A Structural Synthesis System for LCC Protocols

Argyrios Grivas

Master of Science
School of Informatics
University of Edinburgh
2005

Abstract

LCC (Lightweight Communication Calculus) [10] is a language for specifying models of interaction for multi-agent systems. It is an executable specification language in the sense that mechanisms exist for deploying LCC protocols when coodinating software components. It is also a declarative language in the sense that it may be understood and analysed without committment to a specific deployment system. A weakness of LCC from a software engineering point of view is that it has not been provided with methods or tools for structured design of specifications. Currently, designers simply write protocols. The aim of this project is to construct a structured design tool that embodies an incremental design method.

Although LCC is a process calculus, it also has many features in common with Horn clause specification and many of the analytical and deployment methods associated with it are based on forms of inference familiar to logic programmers. It is therefore natural to consider whether structured design methods for logic programs can be applied to LCC. One of the best understood of these design methods is Techniques Editing [2], in which definitions of predicates (as sets of Horn clauses) are constructed incrementally from argument ”slices”. LCC, however, is not simply a Horn clause language - it also is a process specification language. Process oriented methods of design therefore might also be usefully applied, in particular to provide early ”skeletons”: initial partial definitions used as a precursor to more detailed design.

Acknowledgements

I would like to thank my supervisor Dr. David Robertson for his help and guidance throughout this endeavour. I would also like to express my appreciation to the Greek State Scholarships' Foundation for the financial support they provided me during my studies.

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.

(*Argyrios Grivas*)

Table of Contents

Chapter 1

Introduction

The LCC (Lightweight Communication Calculus) language is an approach for decentralized coordination of Multi-Agent Systems (MAS) in open environments. LCC is an executable specification language in the sense that mechanisms exist for deploying LCC protocols when coordinating software components. Although the LCC approach has a strong theoretical basis and seems to provide a good balance between efficient coordination and respect of the agents' independence, there is some work to be done yet from the software engineering point of view. In particular, one of the potential issues that should be addressed is the structured design of LCC specifications. At the moment, the designers have to write protocols by hand and no tool that supports this process exists.

The aim of this project is to develop a structured design editor for LCC protocols. In order to achieve this goal, a closer look to LCC itself is required. LCC is a process calculus, but has a lot in common with Horn clause specification. Particularly, the pre and post conditions, which actually determine the circumstances under which the messages can be sent, have a lot in common with logic programming. It is therefore straightforward to consider using structured design techniques from logic programming in LCC specifications. *Techniques editing* [2] is a method used in Prolog, which seems to adapt well up to a certain point to the LCC case. On the other hand, the fact that LCC is a process calculus allows us to consider the use of process oriented design techniques. A good combination probably is to use process-oriented methods to obtain the ”skeletons” used by *Techniques editing*.

Apart from the practical value of the tool, there is also an interesting research aspect. We would to investigate how patterns can be used in languages like LCC. The results can probably generalise to other similar languages. In particular, we will try to

check whether common programming techniques are involved in the process of protocol building and try to define their types if possible. The prototypical tool is influenced by these observations. The tool can then be used to enable engineers to capture and reuse such patterns when constructing protocols. It can also be used to experiment with patterns in LCC in order to obtain a better knowledge of their role in LCC. The suitability of *techniques editing* is another interesting issue from researchers' point of view. Particularly, we wish to evaluate how useful approaches from logic programming can be in LCC. A long-term goal is to define a reasonable set of patterns, which can be used to built LCC protocols in an efficient way, similar to the use of Prolog techniques.

In the following sections, after discussing some background material on LCC and *techniques editing*, we describe an attempt to develop a tool for building LCC protocols by exploiting common patterns appearing in them. The organization of the thesis is as follows:

- In chapter 2 we describe LCC and its applications
- In chapter 3 we present an approach to the structured building of programs in Prolog
- In chapter 4 a proposed method for the structured building of LCC protocols is described
- In chapter 5 some implementation issues about the prototypical tool are discussed
- In chapter 6 we evaluate the proposed method with regard to the features of our editor
- Finally, in chapter 7 we conclude by summarising the outcome of our effort.

Chapter 2

The LCC language

2.1 Coordination in multi-agent systems

Multi-agent systems (MAS) are a promising approach for the development of large scale applications in open environments like the web. This kind of system introduces a new approach for developing web applications in which the systems are constructed by a number of autonomous components, which cooperate to achieve the overall goals of the system. The fact that these components are autonomous allows components built from different teams to be assembled [7]. However, the development and deployment of MAS raise several problems such as ensuring that the interaction conforms to the rules, while the agents maintain their autonomy, ensuring that an agent can determine its role in the interaction without having to monitor the whole interaction continously and sharing the constraints between agents [10].

The first issue that must be tackled for the development of MAS on the web is the standardization of the communication between agents. The most popular answer to this is the development of Agent Communication Languages (ACLs). Today, the two most popular languages that have been proposed, are the Knowledge Query and Manipulation Language (KQML) [22] and the Foundation for Intelligent Physical Agents Agent Communication Language (FIPA-ACL) [14]. Both languages adopt the theory of speech acts [16] for the interaction between the agents. In particular, these languages define message types such as ”inform” or ”ask” messages for communication between agents. These message types correspond to performatives which define different speech acts. Note that the purpose of these languages is to provide a standardized way of knowledge exchange between the agents. They are not supposed either to define the content of the messages or to determine when a message should be sent.

Another issue in order to achieve meaningful interaction between agents is to determine the conditions under which the interaction must take place. Since a MAS is actually a society of autonomous agents more or less similar to human societies, the agents in a MAS have to adopt some conventions and follow some rules in order to be able to operate as a member of the society. These conventions are said to represent the *social norms* of the society and collections of these related to a specific task form an *agent protocol* [12]. Two of the several approaches for specifying agent protocols are the Electronic Institutions (EI) methodology [17] and the Conversation Policies approach [18].

The second of the two approaches seems to have attracted the attention of many researchers interested in MAS as it provides a comprehensible approach to the engineering of such systems. A formalism of this methodology is given in [17]. The main concept of EI approach is the representation of a MAS as an institution similar to the ones that the humans form. The main features of an electronic institution are the roles, the scenes, the dialogic framework, the performative structure and the normative rules. The original goal of this effort as it is stated in [17] is ”the design and development of architecturally- neutral electronic institutions inhabited by heterogeneous (human and software) agents”. The EI methodology is widely recognized as an important effort on specifying MASs and it is accepted that it manages to answer key coordination issues in MAS.

Although the EI approach tackles most of problems, there are several weaknesses in this method. Some of these weaknesses are highlighted in [12] and [7]. In particular, the lack of a mechanism for protocol dissemination to new agents that enter the Institution, the static definition of the agent protocol which causes problems when we do not exactly know in advance which the next steps of the protocol should be, the fact that in practice, administrative agents must be used to ensure the synchronization in the Institution and the fact that EI approach focuses on the global state of the interaction and not on satisfaction of constraints on individual agents, are some of the issues that are considered as drawbacks of the EI approach. It is therefore obvious that although EIs are of significant importance for the MAS society, there are still issues to be solved for the deployment of efficient MASs in an open environment like the web.

An interesting approach that promises to overcome these problems is the use of process calculus for defining a protocol language. This technique has been further developed to allow constraints to be applied on the agents and therefore to provide a complete framework for the coordination of MASs that seems to have desired properties

that the EI approach lacks. Several aspects of this technique, such as the application of model checking techniques to protocols written in the language and the use of the language to coordinate web services, have been presented in a series of publications, making this approach even more attractive. In the rest of this survey is described the basic concepts of the method and the way that it can be used as mentioned above. Moreover, an overview of some research work that is related in some way is given.

2.2 Use of concepts from logic programming and process calculus in the coordination of multi-agent systems

The first publication of the series that introduces the use of techniques from logic programming in multi-agent coordination problem, is [11]. Particularly, a brokering method based on capability descriptions is introduced. The capability description language that is defined, is based on predicate logic in a style derived from logic programming is employed in this method. Apart from the capability language, a correspondence language is introduced in order to represent correspondences between agents' capabilities. The brokering mechanism, given a query, constructs some brokerage structures, which actually describe what information must be retrieved from where and then assembles a sequence of messages that must be sent to the relevant agents, in order to accomplish the given goal. Even though there is an implementation of the method, it is clearly stated that there are a number of issues that have to be further researched. Namely, the choice between alternative brokerage structures, the method for describing the correspondences, the use of a more sophisticated performative language, the connection between the actual implementation of an agent and the capabilities it advertises and some drawbacks of the current implementation are identified by the authors as points of future improvement.

It is clear, as the authors recognise, that the method described above has to be explored in more detail, in order to become a competitive solution for brokering in multi-agent system. Especially, the use of a more expressive performative language and probably one of the standardized ones (i.e. KQML or FIPA-ACL), the direct mapping of the agents' ontologies to create the correspondences and a more sophisticated implementation seem to be quite important issues for the deployment of such a mechanism in an open environment like the web. However, this approach is considered

very important not only for the advantages of the mechanism itself but also as a part of a wider approach for the coordination of MASs which adopts methods from logic programming. Moreover, the major issues which have to be improved are not shortcomings of the approach at the theoretical level, but simplification assumptions that are totally acceptable for research work in its early stages.

Another intersting method is the one introduced in [12]. This method is derived CCS process algebra and aims to the specification of flexible protocols for the communication between agents. A language for describing such protocols is also defined. This language define a flexible protocol that consists of a number of agent protocols while the agent protocols consist of terms connected by operators. The terms can be performatives, agents or empty actions and the operators are connectors which define the exchange of messages and the exact sequence. The protocol that defines the agents' interaction is described in the protocol language and disseminated to the agents during the dialogue as a part of the messages that they exchange. The protocols are considered to be flexible in the sense that they clearly overcome the restrictions of the static definition and dissemination of protocols that the EI adopts.

