,NewNResults)) then
        a(select_value(I,T, V, OldNResults, B),S)
    ).

a(interval_data(I, T, LB, UB, OldNResults, B),S) ::=
    (
        null ← interval_results(I, T, [LB,UB], OldNResults, NewNResults)
            and (\+ NewNResults=[])then
        a(select_topic(I,T, NewNResults, B),S)
    )
    or
    (
        null ← interval_results(I, T, [LB,UB], OldNResults, New_NResults)
            and (NewNResults=[]) then
        a(suggest_values(OldNResults, B),S)
   ).

a(suggest_values(OldNResults, B),S) ::=
    only_acceptable_values(OldNResults) ⇒ a(reassessment_negotiant, B) then
    (null ← retract(savedResults(_))or null ← not(savedResults(_)))then
    (
        (
            accept(SelectedResult) ⇐ a(reassessment_negotiant, B) then
```

```
        null ← retract(negotiated_list(_)) and
            assert(negotiated_list(SelectedResult))
    )
    or
    negotiation_status(failed) ⇐ a(reassessment_negotiant, B)
).
```

A.3.2 The Negotiant's Dialogue Clauses

```
a(negotiant,B) ::=
    soften_constraints(Error) ⇐ a(negotiation_initiator(I,T,B), S) then
    (
        null ← (\+ Error = unknown) then
        negotiation_status(rejected) ⇒ a(negotiation_initiator(I,T,B), S)
    )
    or
    (
        null ← Error = unknown then
        negotiation_status(accepted) ⇒ a(negotiation_initiator(I,T,B), S) then
        (
            null
            or
            (
                inquiring_topics_ranking(AllTopics) ⇐
                    a(negotiation_initiator(I,T,B), S) then
                topics_ranking(TsRanks) ⇒ a(negotiation_initiator(I,T,B), S)
                    ← topics_ranking(I, TsRanks)
            )
        ) then
        inquiring_topic_preference(T, data_cat, preference) ⇐
            a(negotiation_initiator(I,T,B), S) then
        topic_preference(T, DataCat, TPrefs) ⇒ a(negotiation_initiator(I,T,B), S)
            ← topic_preferences(I, TsPrefs) and member([T, DataCat, TPrefs], TsPrefs)
        then
        (
            null
            or
            a(reassessment_negotiant,B)
        )
    ).
```

```
a(reassessment_negotiant,B) ::=
    only_acceptable_values(Results) ⇐ a(suggest_values(X,B), S) then
    (
        accept(SelectedResult) ⇒ a(suggest_values(X,B), S)
            ← accepted(Results, SelectedResult)
        or
        negotiation_status(failed) ⇒ a(suggest_values(X,B), S)
            ← (\+ accepted(Results, SelectedResult))
    ).
```

Appendix B

The Code

This appendix contains *only* the major prolog files for this dissertation:

– The *negot.pl* file of section B.1 contains the predicates used by the negotiation protocol of Appendix A.3.
– The basic.pl is our major work. It contains all the rewrite rules and their modifications that resulted in the 'induced backtracking' methodology proposed in Chapter 3.
– The loader.pl file contains the predicates that load the institutions.
– The interface.pl deals with the user interface. We have included it in this appendix since it has been slightly modified to keep the user informed of dialogue states.

B.1 negot.pl

```
:- use_module(library(lists)).

subtract([], _, []).
subtract(List1, [], List1).
subtract([H|T], List2, Result) :- member(H, List2), subtract(T, List2, Result).
subtract([H|T], List2, [H|Result]) :- \+ member(H, List2),
    subtract(T, List2, Result).
accepted([H|_], H).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicates to find error type
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
error_test(I, ListofList, UnresolvedTopics, AllErrors) :-
    get_preferences(UnresolvedTopics, ListofList, ListofNewPreferences),
    unacceptable_title(I, [_,_,List]),
    prepare_search(List, ListofNewPreferences , SearchList),
    unacceptable(I, ListToSearch),
    get_error(SearchList, ListToSearch, AllErrors).

get_preferences([H|[]], ListofList, [[H,P]]) :-
    member([H,P], ListofList).
get_preferences([H|[]], ListofList, []) :-
    \+ member([H,_], ListofList).
get_preferences([H|T], ListofList, [[H,P]|Results]) :-
    member([H,P], ListofList),
    get_preferences(T, ListofList, Results).
get_preferences([H|T], ListofList, Results) :-
    \+ member([H,_], ListofList),
    get_preferences(T, ListofList, Results).

prepare_search([H|[]], List, [P]) :-
```

```
    member([H,P], List).
prepare_search([H|[]], List, [null]) :-
    non_member([H,_], List).
prepare_search([H|T], List, [P|Result]) :-
    member([H,P], List),
    prepare_search(T, List, Result).
prepare_search([H|T], List, [null|Result]) :-
    non_member([H,_], List),
    prepare_search(T, List, Result).

get_error(SearchList, ListToSearch, AllErrors) :-
    setof([A,B,SearchList], member([A,B,SearchList], ListToSearch), AllErrors).
get_error(_, _, unknown).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicates to find applicable results for a given list of topics
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
get_results(I, ListOfTopics, Preferences, Results) :-
    acceptable(I, CurrentList),
    search_results(ListOfTopics, I, Preferences, CurrentList, Results).

search_results([], _, _, CurrentList, CurrentList).
search_results([H|T], I, ListOfPreferences, CurrentList, FinalResultingList) :-
    memberchk([H,P], ListOfPreferences), acceptable_title(I, List),
    substitute(H, List, P, NewList), set_search(P, NewList, FinalList),
    setof(FinalList, member(FinalList, CurrentList), ResultingList),
    search_results(T, I, ListOfPreferences, ResultingList, FinalResultingList) .
search_results([H|T], I, ListOfPreferences, CurrentList, FinalResultingList) :-
    \+ memberchk([H,_], ListOfPreferences),
    search_results(T, I, ListOfPreferences, CurrentList, FinalResultingList) .

set_search(_, [], []).
set_search(V, [V|T], [V|R]) :- set_search(V, T, R).
```

```
set_search(V, [X|T], [_|R]) :- \+ X=V, set_search(V, T, R).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicates to find applicable results for an ordinal data category
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ordinal_results(I, Topic, Preferences, [], Results) :-
    acceptable(I, CurrentList),
    search_results([Topic], I, Preferences, CurrentList, Results).
ordinal_results(I, Topic, Preferences, CurrentList, Results) :-
    search_results([Topic], I, Preferences, CurrentList, Results).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Predicates to find applicable results for an interval data category
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
interval_results(I, Topic, [X,Y], [], Results) :-
    acceptable(I, CurrentList),
    fix_boundaries([X,Y], [LB,UB]), acceptable_title(I, List), nth(N, List, Topic),
    compare_interval(CurrentList, N, LB, UB, Results).
interval_results(I, Topic, [X,Y], CurrentList, Results) :-
    fix_boundaries([X,Y], [LB,UB]), acceptable_title(I, List), nth(N, List, Topic),
    compare_interval(CurrentList, N, LB, UB, Results).

fix_boundaries([X,Y], [X,Y]) :- X =< Y.
fix_boundaries([X,Y], [Y,X]) :- X > Y.

compare_interval([H|[]], N, LB, UB, [H]) :-
    nth(N, H, Value),
    Value >= LB, Value =< UB.
compare_interval([_|[]], _, _, _, []).
compare_interval([H|T], N, LB, UB, [H|FL]) :-
    nth(N, H, Value),
    Value >= LB, Value =< UB,
    compare_interval(T, N, LB, UB, FL).
```

```
compare_interval([_|T], N, LB, UB, FL) :-
    compare_interval(T, N, LB, UB, FL).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicates to find Next Topic and Next Value, respectively
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
next_topic(TopicstoResolve, AllTopicsRanking, null, FirstTopic) :-
    get_ranking(TopicstoResolve, AllTopicsRanking, Results),
    get_first_rank(Results, FirstTopic).
next_topic(TopicstoResolve, AllTopicsRanking, CurrentTopic, NextTopic) :-
    get_ranking(TopicstoResolve, AllTopicsRanking, Results),
    member([Rank,CurrentTopic], Results),
    get_higher_ranks(Rank, Results, ListofHigherRanks),
    get_first_rank(ListofHigherRanks, NextTopic).

get_ranking([H|[]], TopicsRanking, [[R,H]]) :-
    member([R,H], TopicsRanking).
get_ranking([H|T], TopicsRanking, [[R,H] | Results]) :-
    member([R,H], TopicsRanking),
    get_ranking(T, TopicsRanking, Results).

next_value(ListofValues, null, FirstValue) :-
    get_first_rank(ListofValues, FirstValue).
next_value(ListofValues, CurrentValue, NextValue) :-
    member([Rank,CurrentValue], ListofValues),
    get_higher_ranks(Rank, ListofValues, ListofHigherRanks),
    get_first_rank(ListofHigherRanks, NextValue).

get_first_rank([[_,Value]|[]], Value).
get_first_rank([[Rank1,Value1]|[[Rank2,_]|[]]], Value1) :- Rank1 < Rank2.
get_first_rank([[Rank1,_]|[[Rank2,Value2]|[]]], Value2) :- Rank1 > Rank2.
get_first_rank([[Rank1,Value1]|[[Rank2,_]|T]], FirstValue) :- Rank1 < Rank2,
    get_first_rank([[Rank1,Value1]|T], FirstValue).
```