The advantages of a method like the one described in [12] are significant. The comparison with EI which is a well established methodology for developing MASs shows that the proposed method gives solution to issues that are quite important for deploying MASs in open environments. The lack of a dissemination mechanism, the static definition of the dialogues and the need for centralized synchronization are all quite crucial for an environment were neither the agents that will participate, nor the exact steps of the interaction, are known at design time. This is definitely a potential problem for the web case and therefore the proposed method is consired to be an improvement at least for these kind of cases. Moreover, the fact that the flexible protocol language is formally defined and based on process algebra will allow the application of several formal methods such as model checking of the protocols. Since this close relation with the CCS process calculus is said to be intentional, the application of such formal techniques can be expected as future work.

Further research on the concepts presented so far lead to the work presented in [15]. The method described in [15] attempts to implement dialogue games using a protocol language based on the one presented in [12]. This approach maintains all the advantages introduced in [12] and extends the protocol language to enable it to represent dialogue games. The addition to the protocol language of preconditions and postconditions, which determine what should be true for an agent before and after

a message is sent, restrict the agents' behavior within the dialogue. In other words these are constraints that represent the rules of the dialogue game. By satisfying the conditions the desired behavior of the agent within a dialogue game is achieved. Since these conditions are part of the protocol which is passed from agent to agent with every message, all the agents obey the rules of the game without any need of global state knowledge or global synchronization. The only requirements for the agent the understanding of the protocol language and the satisfaction of the conditions. Another important modification compared to the method in [12] is that messages apart from the actual performative and the protocol, include the instance of the dialogue as it actually occurs. This instance is called the dialogue clause and is used as a history of the dialogue, a marker of the current state and a record for the current values of the variables.

The proposed method can be considered an attractive alternative to the methods that EI use to define and apply the dialogic framework. Apart from the advantages that has already been identified in [12], this method inherits additional features like the ability to allow an agent to take place in concurrent dialogue games. The main achievement of this work is the introduction of a method for specifying agent protocols which separates the protocol from the communicative model itself. The fact that the only necessary knowledge is the protocol itself minimises the requirements that an agent must meet in order to participate in the interaction. This feature of the method is considered important since it is desireable to keep the agents as autonomous and independent as possible.

The result of the research work described above is the formal definition of the Lightweight Coordination Calculus (LCC) [10]. The abstract syntax of the language is shown in figure 2.2. One aspect of the effort, as it is stated by the author, is the invention of a logic programming language that provides an overall architecture for the coordination of multi-agent systems. The whole architecture is described by showing how the main issues of developing MASs are tackled using techniques taken from programming. In particular, it is explained how:

- the interaction model can be represented in process calculus,
- the social norms that define the message passing behavior of the agents can be applied by the satisfaction of mutual constraints individually to the agents,
- the state change of the interaction maps to the unfolding of a clause with respect to the protocol,

$$
\begin{aligned}
Framework &:= \{Clause, \ldots\} \\
Clause &:= Agent :: Dn \\
Agent &:= a(Type, Id) \\
Dn &:= Agent \mid Message \mid Dn\ then\ Dn \mid Dn\ or\ Dn \mid Dn\ par\ Dn \mid null \leftarrow C \\
Message &:= M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid C \leftarrow M \Leftarrow Agent \\
C &:= Term \mid C \wedge C \mid C \vee C \\
Type &:= Term \\
M &:= Term
\end{aligned}
$$

Where *null* denotes an event which does not involve message passing; *Term* is a structured term in Prolog syntax and *Id* is either a variable or a unique identifier for the agent.

Figure 2.1: Syntax of LCC dialogue framework ([10])

- the expansion of the distributed clause coordinate the agents,
- the interaction scope corresponds to constraints applied to the values that the variables can take and
- the brokering can be achieved by providing additional information for the agents (i.e. capability descriptions)

There are also several more details about LCC which must be highlighted. An interesting point is the fact that the satisfaction of the constraints include in the protocol can be done either internally using the agent's mechanism or externally by using shared knowledge in the form of Horn clauses passed with the messages. Another point is the fact that all the requirements of LCC from an autonomous agent, can be encapsulated in a module, which acts as an intermediary between the agent and the communication medium and supplies the following: an encoder/decoder for the translation between the message language and the LCC expressions, a protocol expander and a constraint solver. These points are very important, since they make clear that the method achieves to separate the protocol from the internal architecture of the agent. The fact that the impact on the engineering of the agents has been minimized is probably one of major advantages of the whole approach.

After having described in general the overall architecture behind LCC, it is worth

summarising additional research that followed from this approach. The first one that is discussed is presented in [3]. In this case a language based on concepts described before is defined (MAP language). It is shown how dialogue protocols written in this language can be verified using model-checking techniques. Particularly, the SPIN model checker is used and therefore a translator between the protocol language and PROMELA (the language that SPIN uses to describe models). The current translation to PROMELA allows only simple properties of the protocols, like termination, to be verified, but there is a way of increasing the number and the complexity of the properties by providing a richer translation to the model checker. The contribution of this work is considered to be significant since it allows the development of reliable protocols by automated verification.

The use of a language which shares most of the techniques used in LCC for coordinating agent-oriented web services is presented in [9]. This work describes how the concept of languages based on process calculus can be used for the coordination of web services. The idea of using these methods in web services leads to a comparison with standardized markup languages for describing web services and especially DARPA Agent Markup Language for Services (DAML-S) [21]. The comparison between the two languages leads to following conclusions: DAML-S can be more expressive in describe the structure of the services while the proposed approach is better in describing the interactions, the proposed language lacks the automatic discovery features and finally the fact that DAML-S is mostly like a type definition language while the proposed is more like a logic programming language. This comparison actually makes clear the different concepts from which the two languages are derived. DAML-S is a standard emphasising the use of typed specification for service discovery, while LCC is an attempt to desrcibe service coordination in as style close to logic programming. Hence these two are, to an extent, complementary.

Another piece of work that uses the MAP language (also used in [3]) for the verification of multi-agent web services is presented in [4]. Since the strategy for applying the model checking is from the theoretical point of view the same as in [3], the most interesting technique introduced in [4], is the approach of constructing multi-agent systems from web services. The approach that is used is the constructions of a close-coupled MAS separate from the web services, in which each agent acts on behalf of a web service. By coordinating the agent in the MAS, the coordination of the services that they represent is achieved. This technique seems to be a good choice for the development of agents based on web services since absolutely no modification is needed on

the web service (it can be a simple web service described in a well known standard). [5] is another case in which this approach for coordinating web services is adopted. In this case, the objective is the design and enactment of e-science experiments which involve a number of different web services to be accessed in a particular sequence.

2.3 Future directions of research in LCC

After describing some different approaches to some problems that the LCC architecture is also involved, it is straightforward that there is no other effort to provide such a general purpose mechanism for coordinating multi-agent systems. The only approach that attempts to provide such specifications, is the EI architecture, which has several shortcomings that we have already discussed. Therefore, the conclusion is, that LCC is the most promising approach its kind at the moment. Some of its important advantages are:

- the flexibility it provides by defining protocols at run-time,
- the fact that the agents participate in dialogues without having to adapt their beliefs or to share any internal knowledge,
- the minimal requirements that an a agent must meet, in order to use the protocol and
- the fact that the methods it employs are well known techniques from logic programming, which have been studied for many years and therefore they are quite efficient solutions.

All these features make LCC appropriate for the case of open, heterogenous systems like the web. In these systems the minimization of the assumptions about the participants' internal processes is the only choice and this is why LCC seems to provide an architecture for the coordination in such systems.

There are a number of issues related to LCC that have to be further researched. Most of these issues are raised in almost every publication of the series. The most important of them are the following:

- The ability of the protocols to be adaptive. This is about allowing the agents to modify the protocol in a safe way acceptable by all the participants, during the dialogue. This will actually result in the evolution of the protocol as the interaction goes on

- The application of more sophisticated ways of constraint management
- The introduction of techniques to make the protocols fault-tolerant
- The verification of more complex properties of the protocols using model checking

Continuing work is not limited to the issues mentioned above. Since this approach has been applied in different ways, there is also work to be done on issues that have to do with specific applications. Examples are the model-checking case and the e-science experiment design. In the first one, the attempt to verify more properties of a protocol is considered important and the addition of a translator that generates a protocol description from a workflow design in the second will be a significant achievement towards an efficient solution for the specific problem. Since these observations are valid only within the specific context of the problem, it is considered vital to be described at this point. As mentioned before, most of them are highlighted in the relevant publications.

Chapter 3

Techniques editing

3.1 Techniques in Prolog

A *programming technique* is, as it is described in [20], a common to the programmers code pattern, specific to a particular language, but irrelevant to the algorithm or the problem domain, which is regularly used. The use of such *techniques* is very common in logic programming. Expert users in Prolog are aware of constructs like the ”accumulator pair” and use them regularly. A good example, taken from [2], is to consider the implementation of ”quick” reverse in Prolog:

```
rev([],R,R)
rev([H|T],R0,R) :-
    rev(T,[H|R0],R).
```

The predicate above consists of two parts. The first one performs the recursion down the list

```
rev([],...)
rev([H|T],...) :-
    rev(T,...).
```

and the second one builds a list during the recursion (accumulator pair)

```
rev([],R,R)
rev(...,R0,R) :-
    rev(...,[H|R0],R).
```

These two parts are Prolog techniques in the sense that they are common code patterns used in wide variety of situations regardless the algorithm being implemented.

A methodology for constructing logic programs using techniques is proposed in [19]. The key points of the method, as they are summarized in [2] are the following:

- The construction of the program is based on the *skeleton*, which determines the control flow of the program.
- There is set of methods for performing simple tasks. These methods are called *additions*.
- *Additions* can be applied to the initial skeleton in order to obtain an *extension*.
- *Extensions* can be composed to produce fully functional logic programs.

Skeletons are used as the basis for constructing programs using this methodology. The control the flow of the program in general. As it stated in [23], skeletons are basic Prolog programs processing inputs in a simple way. The can also be refined futher, in order to produce more complex programs. An illustrative example of a skeleton taken from [23], is the following:

```
s([]).
s([X|Xs]):-s(Xs).
```

This is fould in Prolog programms when traversal of a list is required. We can see how this skeleton is used as the first argument of the list reversal:

```
reverse([],R,R).
reverse([X|Xs],R0,R):-reverse(Xs,[X|R0],R).
```

Note how the *reverse* programm can be constructed by adding new arguments to the clause. The role of the skeleton is to iterate the process by obtaining values from the input and terminating the recursion according to the specified conditions. *Techniques* can be applied to it in order to perform more complex operations.