```
get_first_rank([[Rank1,_]|[[Rank2,Value2]|T]], FirstValue) :- Rank1 > Rank2,
    get_first_rank([[Rank2,Value2]|T], FirstValue).

get_higher_ranks(Rank1, [[Rank2,Value2]|[]], [[Rank2, Value2]]) :- Rank1 < Rank2.
get_higher_ranks(Rank1, [[Rank2,_]|[]], []) :- Rank1 >= Rank2.
get_higher_ranks(Rank1, [[Rank2,Value2]|T], [[Rank2,Value2]|Results]) :-
    Rank1 < Rank2, get_higher_ranks(Rank1, T, Results).
get_higher_ranks(Rank1, [[Rank2,_]|T], Results) :-
    Rank1 >= Rank2, get_higher_ranks(Rank1, T, Results).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.2 basic.pl

```
:- use_module(library(lists)),
    use_module(library(random)).
:- ensure_loaded([util,negot]).
:- op(900, xfx, '::='),
    op(900, xfx, '::'),
    op(900, xfx, '>>'),
    op(800, xfx, '=>'),
    op(800, xfx, '<='),
    op(830, xfx, '<--'),
    op(820, xfy, and),
    op(850, xfy, par),
    op(850, xfy, then),
    op(850, xfy, or).

%%%%%%%%%%%%%%%%%%%% starting the expansion process %%%%%%%%%%%%%%%%%%
% modified to deal with incoming failed message notices
expansion(Agent, Ms, Os, P, FinalMs, FinalOs, FinalP) :-
    check_failed_message(Agent, Ms, P, Msi, Pi),
    expansion_step(Agent, Msi, Os, Pi, NewMs, NewOs, NewP),
    check_EOF_expansion(Agent, Ms, Os, P,
        NewMs, NewOs, NewP, FinalMs, FinalOs, FinalP).
expansion(Agent, Ms, Os, P, Ms, Os, P) :-
    \+ expansion_step(Agent, Ms, Os, P, _, _, _).

% added to check for failed message notices
% and fix the protocol, if necessary
check_failed_message(At, [m(At,f(OldM => Af) <= Af)], Prot, [], NewProt) :-
    !, mark_failed(At, c(OldM => Af), Prot, NewProt).
check_failed_message(_, Ms, Prot, Ms, Prot).
check_EOF_expansion(_, Ms, Os, P, NewMs, NewOs, NewP, Ms, Os, P) :-
    NewMs = Ms,
```

```
    NewOs = Os,
    NewP = P.
check_EOF_expansion(Agent, _, _, _, NewMs, NewOs, NewP,
        FinalMs, FinalOs, FinalP) :-
    expansion(Agent, NewMs, NewOs, NewP, FinalMs, FinalOs, FinalP).

%%%%%%%%%%%%%%%%%% selecting dialogue clause to exapnd %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% and saving it after expansion %%%%%%%%%%%%%%%%%%%%
expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    protocol_select(agent, P, (a(ARole,Id) ::= Def), P1),
    expand_protocol((a(ARole,Id) ::= Def), Role, Id, Ms, Os, P1,
        NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).
expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    \+ protocol_select(agent, P, (a(_,Id) ::= _), _),
    protocol_member(dialogue, P, Clause),
    Clause = (a(Role,Id) ::= Def),
    expand_protocol((a(Role,Id) ::= Def), Role, Id, Ms, Os, P,
        NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).

%%%%%%%%%%%%%%%%%%%%%%%%%% the rewrite rules %%%%%%%%%%%%%%%%%%%%%%%%%%
% first 2 cases added to deal with messages received after closing the protocol
expand_protocol(Role ::= Def, _, _, [], Os, P, Role ::= Def, [], Os, P) :-
    closed(Def), !.
expand_protocol(Role ::= Def, _, Id, Ms, Os, P, Role ::= E, Mf, Of, Pf) :-
    closed(Def),
    open_and_fail(Def, NDef),
    expand_protocol(NDef, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(Role ::= Def, _, Id, Ms, Os, P, Role ::= E, Mf, Of, Pf) :-
    expand_protocol(Def, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
% modified to return A or B instead of only expanded part)
% also added 2 cases for failure (cases 1 and 2)
```

```
expand_protocol(A or B, _, _, Ms, Os, P, f(A or B), Ms, Os, P) :-
    failed(A or B).
expand_protocol(f(A) or B, Role, _, [], Os, P, f(A) or B, [],
        [m(Role,f(M => Role) => S) | Os], P) :-
    receiving_message(A, M, S),
    receiving_message(B).
expand_protocol(A or B, Role, Id, Ms, Os, P, E or B, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(A or B, Role, Id, Ms, Os, P, A or E, Mf, Of, Pf) :-
    expand_protocol(B, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
% A then B modified
% 2 cases (2nd and 4th) added for failure cases
% NOTE: if only B failed then we first need to close A as failed
expand_protocol(A then B, Role, Id, Ms, Os, P, A then EB, Mf, Of, Pf) :-
    closed(A),
    expand_protocol(B, Role, Id, Ms, Os, P, EB, Mf, Of, Pf).
expand_protocol(A then B, _, _, Ms, Os, P, FA then B, Ms, Os, P) :-
    closed(A),
    failed(B),
    fail_clause(A, FA).
expand_protocol(A then B, Role, Id, Ms, Os, P, EA then B, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, EA, Mf, Of, Pf).
expand_protocol(A then B, _, _, Ms, Os, P, f(A then B), Ms, Os, P) :-
    failed(A).
% A par B modified
% also added 2 cases for failure issues (cases 3 and 4)
expand_protocol(A par B, Role, Id, Ms, Os, P, EA par EB, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, EA, Mn, On, Pn),
    expand_protocol(B, Role, Id, Mn, On, Pn, EB, Mf, Of, Pf).
expand_protocol(A par B, Role, Id, Ms, Os, P, EA par EB, Mf, Of, Pf) :-
    expand_protocol(B, Role, Id, Ms, Os, P, EB, Mn, On, Pn),
    expand_protocol(A, Role, Id, Mn, On, Pn, EA, Mf, Of, Pf).
expand_protocol(A par B, _, _, Ms, Os, P, f(A par B), Ms, Os, P) :-
```

```
    failed(A).
expand_protocol(A par B, _, _, Ms, Os, P, f(A par B), Ms, Os, P) :-
    failed(B).
expand_protocol(C <-- M <= A, Role, Id, Ms, Os, P, c(M <= A), Mf, Os, Pf) :-
    memberchk(m(Role,M <= A), Ms),
    satisfied(Id, P, C, Pf),
    select(m(Role,M <= A), Ms, Mf).
% added to mark term as failed when constraint not satisfied
% this is the only case where it is not closed as failed
% and a message is sent to inform failure
expand_protocol(C <-- M <= A, Role, Id, Ms, Os, P, C <-- M <= A, Mf,
        [m(Role,f(M => Role) => A) | Os], P) :-
    select(m(Role,M <= A), Ms, Mf),
    \+ satisfied(Id, P, C, _).
expand_protocol(_ <-- M <= A, _, _, Ms, Os, P, f(M <= A), Ms, Os, P) :-
    memberchk(m(_, _ <= _), Ms).
expand_protocol(M => A <-- C, Role, Id, Ms, Os, P, c(M => A), Ms,
        [m(Role,M => A) | Os], Pf) :-
    satisfied(Id, P, C, Pf).
% added to mark term as failed when constraint not satisfied
expand_protocol(M => A <-- C, _, Id, Ms, Os, P, f(M => A), Ms, Os, P) :-
    \+ satisfied(Id, P, C, _).
expand_protocol(M <= A, Role, _, Ms, Os, P, c(M <= A), Mf, Os, P) :-
    select(m(Role,M <= A), Ms, Mf).
expand_protocol(M <= A, _, _, Ms, Os, P, f(M <= A), Ms, Os, P) :-
    memberchk(m(_, _ <= _), Ms).
expand_protocol(M => A, Role, _, Ms, Os, P, c(M => A), Ms,
        [m(Role,M => A) | Os], P).
expand_protocol(Role <-- C, _, Id, Ms, Os, P, Role ::= Def, Ms, Os, Pf) :-
    Role = a(_,_),
    satisfied(Id, P, C, Pf),
    protocol_member(dialogue, P, (Role ::= Def)).
% added to mark term as failed when constraint not satisfied
```

```
expand_protocol(Role <-- C, _, Id, Ms, Os, P, f(Role), Ms, Os, P) :-
    Role = a(_,_),
    \+ satisfied(Id, P, C, _).
expand_protocol(Role, _, _, Ms, Os, P, Role ::= Def, Ms, Os, P) :-
    Role = a(_,_),
    protocol_member(dialogue, P, (Role ::= Def)).
expand_protocol(null <-- C, _, Id, Ms, Os, P, c(null), Ms, Os, Pf) :-
    satisfied(Id, P, C, Pf).
% added to mark term as failed when constraint not satisfied
expand_protocol(null <-- C, _, Id, Ms, Os, P, f(null), Ms, Os, P) :-
    \+ satisfied(Id, P, C, _).
expand_protocol(null, _, _, Ms, Os, P, c(null), Ms, Os, P).

%%%%%%%%%%%%%%% backtracking marking messages as failed %%%%%%%%%%%%%%%
%%%%%%%%%%%%% testing for actions of receiving a message %%%%%%%%%%%%%%
% added to fail 'A' in 'A then B'
% when 'A' is closed and 'B' fails
fail_clause(A then B, f(FA then FB)) :-
    fail_clause(B, FB),
    fail_clause(A, FA).
fail_clause(A par B, f(FA par FB)) :-
    fail_clause(A, FA),
    fail_clause(B, FB).
fail_clause(A or B, FA or B) :-
    closed(A),
    fail_clause(A, FA).
fail_clause(A or B, A or FB) :-
    closed(B),
    fail_clause(B, FB).
fail_clause(_ <-- M <= A, f(M <= A)).
fail_clause(M => A <-- _, f(M => A)).
fail_clause(M <= A, f(M <= A)).
fail_clause(M => A, f(M => A)).
```

```
fail_clause(c(X), f(c(X))).

% added to check if the first action in a definition
% is that of receiving a message (needed when backtracking to know
% if we need to inform sender a failure notice)
receiving_message(_ <= _).
receiving_message(_ <-- _ <= _).
receiving_message(A then _) :-
    receiving_message(A).
% searching for the message to be sent back to the sender
% informing it of its failure
receiving_message(c(M <= A), M, A).
receiving_message(c(_ <-- M <= A), M, A).
receiving_message(f(B) then _, M, A) :-
    receiving_message(B, M, A).

%%%%%%%%%%%%%%%% testing for closed or failed clauses %%%%%%%%%%%%%%%%%
closed(c(_)).
closed(A or _) :-
    closed(A).
closed(_ or B) :-
    closed(B).