In this methodology as it is adopted in [23], a *technique* is described as a sequence of single-argument programs called *additions*. This can be done by isolating the arguments and subgoals of which a *technique* consists. *Techniques* are applied on a *skeleton* by adding these arguments and subgoals in order to produce more complex programs. An example illustrating these relationships is also presented in [23]. Consider a *technique* for counting the number of items during a loop:

```
t(...Count,Total):-
        .
        .
        Total = Count.
t(...Count,Total):-
        .
        .
        Count1 is Count+1.
        t(...Count1,Total)
```

This *technique* can be thought as being composed be the following two *additions*:

```
q(Count):-
        Var = Count.
q(Count):-
        Count1 is Count+1.
```

and

```
r(Total):-
        Total = Var.
r(Total):-
        r(Total).
```

If the above *technique* is applied to the basic *skeleton* for travesing lists, which we have also described above, the result is the following Prolog program:

```
s([],Count,Total):-
        Total = Count.
s([X|Xs],Count,Total):-
        Count1 is Count+1.
        s(Xs,Count1,Total).
```

This program essentially counts the elements of list given as the first argument. The result of applying the *technique* to the initial *skeleton* is said to be an *extension*.

3.2 Applications of techniques editing

There are several aspects of Prolog programming, in which these common pattern can be useful. In [2], Prolog editing, automated program analysis and program tracing are

said to be fields, in which a *techniques* based approach can be usefully applied. We are mainly concerned about the application of *techniques* in editing (*Techniques editing*). There have been several attempts for developing an editor which enables the building of programs using *techniques*. We are going to describe briefly two of these editors. Note that we will refer to this description later, when trying to compare them with the outcome of project.

Robertson's editor, described in [6], follows the methodology described above. It basically aims to help novices learn Prolog, by becoming aware of common programing techniques that Prolog experts exploit when writing programs The program is constructed by applying different combinations of techniques on the initially selected skeleton. A set of predefined skeleton and techniques are provided. These are represented using a simple notation proposed in [6]. Information about valid combinations of clauses and about the mapping of arguments are included in the proposed representation. Other information aiming to help the user understand the effect of applying a pattern is also stored. This allows the editor to provide guidance by suggesting appropriate techniques for each case and judging the user's decisions. by testing the correctness of the program. The interface although it is not graphical, it is considered to be very usable, providing all the information that is needed by its user. The library of pattern is limited to a small set of skeleton and techniques. Despite the limitations regarding the number of available patterns, the editor described in [6] is a valuable implementation of the *techniques editing* approach.

The second techniques-based editor we are going to discuss, is named Ted and it is extensively described in [1]. Techniques in Ted are defined as relationships between the head and recursive arguments in the recursive clauses of a program. Clauses are thought to constist of a combination of such techniques, sharing variables between them. This approach is quite different from the skeleton-addition approach desrcibed before. As it is pointed out in [2], there are some limitations about the patterns that can be described in Ted. In particular, mutually recursive predicates and doubly recursive clauses are not supported. Moreover, the allowed data-structures are limited to lists, atoms and numbers. Although Ted has a graphical interface for applying patterns, it does seem to provide the amount of information provided by Robertson's editor. It also checks the suitability of a specific pattern by checking the given constraints, but the user is not guided through the steps of the refinement. Despite its limitations in the representation of patterns, Ted demonstrates that techniques can be useful for teaching Prolog to novice users. The study of both the editors suggests that a techniques-based

approach to editing has several advantages and should be applied in cases when a sufficient set of skeletons and additions can be defined.

Chapter 4

Editing operations

4.1 Recognizing the problem

Although LCC has a lot in common with logic programming languages, there are several differences which will make the effort of directly applying *techniques editing* in the LCC case hard or even impossible. The most important of these differences is that LCC is a process calculus. Its syntax is similar to logic programming, but the problems it is supposed to tackle are different from those of logic languages. This means that there is no guarantee that the specific patterns or even the whole method that has been proved to be useful in the Prolog case will be appropriate for LCC. Moreover, we can not be sure yet that useful frequently occurring patterns exist in LCC protocols. This means that a new method particularly suitable for LCC might be designed based on similar approaches in logic programming.

Another difference from related work in *techniques editing* is the purpose for building the editor and the whole method upon it is based. Existing Prolog editors aim to help novices to get familiar with logic programming using common programming patterns. In our case the goal is to provide a useful tool for knowledge engineers in order to enable to the building of flawless protocols more easily and quickly than simply writing the protocol in LCC. The use of patterns is expected to speed up the process of building and reduce the number faults by reusing parts of protocols which are known to be reliable. Although this difference in the objectives is not expected to affect our effort as much as the fact that we are dealing with a process specification language, it is likely that several decisions may be different than the ones taken in Prolog editors.

4.2 The LCC case

4.2.1 Types of patterns

Since the patterns used in techniques editing based editors does not seem directly applicable to LCC, the first step is to try to identify the different kinds of protocols involved. After studying a number of existing LCC protocols and particularly the example protocols described in [9], [10] and [8], three kinds of interesting patterns were identified:

1. *Skeletal*, which describe the structure of a clause
2. *Role refinement*, which describe a clause in detail
3. *Clause interaction*, which describe the interaction between clauses

These patterns describe a clause or a set of clauses in variable detail and therefore we expect to see a major difference in the frequency of occurrence for each type. It is also the case that combinations of the above types may be useful in particular cases, but it is generally true that defining a new pattern as a combination of existing ones will make the new one too specific to have an acceptable level of repeatability. In the next three sections we are going to describe the features of these types of patterns and the role that they can play in assisting the process of building LCC protocols.

Skeletal patterns

The first type of patterns appearing in LCC protocols are the ones which can provide early skeletons for a clause. Their purpose is to determine the general structure of the clause and therefore the behavior of the clause in general. These patterns are quite abstract in the sense that the details of a role are not specified. Only the flow of control within the role definition is specified. They can be said to play a similar role with the skeletons in the *techniques editing* approach. An example of such a pattern is presented below:

```
a(R,x) :: (<def> then a(R,x))
                    or
               null <- <con>
```

In this example, `<def>` statements represent definitions which are not yet specified and `<con>` statements represent unspecified conditions. They can be filled with any definition or condition respectively according to the LCC syntax. `R` is the role being

defined in the clause, `x` is the agent identifier and `null` represents the null message. The pattern presented above represents a recursive clause and this is expressed by the fact that the role remains the same. The recursion itself is not specified, nor is the condition for terminating the recursion. The role of such a pattern is to define a clause as a generic recursive clause. The details can be specified later either by applying a more detailed pattern or by filling the missing parts manually. Such patterns are expected to be rather useful due to the fact that many different clauses seem to follow the same or at least very similar general structure. By experimenting with the detail, a reasonable set of skeletal patterns with high repeatability can probably be constructed.

Role refinement patterns

The second type of patterns that we were able to identify, are more detailed and aim to refine the initial skeletons in order to get a more precise definition of the role defined by the clause. Several role refinement patterns can be applied to a skeletal pattern in order to refine a clause. The result can be either a fully specified clause or a clause which need to further manual refinement. An example is the following:

```
a(F(A1...An),x) :: (<def> then a(F(A1...An-1,An'),x))
                                    or
                                null <- <con>
```

In this case `F` represents the predicate of the role and `A1...An` are its arguments. The predicate `F` remains the same denoting the recursive nature of the clause. The arguments `A1...An-1` do not change as well. The fact that only `An` is replaced by `An'` shows that the example represents a recursive clause in which the recursion in made over one of its arguments. Comparing it with the skeletal example, it is obvious that the role refinement describes the clause in greater detail. A pattern like that could be applied to the skeleton presented before in order to specify that the recursion concerns one of arguments. The detail of such patterns is a major issue, since it is closely related with their repeatability (too detailed patterns can be too specific and therefore they will occur rarely). It is also interesting to see how they should be related to skeletal patterns in order to make their combination both effortless and efficient.

Clause interaction patterns

Unlike the first two types of patterns, clause interaction patterns involve more than one clauses. They aim to capture common patterns of interaction between clauses such as

message passing. They can be considered as skeletal in the sense that they can be used as a skeleton for building the agents involved, but they can also be rather detailed in specifying the messages. An illustrative example of such a pattern is the following:

```
a(R1,x) :: <def> then
           M=>R2

a(R2,y) :: M<=R1 then
           <def>
```

As in the skeletal case, `R1` and `R2` represent the agent roles. `M` is the message which is sent by the agent `R1` and received by agent `R2`. The general structure can be described as an interaction in which `R1` sends a message after doing some processing and `R2` will do something else after receiving that message. In this example the message and the agent roles are abstract but they could have been more specific. Patterns like this can be very useful in LCC protocols. They are valuable both in the sense that they can save effort required for building the protocol (the effort for building the above clauses manually is not insignificant) and because of the increased frequency in which they appear in protocols (every message has to be caught by some agent).

4.2.2 Use of patterns in LCC

There are two points about patterns in LCC that are worthwhile to discuss further. The first one is that the distinction between different types of patterns is not very clear. Considering the examples mentioned before, we can observe that the role refinement example pattern can be used as a skeleton for building an agent from scratch. The increased detail of the role refinement patterns will probably reduce the possibility that such skeletons will occur occasionally, but if so they are very useful since we can get a more detailed definition in only one step. The patterns seem to be distinguished more by the level of their detail than by its role in building the protocol. The conclusion drawn by this observation is that it is probably a good decision not to distinguish the different types of patterns in such a way that restricts their usage in protocols. Especially the possibility of using detailed patterns as skeletons is likely to be worthwhile.