closed(A then B) :-
    closed(A),
    closed(B).
closed(A par B) :-
    closed(A),
    closed(B).
closed(_ ::= Def) :-
    closed(Def).

% in the definitions of the following predicate,
% 'f()' is used instead of 'failed'
```

```
% since we want to close each part by itslef
% to be be able to check for sending/receiving messages
failed(f(_)).
failed(f(_) or f(_)).
failed(f(_) then _).
failed(_ then f(_)).
failed(f(_) par _).
failed(_ par f(_)).
failed(_ ::= f(_)).

%%%%%%%%%%%%% marking closed nodes as failed when needed %%%%%%%%%%%%%%
% added to mark closed nodes as failed
% when a failure message notice is received
mark_failed(a(_,Id), CM, P, NewP) :-
    protocol_select(agent, P, (a(Any_Role,Id) ::= Def), P1),
    mark_failed_message(a(Any_Role,Id) ::= Def, CM, Pf),
    protocol_add(agent, P1, Pf, NewP).
mark_failed(a(_,Id), _, P, P) :-
    \+ protocol_select(agent, P, (a(_,Id) ::= _), _).

mark_failed_message(A ::= c(Def), CM, A ::= Def2) :-
    mark_failed_message(Def, CM, Def2).
mark_failed_message(A ::= Def, CM, A ::= Def2) :-
    mark_failed_message(Def, CM, Def2).
mark_failed_message(A or B, CM, A2 or B2) :-
    mark_failed_message(A, CM, A2),
    mark_failed_message(B, CM, B2).
mark_failed_message(A then B, CM, A2 then B2) :-
    mark_failed_message(A, CM, A2),
    mark_failed_message(B, CM, B2).
mark_failed_message(A par B, CM, A2 par B2) :-
    mark_failed_message(A, CM, A2),
    mark_failed_message(B, CM, B2).
```

```
mark_failed_message(c(M => A), c(M => A), f(c(M => A))).
mark_failed_message(X, _, X).

% added to mark closed leaf nodes as failed
% when a completed dialogue receives a normal message
% NOTE: marking closed leaf nodes as failed
% automatically opens the protocol
open_and_fail(A then B, A then NB) :-
    open_and_fail(B, NB).
open_and_fail(A or B, NA or NB) :-
    open_and_fail(A, NA),
    open_and_fail(B, NB).
open_and_fail(c(A), f(c(A))).
open_and_fail(X, X).

%%%%%%%%%%%% testing for satisfied constraints predicates %%%%%%%%%%%%%
satisfied(Id, P, A and B, Pf) :- !,
    satisfied(Id, P, A, Pn),
    satisfied(Id, Pn, B, Pf).
satisfied(Id, P, X, Pf) :-
    meta_pred(Id, X, P, Pf, Call), !,
    Call.
satisfied(Id, P, X, P) :-
    \+ meta_pred(Id, X, P, _, _),
    call_direct(X),
    X.
satisfied(Id, P, X, P) :-
    protocol_member(common_knowledge, P, known(Id, X)).
satisfied(Id, P, X, Pf) :-
    protocol_member(common_knowledge, P, known(Id, X <-- C)),
    satisfied(Id, P, C, Pf).

call_direct(X) :-
```

```
    (predicate_property(X, built_in) ;
    predicate_property(X, interpreted) ;
    predicate_property(X, imported_from(_))), !.

meta_pred(Id, not(X), P, P, \+ satisfied(Id,P,X,_)).
meta_pred(Id, retract(X), P, Pf, protocol_remove(common_knowledge,P,known(Id,X),Pf)).
meta_pred(Id, assert(X), P, Pf, protocol_add(common_knowledge,P,known(Id,X),Pf)).

%%%%%%%%%%%%%%%%%% managing the protocol predicates %%%%%%%%%%%%%%%%%%
protocol_component(agents, def(Clauses, _, _), Clauses).
protocol_component(dialogue, def(_, Clauses, _), Clauses).
protocol_component(common_knowledge, def(_, _, Clauses), Clauses).

protocol_member(agent, def(Clauses,_,_), Clause) :-
    member(Clause, Clauses).
protocol_member(dialogue, def(_,Clauses,_), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).
protocol_member(common_knowledge, def(_,_,Clauses), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).

protocol_select(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
protocol_select(dialogue, def(A,Clauses,B), ClauseCopy, def(A,R,B)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).
protocol_select(common_knowledge, def(A,B,Clauses), ClauseCopy, def(A,B,R)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).

protocol_remove(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
```

```
protocol_remove(dialogue, def(A,Clauses,B), Clause, def(A,R,B)) :-
    select(Clause, Clauses, R).
protocol_remove(common_knowledge, def(A,B,Clauses), Clause, def(A,B,R)) :-
    select(Clause, Clauses, R).

protocol_add(agent, def(Clauses,A,B), X, def([X|Clauses],A,B)).
protocol_add(dialogue, def(A,Clauses,B), X, def(A,[X|Clauses],B)).
protocol_add(common_knowledge, def(A,B,Clauses), X, def(A,B,[X|Clauses])).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.3 loader.pl