The second point is the balance between the detail of the patterns and the frequency of occurrence. It is true that more detailed patterns are more useful then abstract ones in the sense that they can do more for us when they are applied. The problem in that case is that if a pattern becomes too specific it is less likely to occur frequently. It is therefore

obvious that the optimal level of detail for each pattern type has to be determined. A decent solution is to experiment with different levels of detail and evaluate the result by counting the frequency with which the patterns occur when trying to build a new protocol from scratch. It is rather intuitive that in order to do that the method and the tool, which are going to be built, have to provide a convenient way to define patterns. We should bear that in mind when making decisions about them.

4.3 A method for structured building of LCC protocols

4.3.1 Design of a suitable method

After considering the observations made about the types of protocols appearing in LCC and their use in building protocols, the result was a pattern-assisted incremental method. It is said to be pattern-assisted in the sense that it does not enforce the use of patterns in the process of building. Patterns are used by the knowledge engineers only in cases where they are expected to reduce the required effort according to their judgment. The method is basically an incremental approach in which missing parts of a clause can be filled in with statements according to the LCC syntax. This is flexible in the sense that the engineer is allowed to build everything that would have built when writing the protocol by hand. The tradeoff is that the effort is probably also comparable with the case of writing protocols by hand. This is where patterns are expected to assist the process of building by reducing the effort required. The application of a pattern in any stage of the development (either in specifying an initial skeleton or in refining the role) will save us from considerable number of incremental steps. It is quite obvious that a combination like this will be able to give us the required flexibility while making in the same time the process of building easier and effortless.

The reason for choosing such an approach has mainly to do with the objectives of the project. As we have stated before, one of the reasons that motivated us was the lack of support to LCC from the engineering point of view. It is therefore expected that the outcome of the project will have considerable value for the knowledge engineers who wish to use the LCC approach for developing multi-agent systems. A pure pattern-oriented approach would probably be too risky in that case, since it is not yet very clear what kind of patterns are suitable for LCC and how useful they can be. By choosing to base our work in an incremental approach assisted by patterns when possible, we ensure the value of the outcome from the knowledge engineer's point of view (the

method is very likely to be more efficient than the manual building of protocols by hand) while providing a method which will allow us to experiment with patterns and their use in LCC.

As mentioned before, the role of patterns in the method is to assist the building of the protocol by the reuse of common programming LCC techniques. The patterns supported by the method correspond to the types identified before. There is no strict distinction between them in the sense that it is up to the engineer to decide how a specific pattern will be used. The reason for that has already been discussed and it is considered an advantage that the engineer has this kind of freedom. Another feature supporting the flexibility of the tool is the convenient way in which patterns can be specified and stored for later reuse. The same incremental approach used for building the protocols is used for specifying custom patterns. In fact, patterns are unfinished clauses or sets of clauses which are stored at the right level of abstraction in order to be appropriate for reuse. This approach to the use of patterns seems to serve the goal of reducing the effort for building protocols quite well without violating the flexibility of the method.

4.3.2 Applying the method to a structured editor

The ideas described above has been implemented and the result is an editor with a graphical interface providing the required functionality. We prefer not to get into the details of the interface and its implementation at this point. Our aim is to demonstrate how the method can be used in order to build an example clause. We will use an illustrative example which demonstrates the different steps and operations required for building protocols using the proposed method. The example is taken from [9] and represents the role of the locator in the protocol. The locator in the example protocol interacts with the finder F in order to obtain the Web locations $\mathcal{L}$ of people in set S by asking recursively the finder F for the location X_l of person X_p. The clause, as it is

presented in [9], is the following:

$$
\begin{aligned}
&a(locator(F,\mathcal{S},\mathcal{L}),L) :: \\
&\quad \left(\begin{array}{l}
ask(locate(X_p)) \;\Rightarrow\; a(finder,F) \;\leftarrow \\
\qquad \mathcal{S} = [X_p|\mathcal{S}_r] \wedge \mathcal{L} = [X_l|\mathcal{L}_r] \;\; then \\
inform(located(X_p,X_l)) \;\Leftarrow\; a(finder,F) \;\; then \\
a(locator(F,\mathcal{S}_r,\mathcal{L}_r),L)
\end{array} \right) \\
&\quad or \\
&\quad null \;\leftarrow\; \mathcal{S} = [] \wedge \mathcal{L} = []
\end{aligned}
\tag{4.1}
$$

For the needs of the scenario we will assume that two patterns have been defined and are available for use. The first one, which is going to be used as a skeleton, is presented below:

```
a(<func1>(<arg1>,<arg2>),<arg5>)::

    (<def6> then
    a(<func1>(<arg1>,<arg6>),<arg5>))
    or
    (<def4>)
```

The `<funcX>` statements represent predicate names which are left to be specified later. This is essential when defining patterns since the names given to variables and predicates must be specified in the context of the protocol in which the pattern is being applied to. Note that the use of `<func1>` in both the head of the clause and the agent definition, places the constraint that these predicate names must be the same. In other words, the pattern defines a recursion since it calls the same role with different arguments. The number of arguments involved in the recursion will be determined by the difference between their argument lists.

The `<argX>` statements are somehow different from the predicate names. They represent arguments which have not been specified yet. An argument can be specified as a variable or constant, a predicate, a Prolog-like list or a sequence of other arguments (the sequence `<arg1>,<arg2>` is itself an argument). Considering all these, the pattern requires that the two agent definitions (the one in the head of the clause and the one in the definition) must have a common part of their argument list (since `<arg1>` can be an argument list itself) and the rest of it should be different. The fact that `<arg5>` (the identifier) is common shows that the agent taking the role will be the same again.

Thinking in terms of a recursion, this structure is the skeleton of most recursive clauses in LCC. `<arg1>` represents the arguments that are not affected by the recursion and `<arg2>` represents the arguments that will change until the next call in order to provide the recursion. The skeleton does not specify the stopping condition of the recursion. Its purpose is to provide a generic structure which will be later refined by another pattern of by incrementally specifying its unspecified parts.

The second pattern contains some more detailed definitions and aims to refine the initial skeleton:

```
a(<func1>(<arg1>,<arg2>),<arg5>)::

    (<def12> then
    <func3>(<arg7>) => a(<func2>(<arg6>),<arg7>)
    <- (<arg1> = [<arg9>|<arg10>] &&
        <arg2> = [<arg12>|<arg13>]) then
    <def10> then
    a(<func1>(<arg10>,<arg13>),<arg5>))
    or
    (null <- (<arg1> = [] && <arg2> = []))
```

It represents a clause taking two lists as arguments. The clause recursively sends a message for each element in the lists. There is no constraint on how the elements of the lists will be used. It just specifies that the role will send messages somewhere for as long as the lists are not empty. A pattern like this could have been used as a skeleton in a case where we need only the two lists to be the arguments for the role. Since this constraint is too restrictive, it is considered useful to provide mechanisms for applying such a pattern on a more generic skeleton. Other patterns can then be applied to same skeleton in order to refine the role further. We are going to show how the second pattern can be applied to first in order to get a more detailed definition for the role shown in 4.1 using the functionality provided by the editor. Finally, we will show how the result of combining the patterns can be refined using the incremental approach to produce the full definition of the clause. The full set of screenshots taken from the editor during the process of building the clause are provided in appendix A.

The first step in the process of building the clause is to use the stored skeletal pattern in order to obtain an initial skeleton. Any set of clauses can saved and reused as a pattern. In our scenario the skeletal pattern is saved as ”recursion” and the detailed

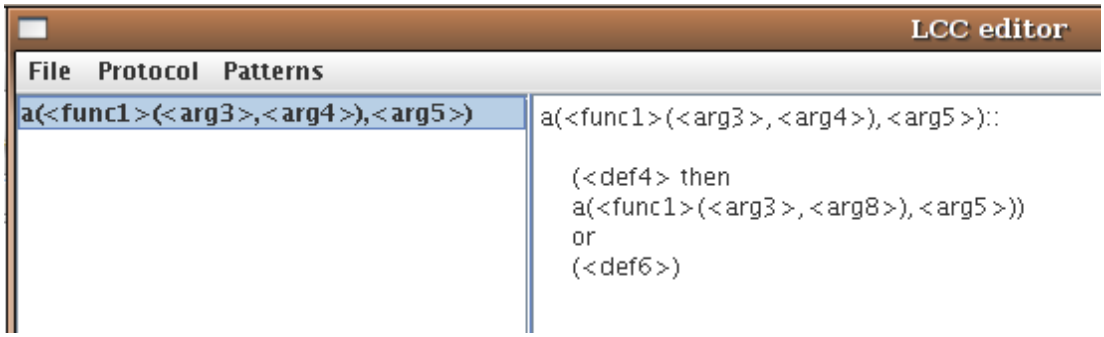

Figure 4.1: The initial skeleton

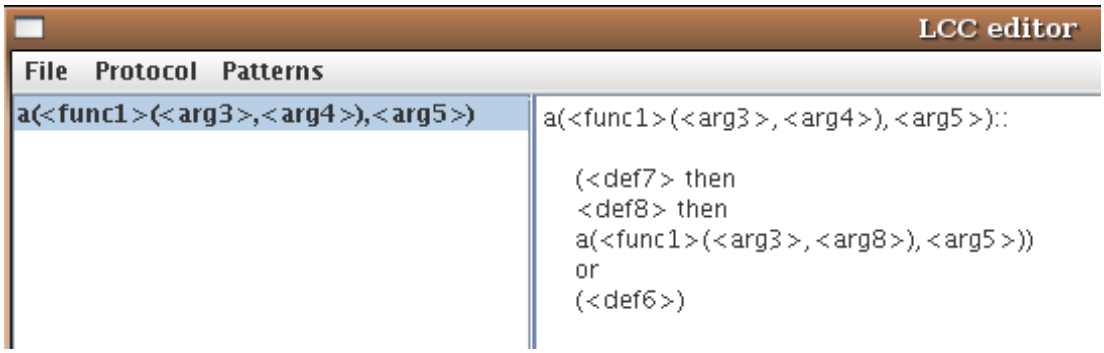

Figure 4.2: The resulting clause after adjusting the skeleton

pattern as ”2ListRecursion”. Using the interface of the editor we can add a new clause to the protocol based on a saved pattern. The interface operations required are shown in figures A.1 to A.4. The result at this point, shown in figure 4.1, is a clause similar to the skeletal pattern.

The next step is to apply the detailed pattern to the initial skeleton. Before doing that, we will have to manually edit the skeleton in order to meet our demands. In particular, we have to replace `<def4>` with a sequence of definitions in order to make the skeleton appropriate for the target clause. It is obvious at this point how the method combines the incremental building of protocols with the use of patterns in order to achieve its goals. The required operations are shown in figures A.5 to A.7 and the result is shown in figure 4.2.

At this point we are ready to apply the role refinement pattern to the existing clause. This will be done by replacing parts (basically arguments and definitions) of the skeleton with others from the detailed pattern. In particular, `<arg4>` ,`<def7>` and `<def6>`of the skeleton must be replaced by `<arg11>`, `<arg12>` , the definition describing the conditional message and the definition representing the stopping condition respectively.

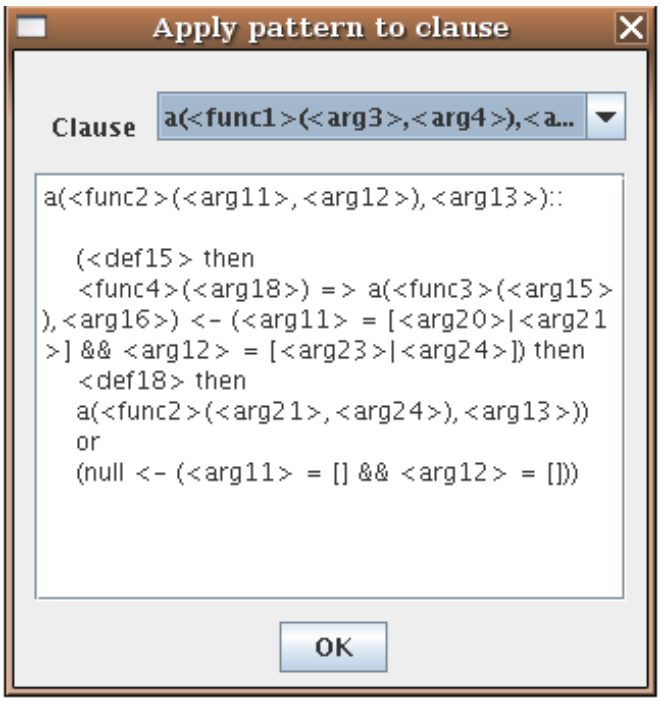

Figure 4.3: Preview the pattern with the new numbering

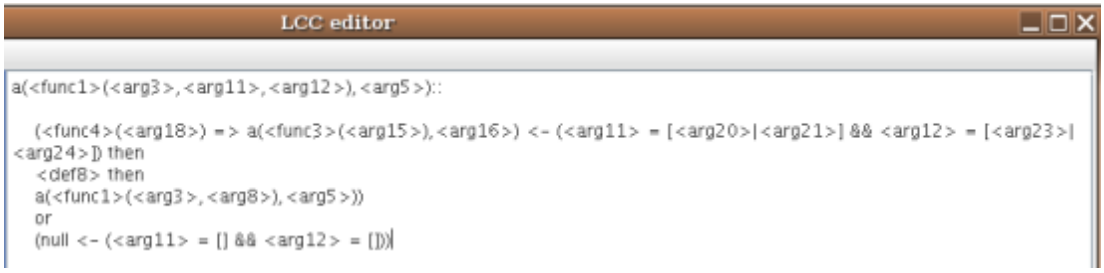

Figure 4.4: The clause after applying the pattern

Note that since the context in which the patterns were constructed and saved is different from the one they are used in, the numbering of the statements in a pattern may be conflicting with the existing clauses. This is why when a pattern is being applied, the numbering of its contents is adjusted to the existing protocol. To avoid any misunderstandings, the resulting pattern, after making those changes, is presented to the user in a preview window right after the file is selected. In our example, the role refinement pattern after these changes is shown in figure 4.3.

Given the changes in the numbering shown in figure A.10, the result after replacing the relevant arguments and definitions of the skeleton with the ones from the role refinement pattern is shown in figure 4.4.

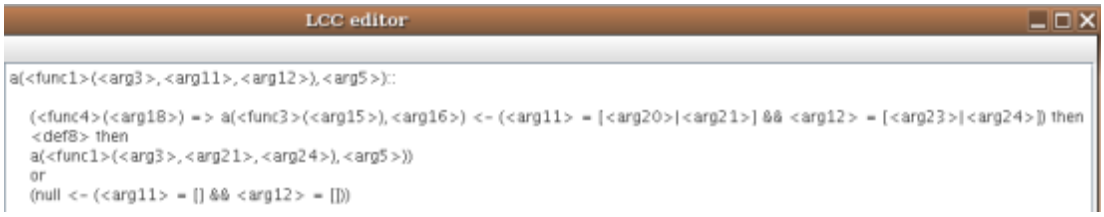

Figure 4.5: The result after completing the application of the detailed pattern

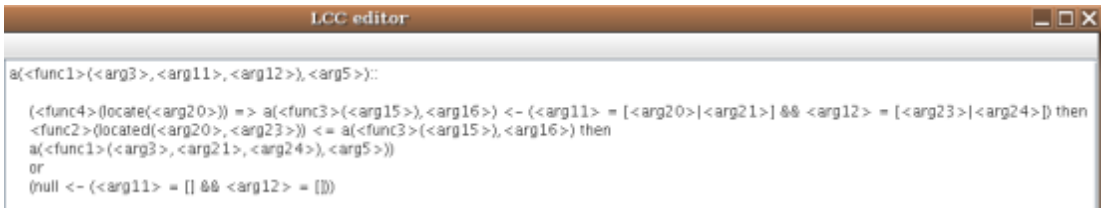

Figure 4.6: The result after the addition of the message

The precise procedure for loading the pattern and making the above mentioned mappings is shown in figures A.8 to A.14. There is one more replacement which must be done manually in order to completely apply the new pattern. It is the replacement of `<arg8>` with `<arg21>,<arg22>` which are the remainders of the two lists. An obvious improvement to the editor, in order to avoid this step, is to allow the mapping of definitions from the detailed pattern to existing specified definitions of the skeleton. In the specific case, it would be useful to be able to map the agent definition in the role refinement pattern to the one in the skeleton. The mechanism for doing that would be similar to the one for mapping the heads of the pattern clauses and is included in the list for future improvements. The operations are shown in figures A.15 to A.18 and the result is shown in figure 4.5.

The next step after applying both patterns is the incremental refinement of the clause. Particularly, we will have to refine the message added by the pattern and to add the second (incoming) message by replacing `<def8>`. For the refinement of the outgoing message we need to specify `<arg18>` as the predicate `locate<arg20>`. The incoming message can be added by replacing `<def8>` with an appropriate incoming message definition. The full set of screenshots for these transformations are shown in figures A.19 to A.33 and the result is in figure 4.6. The clause is essentially finished at this point. The last step is to replace the abstract statements with more specific ones. In particular, `<func>` and `<arg>` statements must be replaced with named functions and arguments according to the target clause. An example of the procedure for doing so is described in figures A.34 to A.35 for `<func>` statements, while the procedure

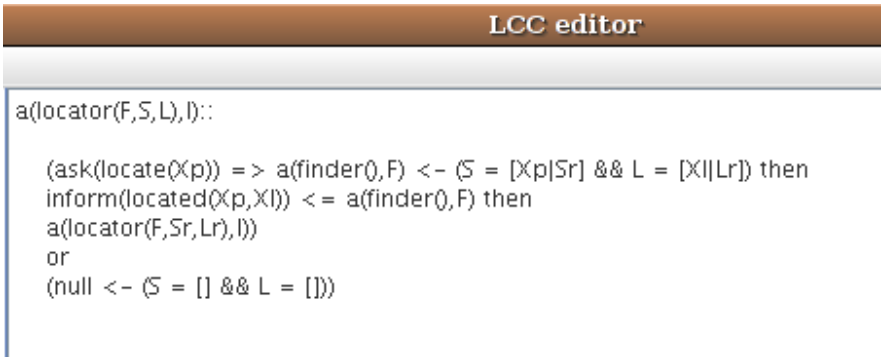

Figure 4.7: The final result

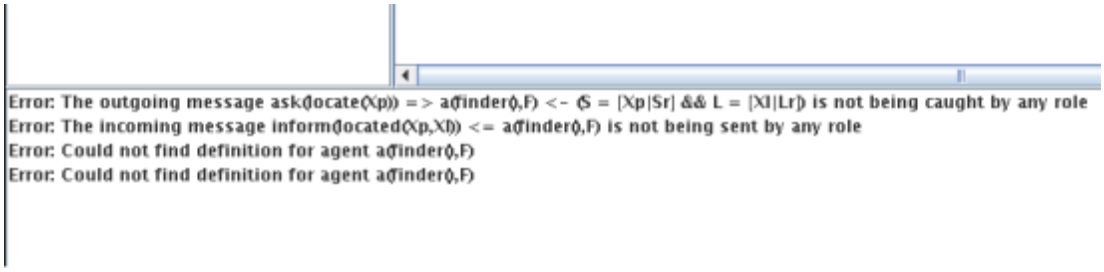

Figure 4.8: The final result

for replacing arguments has been shown before. Note that `<arg15>` is a special case because the role of the *finder* takes no arguments. This can be denoted by specifying `<arg15>` as an empty argument list. The final result after naming all the unspecified parts of the clause is presented in figure 4.7.

An interesting feature of the editor which is not directly related to the process of incrementally building clauses, is the validation of the protocol. The validation checks aspects of the consistency of the message and agent definitions included in the protocol. In particular, it is ensured that for every outgoing or incoming message there a corresponding incoming or outgoing message respectively and for every agent definition appearing in the protocol, the clause defining the relevant role exists. This is considered an important feature since it is known that many errors occur due to this kind of inconsistencies when building LCC protocols manually. In our example, the validation of the protocol returns the result appearing in figure 4.8. We can see the error messages produced since there are no corresponding messages for the ones mentioned in the *locator* definition and the role of *finder* is not defined in our protocol. It is obvious that the validation feature can be a useful tool for the knowledge engineers especially when building lengthy protocols with many roles.