```
:- ensure_loaded(basic).

load_institution(Institution, InstDef) :-
    concat(Institution,'.inst',File),
    see(File),
    read_institution(InstDef),
    seen.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

read_institution(InstDef) :-
    read_institution1(def([],[],[]), InstDef).
read_institution1(InstDef, FinalInstDef) :-
    read(Clause),
    \+ Clause = end_of_file, !,
    add_to_institution_def(Clause, InstDef, NewInstDef),
    read_institution1(NewInstDef, FinalInstDef).
read_institution1(InstDef, InstDef).

add_to_institution_def((Head ::= Body),def(I,D,K),def(I,D1,K)) :-
    fix_protocol((Head ::= Body), Fixed_Protocol),
    append(D, [Fixed_Protocol], D1).
add_to_institution_def(known(Agent,Clause),def(I,D,K), def(I,D,K1)) :-
    append(K, [known(Agent,Clause)], K1).

% 'fix_protocol' to change definitions of the form ((A or B) then C)
% into ( (A then C) or (B then C) )
fix_protocol(P, Pf) :-
    fix(P, Pi),
    \+ Pi = P, !,
    fix_protocol(Pi, Pf).
```

```
fix_protocol(P, Pf) :-
    fix(P, Pf).


fix(A ::= Def, A ::= P) :-
    fix(Def, P).
fix(A or B, P1 or P2) :-
    fix(A, P1),
    fix(B, P2).
fix((A1 or A2) then B, ((P1 then P3) or (P2 then P3))) :-
    fix(A1, P1),
    fix(A2, P2),
    fix(B, P3).
fix(A then B, P1 then P2) :-
    fix(A, P1),
    fix(B, P2).
fix(A par B, P1 par P2) :-
    fix(A, P1),
    fix(B, P2).
fix(Unit, Unit).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