Although the example described above does not illustrate the full range of functionality covered by the editor, it is considered to be a good scenario for demonstrating the use of our method. The editor does not enforce the application of patterns in the way we have described and therefore it is up to engineer to decide how and to what extent the patterns should be used in a particular case. Different degrees of pattern usage may be appropriate for different kinds of clauses. The method we have described seems to be flexible enough from the engineering point of view, while at the same time it provides us a tool to experiment with patterns in LCC. It is true that patterns in LCC may also exist at a different level of abstraction , which is not supported by the described method. However, we believe that the currently supported pattern types and the proposed method in general, provide a pragmatic initial approach.

Chapter 5

Implementation issues

5.1 Design of an editor implementing the method

At this point, we discuss some issues concerning the implementation of our editor. In the example scenario presented in 4, we demonstrated how the method can be applied without getting into the details of the underlying implementation. We used the interface and its operations to construct the example clause, but it is not clear yet how the objectives were accomplished at the implementation level. The purpose of this chapter is mainly to discuss how the internal structure of the editor supports the operations shown before and how the interface uses this structure to achieve the desired result.

Before doing that, we discuss some key engineering decisions made. The first one is the choice of platform and programming language, in which the editor would be implemented. Our decision was to use Java as the programming language. Because of its platform independence, its fully object-oriented nature and the convenient way it provides for building graphical user interfaces using SWING, Java is particularly suitable for this project. We will briefly discuss how our effort can benefit from these features of the language.

Given the fact that Java is a cross-platform language providing the required tools for the development of the editor, we consider that it is preferable not to bound our work to a specific platform, unless there is another strong factor discouraging us from doing so. The object-oriented software development is the dominant approach for building software these days and therefore it is considered an advantage to use a language which natively supports such an approach. Finally, the existence of native language libraries for building graphical interfaces is maybe the most important of the factors leading us to use Java. Since the graphical user interface has a key role for the value of the editor

from the knowledge engineer's point of view, the convenient way in which Java allows the implementation of graphical interfaces without sacrificing portability, is probably the main reason for choosing it.

Apart from the advantages described above, Java has also some major shortcomings. The most important of them are definitely performance and memory efficiency. It is generally accepted that Java is considerable slower than conventional compiled languages like C++, while it also demands greater amounts of memory. Although these disadvantages are not minor, they are not expected to affect our work since our software is not considered to be performance critical or particularly memory consuming. Since we estimate that the main disadvantages of Java will not cause any major problems in our case, there is probably no strong argument against it. In any case, the choice of language is not a key decision in our work. We wish to focus mostly on the method for building LCC protocols and the interface providing the required operations. Any decent choice seems to be good enough for our purpose and Java is surely one of them.

Another interesting issue is the representation of the protocols internally. This is a key issue since it will affect the outcome much more than the choice of language described before. A good representation must be both flexible enough to support the operations required by the method and simple enough to be implementable with reasonable effort. Moreover, the complexity of the code needed to implement it should be kept at a reasonable level. A solution for this problem is to use a class structure describing the LCC syntax. This choice allows us to build an expressive representation using an object-oriented approach, which can ensure that the required code will be well structured. The fact that Java is an excellent tool for deploying object-oriented solutions is also a strong argument for our decision. Using the mechanisms provided by Java, we have built a fairly simple object structure adequate to express the full range of protocols written in LCC. Other solutions for representing the LCC protocols may also exist, but objects and inheritance seem to be beneficial in terms of simplicity and quality of structure of the resulting code. The correctness of our choice will be more obvious in the next sections where we describe a natural way to represent LCC syntax by a hierarchy of classes.

In the following sections we have a closer look at the class structure, the interface and its relation with the underlying structure. Finally, we briefly describe the internal operations required for each step of the example scenario.

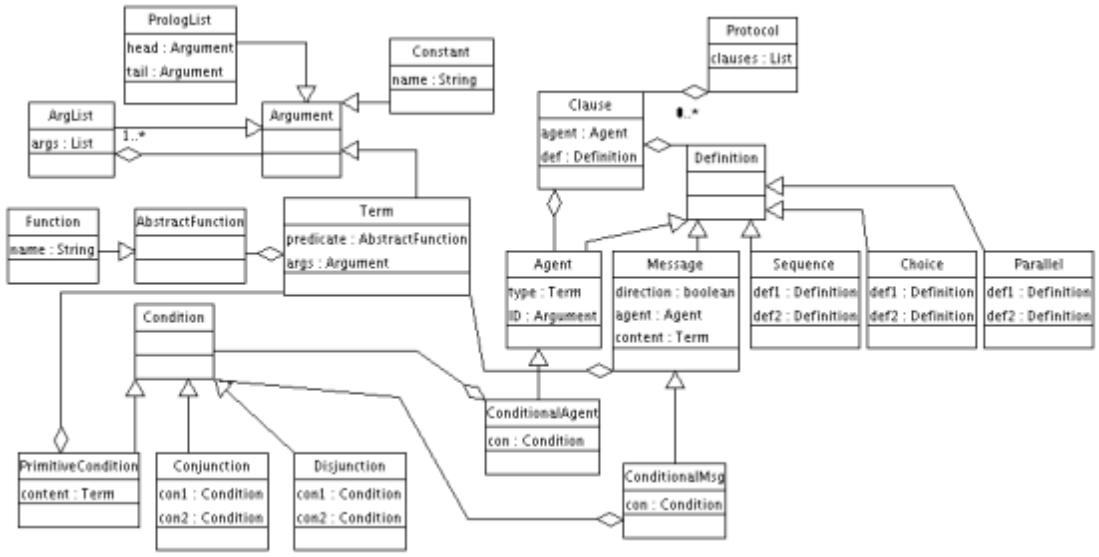

Figure 5.1: The class structure

5.2 Internal structure of the editor

5.2.1 A class structure based on LCC syntax

One of the key components of the implemented system is the representation of the protocols. As we have argued before, we have chosen to built a class structure which represents the abstract LCC syntax shown in figure 2.2. The class structure extensively uses inheritance, and polymorphism is also widely used in order to achieve its goals. A class diagram describing the hierarchy of classes is shown in figure 5.1. The figure gives an abstract illustration of the class structure. Its details such as methods or other members of each will be discussed later when necessary. Before that, there are several issues concerning the structure and its relation to the syntax of LCC which would be worthwhile to discuss. The class structure in general is derived directly from the syntax of the language. The central idea is similar to the one behind the *composite design pattern*. The idea in the *composite design pattern* as it presented in [13], is to represent part-whole hierarchies by composing the objects into tree structures. The goal is to let clients (the graphical interface in our case) handle objects and compositions of objects uniformly. The essence of the solution suggested is to let composite and leaf object to inherit from a common class (the class ”Component” in [13]). The result is that by manipulating the composite and leaf objects as ”Component” objects we achieve to uniformly treat each object ignoring its actual type.

We do not actually claim to have used the composite pattern as described in [13].

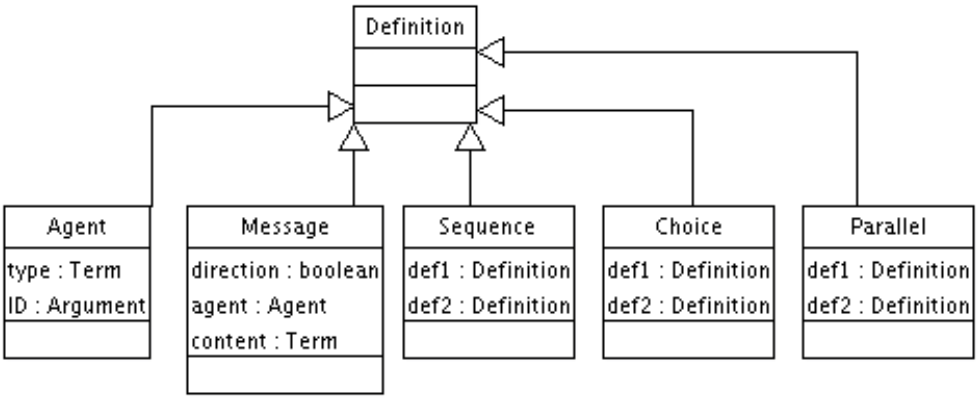

Figure 5.2: The composition of definition

What we have done is to use its basic idea for the purposes of our class structure. In particular, the classes ”Definition”, ”Condition” and ”Argument” play the role of ”Component” in the composite case. For the first two cases it is quite clear from the LCC syntax why such an approach is suitable. A *Dn* in LCC can be an *Agent*, a *Message*, a sequence, choice or parallelism of definitions. In this case, *Agent* and *Message* are leafs in the sense that they do contain other definitions, while the rest are composite since they consist of other definitions (figure 5.2).

The *C* case is similar since *Term* can be considered as leaf while the conjunction and disjunction are composite (figure 5.3). It is worth explaining why the ”PrimitiveCondition” is introduced instead of having the class ”Term” directly inheriting from the ”Condition” class. Note that the LCC syntax defines *Term* as a possible instance of a *C*. The ”Term” class can also be an argument, the role of which will be explained in a while. Since Java does not support multiple inheritance we had to choose a class from which ”Term” would inherit. We have chosen to define ”Term” as a specialization of the ”Argument” class and therefore we introduce ”PrimitiveCondition” as an intermediate class having a ”Term” object as its member. There are no strong arguments why the decision was not the opposite, but we believe that is more natural to consider the *Term* as a special case of an argument than as a *C*.