B.4 interface.pl

```
:- ensure_loaded([util,
       basic,
       display_util,
       '/home/s0233771/agent/PTK/ptk.pl',
       loader]).
:- dynamic pwin/5, rwin/5, swin/5.
agent_dir('/home/s0233771/agent/agents').
institution(eCommerce, buyer, [item,seller]).
institution(eCommNegt, buyer, [item,seller]).
institution(testingEx, first, [second,middle]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
manager :-
    setof(w(I,R,P), institution(I,R,P), Institutions),
    institution_items(Institutions, Frames),
    tk_new_window(
        [frame(idframe,
            [(borderwidth,5),(relief,raised),(bg,yellow)],
            pack([(side,top),(anchor,w),(fill,x)]),
            [button(identity,
                [(background,'SlateGrey'),(text,'Identity :')],
                    pack([(side,left)])),
            entry(id,
                    [(width,20)],
                    pack([(side,left)])),
            button(react,
                    [(background,'SlateGrey'),(text,'New Messages')],
                    pack([(side,right),(fill,x)]))
            ]),
        frame(arole,
            [(borderwidth,5),(bg,yellow)],
```

```
          pack([(side,top),(anchor,w),(fill,x)]),
          [message(arole,
              [(text,'Roles'), (width,200),(bg,yellow)],
                  pack([(side,top)]))]),
      frame(main,
          [(height,200),(width,300)],
          pack([(fill,both),(expand,true)]),
          Frames)
      ],
      manager_callback,
      _).

institution_items([E|T], [I|R]) :-
    institution_item(E, I),
    institution_items(T, R).
institution_items([], []).

institution_item(w(I,R,P), Frame) :-
    IItem = button(inst,
            [(background,'SlateGrey'),(text,I)],
            pack([(side,left),(fill,x)])),
    RItem = message(R,
            [(text,R),(width,200),(bg,yellow)],
            pack([(side,left)])),
    institution_args(P, Args),
    append([IItem,RItem], Args, Items),
    Frame = frame(I,
            [(borderwidth,5),(relief,raised),(bg,yellow)],
            pack([(side,top),(anchor,w),(fill,x)]),
            Items).

institution_args([E|T], [A|R]) :-
    institution_arg(E, A),
```

```
    institution_args(T, R).
institution_args([], []).

institution_arg(Id, Arg) :-
    atom(Id),
    Arg = frame(Id,
            [(borderwidth,2),(relief,raised),(bg,yellow)],
            pack([(side,left),(anchor,w),(fill,x)]),
            [message(m,
                [(text,Id),
                (width,200),(bg,yellow)],
                pack([(side,left)])),
            entry(id, [(width,10)], pack([(side,left)]))]).

manager_callback(Interp, button, _, identity) :-
    tk_operation('$TclInterp'(Interp), entry, '.idframe.id', get(Id)),
    load_agent(Id).
manager_callback(Interp, button, I, inst) :-
    institution(I,R,Items),
    tk_operation('$TclInterp'(Interp), entry, '.idframe.id', get(Id)),
    get_institution_args(Interp, I, Items, Args),
    Role =.. [R|Args],
    load_institution(I, Prot),
    postit(a(Role,Id), [], Prot).
manager_callback(Interp, button, _, react) :-
    tk_operation('$TclInterp'(Interp), entry, '.idframe.id', get(Id)),
    retrieve_message(_, Id, Dialogue),
    Dialogue = protocol(m(Af,M => At),Prot),
    postit(At, [m(At,M <= Af)], Prot).

get_institution_args(Interp, I, [Id|T], [Arg|R]) :-
    concat_list(['.main.',I,'.',Id,'.',id], Obj),
    tk_operation('$TclInterp'(Interp), entry, Obj, get(Arg)),
```

```
    get_institution_args(Interp, I, T, R).
get_institution_args(_, _, [], []).

load_agent(Name) :-
    agent_dir(Path),
    concat_list([Path,'/',Name,'.agent'],File),
    see(File),
    read_agent,
    seen.

read_agent :-
    read(Clause),
    \+ Clause = end_of_file, !,
    assertz(Clause),
    read_agent.
read_agent.

% 'postit' was modified to call 'displaybox'
% instead of doing the job itslef
% since we now allow three cases of the box to be displayed
postit(Role, IMessages, Prot) :-
    expansion(Role, IMessages, [], Prot, RMessages, Messages, EProt),
    displaybox(RMessages, Messages, EProt).

% the first case of 'displaybox' added
% to inform the user of dialogue failure
displaybox(RMessages, Messages, EProt) :-
    RMessages = [],
    Messages = [],
    protocol_component(agents, EProt, AProt),
    \+ all_closed(AProt),
    final_message('Dialogue has failed!').
```

```
% the second case of 'displaybox' added
% to inform the user of dialogue success
displaybox(RMessages, Messages, EProt) :-
    RMessages = [],
    Messages = [],
    protocol_component(agents, EProt, AProt),
    all_closed(AProt),
    final_message('Dialogue has completed successfully!').

% the last case of 'displaybox' is
% the case originally covered by 'postit'
displaybox(RMessages, Messages, EProt) :-
    RMessages = [],
    Messages = [m(Agent,Message)],
    protocol_component(agents, EProt, Insts),
    copy_term((Agent,Message,Insts), (CAgent,CMessage,CInsts)),
    term_to_atom(CAgent, AAgent),
    term_to_atom(CMessage, AMessage),
    goal_sequence_to_display(CInsts, Text),
    BColour = 'SlateGrey',
    tk_new_window(
        [frame(arole,
            [(borderwidth,5),(relief,raised),(bg,yellow)],
            pack([(side,top),(anchor,w),(fill,x)]),
            [message(arole,
                [(text,AAgent), (width,400),(bg,yellow)], pack([(side,top)]))]),
        frame(initiate,
            [(height,200),(width,300)],
            pack([(fill,both),(expand,true)]),
            [frame(dialogueframe,
                [(borderwidth,5),(relief,raised),(bg,yellow)],
                pack([(side,top),(anchor,w),(fill,x)]),
                [message(m1,
```

```
            [(text,'Dialogue :'),
            (width,200),(bg,yellow)],
                pack([(side,left)])),
        scrolling_text(text,
                [(width,70),(height,10), (wrap,none),
                (borderwidth,5),
            (font,'-adobe-courier-bold-r-normal-*-14-*-*-*-*-*-*')],
                pack([(side,left),(fill,both),(expand,true)]),
                xyscroll([],[]),
                maketext(Text, []))
            ]),
    frame(responseframe,
        [(borderwidth,5),(relief,raised),(bg,yellow)],
        pack([(side,top),(anchor,w),(fill,x)]),
        [frame(buttons,
            [(borderwidth,3),(relief,raised),(bg,yellow)],
            pack([(side,left),(anchor,w),(fill,both)]),
                [button(respond,
                [(background,BColour),
                (text,'Send message')],
                pack([(side,top)])),
                button(quit,
                [(background,BColour),
                (text,'Forget it')],
                pack([(side,top)]))
                ]),
            scrolling_text(response,
                [(width,50),
                (height,3),
                (borderwidth,5),
            (font,'-adobe-courier-bold-r-normal-*-14-*-*-*-*-*-*')],
                pack([(side,left),
                (fill,both),
```

```
                        (expand,true)]),
                        xyscroll([],[]),
                        maketext([AMessage],[]))
                    ])
            ])
        ],
        postit_callback,
        Interp),
    assert(pwin(Interp,[m(Agent,Message)],EProt)).

% the 'all_closed' predicate was added
% to help test whether the protocol is closed or not
all_closed([]).
all_closed([_ ::= Def | []]) :- closed(Def).
all_closed([_ ::= Def | Y]) :- closed(Def), all_closed(Y).

% the 'final_message' predicate was added
% to display the final message which could either be
% a success or failure notice
final_message(Message) :-
    tk_new_window(
        [frame(lasttitle,
            [(borderwidth,5),(relief,raised),(bg,yellow)],
            pack([(side,top),(anchor,w),(fill,x)]),
            [message(lastmsg,
                [(text,'End of Dialogue'), (width,400),(bg,yellow)],
                pack([(side,top)]))]),
        frame(lasttext,
            [(borderwidth,5),(relief,raised),(bg,yellow)],
            pack([(side,top),(anchor,w),(fill,x)]),
            [scrolling_text(text,
                [(width,40),(height,5), (wrap,none),
                (borderwidth,5),
```

```
                (font,'-adobe-courier-bold-r-normal-*-14-*-*-*-*-*-*-*')],
                pack([(side,left),(fill,both),(expand,true)]),
                xyscroll([],[]),
                maketext([Message], []))
            ])
        ],
        _,
        _).

postit_callback(Interp, button, _, quit) :-
    delete_proaction_win(Interp).
postit_callback(Interp, button, _, respond) :-
    pwin('$TclInterp'(Interp), Messages, Prot),
    send_protocol_messages(Messages, Prot),
    delete_proaction_win(Interp).

delete_proaction_win(Interp) :-
    retractall(pwin('$TclInterp'(Interp),_,_)),
    tk_main_window('$TclInterp'(Interp), Win),
    tk_destroy_window(Win).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
agent_id_from_role(a(_,Id), Id).

send_protocol_messages([m(Af,M => At)|T], Prot) :-
    agent_id_from_role(Af, From), nonvar(From),
    agent_id_from_role(At, To), nonvar(To),
    send_message(From, To, protocol(m(Af,M => At),Prot)),
    send_protocol_messages(T, Prot).
send_protocol_messages([], _).

send_message(From, To, Message) :-
    find_server(Server, PID),
```

```
    add_message(Server, PID, From, To, Message),
    write('Sent:  '), write(Message), nl, !.

retrieve_message(From, To, Message) :-
    find_server(Server, PID),
    read_message(Server, PID, From, To, Message),
    write('Read:  '), write(Message), nl, !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```