The ”Argument” case is somehow different in the sense that the composition is not obvious in the LCC syntax. The reason is that the syntax shown in [10] is not an exhaustive one. However, it is made clear that *Term*, which is not defined in 2.2, follows the composition of a term in the Prolog syntax. Since we do not wish, at least at this stage, to get into the details of Prolog syntax, the composition of the *Term* has been determined by its use in existing protocols. The result can be seen in the class structure

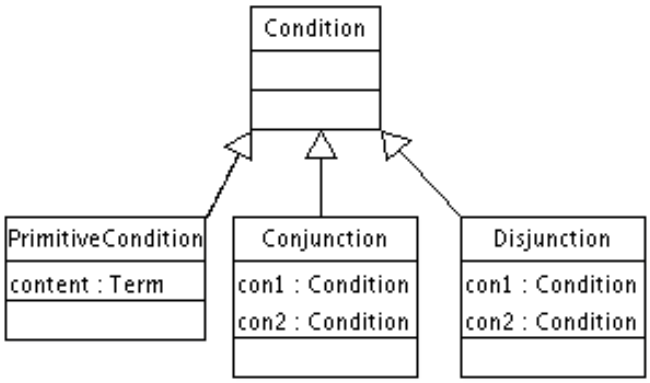

Figure 5.3: The composition of condition

shown in figure 5.4. As you can see, a ”Term” object consists of an ”AbstractFunction” and an ”Argument” object. The concept of function, in general, represents the name of a predicate (e.g. in the predicate $Father(John)$ *Father* is considered to be the function). It is used here in conjunction with the ”Argument” object in order to represent a predicate. The role of the ”AbstractFunction” object and its difference with ”Function” is not clear at this point, but we will get back to this issue later. An argument object can be either a list of arguments (”ArgList” class), a constant or variable (”Constant” class), a Prolog-like list (”PrologList” class) or another *Term*. Figure 5.4 shows how class inheritance and composition is used to express such a structure. The fact that any of the specializations of the ”Argument” class must be treated uniformly as arguments of a predicate leads us use the ”Argument” class in the same way as the ”Component” class in the *composite design pattern*.

A point which has not been made clear yet, is the role of ”AbstractFunction”. The classes ”Definition”, ”Condition” and ”Argument”, apart from playing the role of ”Component”, are also used for another purpose. As we saw in the example shown in 4, there are cases in which parts of the protocol are left unspecified. This feature is essential for enabling the development of patterns using the same representation of protocols. The definition of such statements at the object level can be done by creating instances of the abstract classes ”Definition”, ”Condition” and ”Argument” for the corresponding types of statements. This is a good solution since we are taking advantage of existing classes, thus keeping the class structure as simple as possible. The problem with the concept of function is that there exists no abstract class to play the role of an unspecified function. This is why we introduce the class ”AbstractFunction” from which the ”Function” class inherits. By creating an instance of ”AbstractFunction” we

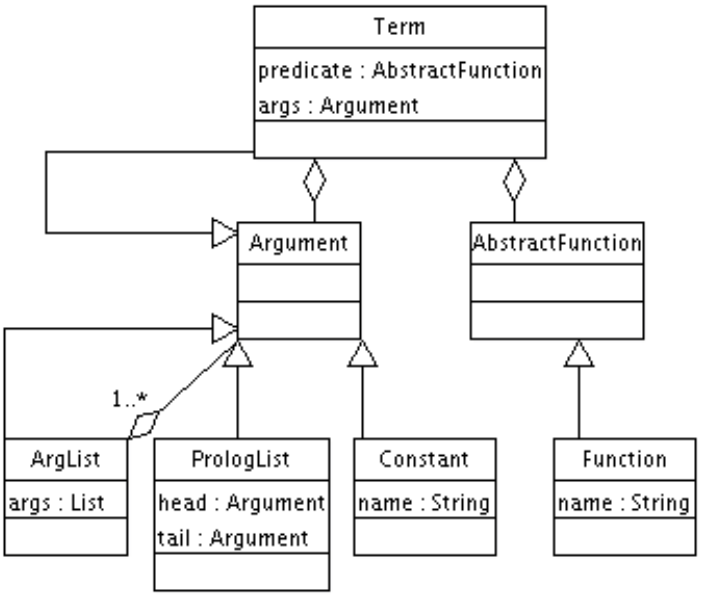

Figure 5.4: The composition of term

are basically defining a function whose name has not been specified yet.

There are still some classes appearing in 5.1, which have not been discussed yet. Two of them are the ”ConditionalMsg” and ”ConditionalAgent” classes. As you can see from the class diagram they inherit from the ”Message” and ”Agent” class and they represent conditional messages and conditional role transitions respectively. For the ”Message” case, the existence and form of the conditional message is obvious. For the ”Agent” case, it was the study of existing protocols which revealed the need of having conditional transitions to agent roles. Both classes are implemented by inheriting the functionality of the relevant parent class and adding a ”Condition” object as a member of the class. Finally, the last two classes which have not been mentioned yet, are the ”Clause” and ”Protocol” classes. The role and definition of both classes is easy to guess from the syntax of the language. The ”Clause” class consists of the head of the clause, which is basically the agent role being defined, and the definition for the role. The ”Protocol” class is nothing more than a set ”Clause” objects, defining the agent roles involved in the protocol.

An interesting observation about several classes, is that some members of them are defined as abstract types (the classes ”Definition”,”Constant”,”Argument” and ”AbstractFunction” are refered as abstract since can be specialized as other types), while specializations of them seem to be more suitable according to the syntax of LCC. An illustrative example is the ”ID” member of the ”Agent” class. According to the syntax

of the language, as it is presented in 2.2, the identifier of an *agent* should be a *constant*, which more or less corresponds to the ”Constant” class. However, we can see that the ”ID” member of the agent class is declared to be an ”Argument”. The reason why the abstract type is chosen, is to enable these fields to be filled with objects denoting unspecified statements. In our example, the result is that the ”ID” can be filled either with a ”Constant” object defining the identifier of the agent or with an ”Argument” object, which can be later substituted with a constant. If ”ID” was defined as a ”Constant” object, no way to define an unspecified agent identifier would exist. The class structure does not restrict the use of other types deriving from ”Argument” and this is less than desirable.

The reason why we did not choose to refine the structure in order to apply such constraints, is to keep it as simple as possible. Increasing the complexity of the structure at this point will make the possible changes, which may arise from the process of experimenting with patterns, much harder. Keeping a flexible structure, which can support the full range of statements that may come up when building patterns, is expedient. Moreover, this kind of constraint can be applied by the interface of the editor at any time and with reasonable effort. Note that this is the strategy we followed in several decisions throughout our work. As it will extensively be discussed in 6, there is a lot of interesting future work to be done on the editor and therefore we wish to keep the design as flexible as possible during the process of revealing the role and use of patterns in LCC.

5.2.2 Implementation of basic operations

After discussing the structure of the class hierarchy and the reasons which lead us to it, the next step is to describe how this structure of classes implements the basic operations required for the editor. In order to do that, we first have to identify the required set of operations. After experimenting with several different primitive operations for the classes, the final set of them, which is adequate to support both protocol and pattern building, consists of the following:

- Printing statements and the protocol in whole
- Replacing statements with others
- Retrieving sets of statements of the same type, included in the protocol

Although the list above is quite short, it is adequate for supporting the full set of operations provided by the interface. In particular, if any class included in the diagram shown in 5.1 can successfully return its textual representation, check its members and replace them with a given object if necessary and return the list of statements of a given type which are included in it or its members, we can then built an editor which provides the functionality shown in the example scenario of 4.3.2. The way, in which these primitive operations are combined to provide the visible result, will be discussed later in this chapter. First, we would like to take a closer look on how these operations are implemented by the class hierarchy.

An approach which has been extensively used for providing the required functionality, is the use of polymorphism. This is expected since the whole idea of the *composite design pattern* is based on polymorphism. The case of printing the protocol on the screen is illustrative. Every class representing a statement, has a member function called `toString()`. The leaf objects, such as ”Constant” objects, implement the method to return their simple textual representation (e.g. ”Constant” returns just the name of the constant or variable). Composite objects should return their textual representation by getting asking from their member objects to return theirs and combine them appropriately (e.g. an Agent for example should return something like `"a("+role.toString()+","+ID.toString()+")"` in Java syntax). Polymorphism ensures that we can call the `toString()` function of a ”Definition” (or any other abstract type) object and the get the correct result since the relevant code according to the actual type (”Message”,”Agent”,”Sequence”,”Choice” or ”Parallel”) will be executed.

The same approach is more or less used for the other two basic operations as well. For the replacement case, each class checks if the statement to be replaced belongs to it and then calls the same method of its members. The result is that all the objects (and therefore the statements) in the protocol are recursively checked and all occurrences of the given statement are replaced. In order to achieve that, for each statement type (”Definition”, ”Condition”, ”AbstractFunction” or ”Argument”) all classes implement a method for replacing statements of this type (e.g.

```
void replaceDefinition(Definition oldDef,Definition newDef)
```

for definitions). The case of retrieving all the statements of a given type is similar. All classes implement a method for each type of statements that we want to retrieve, taking as an argument, a list of objects of this type (e.g.

```
void getDefinitions(List<Definition> defs)
```

for definitions). This method checks if this object (the one which owns the method being executed at a given time) is of the relevant type (e.g. for the `getDefinitions` method, it checks if it is a definition itself) and if so, adds itself to the list. The ”Protocol” object provides a method for returning a list of the statements to the client. It creates an empty list and then asks from all its ”Clause” objects to fill that list. The call recursively reaches every statement in the protocol, in the way we described before. The result after traversing all the statements is that the list contains all the objects of the given type. These three basic operations is essentially all that is needed by the interface in order to accomplish its objectives. The exact way in which they are combined will shortly become clearer.

5.3 A Graphical User Interface for protocol building

5.3.1 Description of the graphical interface

A major part of the implemented system is the graphical interface of the editor. The role of the interface is important for the goals of the project and it is therefore interesting to briefly discuss its design. As we have stated before, the interface was implemented using SWING, which is native Java library for building GUIs. We do not wish to get into the details of the implementation at the code level, as nothing really innovative is involved there. On the other hand, its design was a big issue during the development of the editor, as we had to ensure the efficiency of the protocol building process. The following discussion will focus on the design of the most important forms, since some of the forms are rather intuitive in terms of design and functionality.

The central part of the interface is the main display shown in figure 5.5. The menu of the window consists of three groups of items:

- the *File* submenu, which provides access to the functionality for starting a new protocol, opening an existing one and saving the current protocol
- the *Protocol* submenu, allowing the addition of new clauses, the replacement for all kinds of statements, the reordering of argument lists, the validation of the protocol, the deletion of the selected clause and the replacement of a definition with an unspecified one and
- the *Patterns* submenu, which allows the user to save a set of clauses as a pattern and apply a saved pattern either as a skeletal or as a refinement of an existing

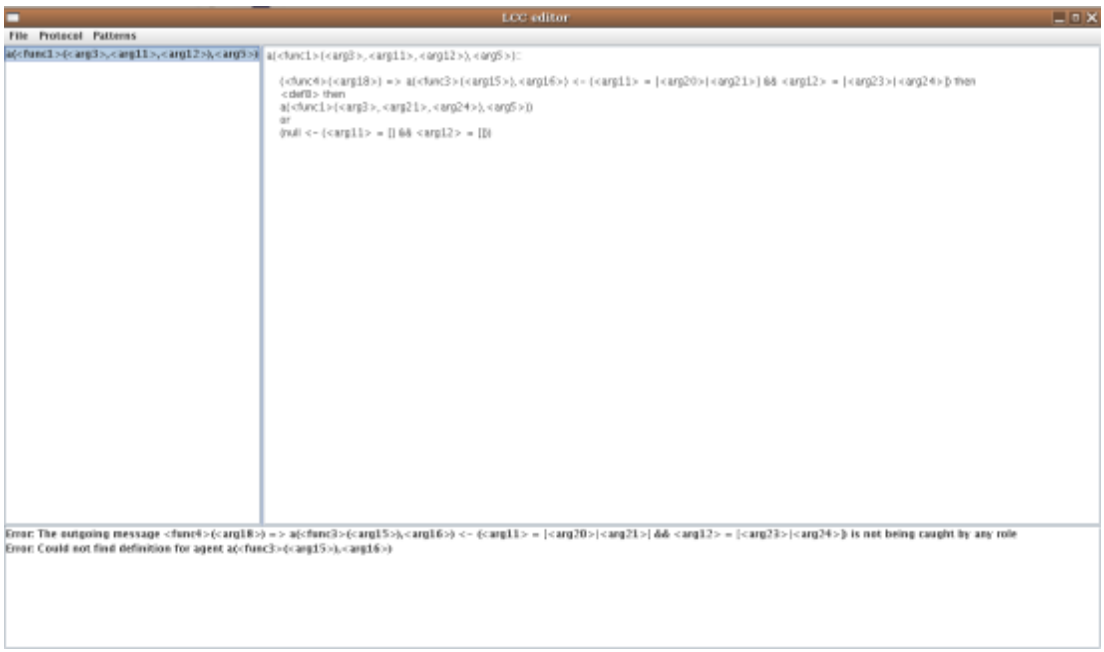

Figure 5.5: The main window

clause

In most of the above cases other forms are called to provide the actual functionality. In fact, the only part of functionality implemented directly on the main window is the one involved in the *File* submenu. Apart from the main menu of the editor, the main window is used for the visual presentation of the protocol. On the upper left side of the window there is a list of the agent roles defined so far. When clicking on an item of the list, the definition for the selected role is presented on the upper right side. Finally, the lower part of the screen is used to show the error messages produced by the validation process.

The next display we are going to discuss is the one for replacing unspecified definitions with other definition statements. Its layout is shown in figure 5.6. The user can select the definition to be replaced from the drop down list on the top of the window and the type of the replacement by selecting one of the available types. If the selection is a composite definition (sequence, choice or parallel), the definition is replaced by an object of the selected type. If the definition is to be replaced by a message or agent, then other forms allowing the user to build them will be called. The result returned from the relevant window will then replace the unspecified definition. The window for building a new agent definition is shown in figure 5.7. The check box on the top of the window allows the user to select if the transition to defined role is conditional. The

Figure 5.6: The window for specifying definitions

Figure 5.7: The window for building agent definitions

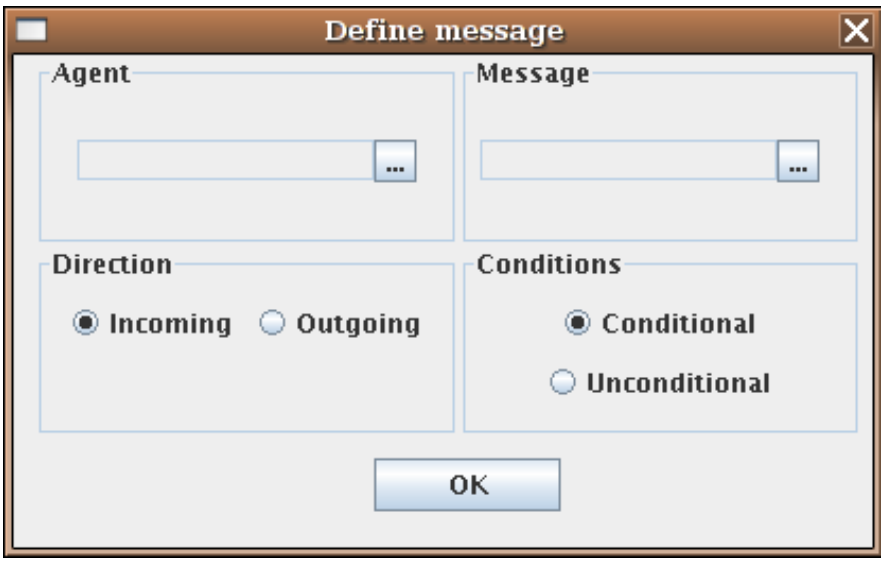

Figure 5.8: The window for building message definitions

panel below that is for specifying the role of the agent. It can either be unspecified (a ”Term” object with unspecified function and unspecified argument) or the window for specifying a term can be called by pressing the button beside the text box showing the term. In the second case, the ”Term” object returned by the window is shown in the text box. On the bottom of the display there are three options for specifying the ID of the agent. The ID can be either an already defined constant, a new unspecified argument or a new constant. In the later case, the window for building new arguments will appear in order to build and return a new object. Note that the agent window is also used for building a new clause. The only difference is that the returned ”Agent” object is used for the head of the clause. A new unspecified definition is used as the body of the clause.

The other window, which can be called from the definitions window, is the one used for building message definitions (figure 5.8). At the upper left part of the display the user is required to define the agent sending or receiving the message, by calling the agent window we have described just before. Beside that, the content of the message has to be specified by calling the window for building terms. At the lower part the direction of message the its specific type (conditional or not) can be chosen. By not specifying the agent and the content of an outgoing conditional message, the user can build a *null* message statement.

There are two forms which have already been mentioned and would be worthwhile to discuss. The first is the window for building terms and is shown in figure 5.9. The

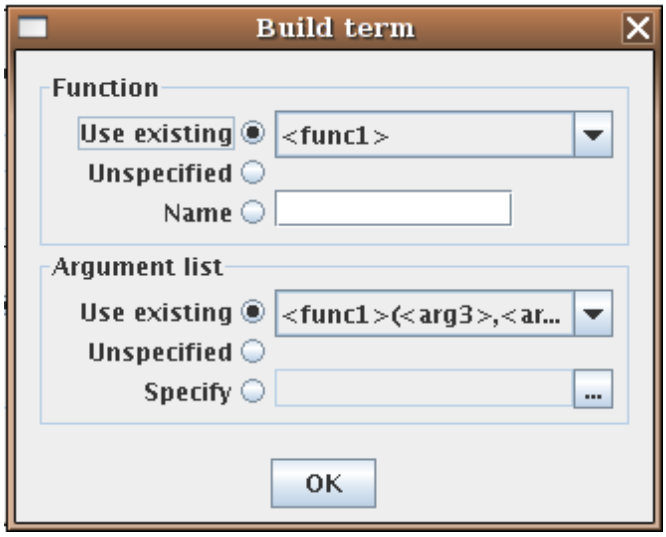

Figure 5.9: The window for building terms

Figure 5.10: The window for building arguments

general approach to the design of the display is similar to the one described in the agent window and is generally adopted for designing the forms of the editor. On the top of the display the user can specify the function of the term as an existing, a new unspecified or a new named function. Below that part, the argument can be specified in a similar way. The only difference is that the argument window is used in order to define a specific argument.

The second window is one for building arguments. Its layout is shown in figure 5.10. The argument can be specified as a constant, a list (in the Prolog sense), a term or a sequence of arguments. In the constant case only the name of the constant has to be provided. For the other three options the relevant window for building each statement has to be called. Apart from the term window, which has already been discussed, the two other forms are shown in figures 5.11 and 5.12. As far the list

Figure 5.11: The window for building arguments

Figure 5.12: The window for building arguments

window is concerned, the layout can easily be guessed considering what has already been discussed. The head and the tail of the list can be specified in a similar way. They can be either an existing argument, a new unspecified one or a new constant. The argument list window is somehow different from the others. At the rightmost part, there is a list presenting all the arguments which have already been added to the list. The argument specified on the left part of the display can be added to the list by pressing the ”Add” button. The options for the argument are the same as those in the term window.

The next window we are going to discuss is the one for replacing abstract conditions (figure 5.13). Its design is similar to the one for replacing definitions. The conjunction and disjunction options are handled by the window itself since no additional information is required (replaces the selected condition with a conjunction or

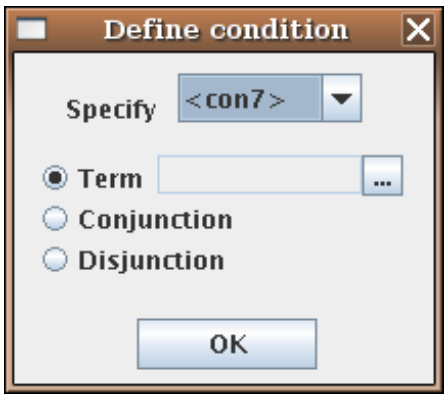

Figure 5.13: The window for replacing unspecified conditions

disjunction of unspecified conditions). For the case of specifying a condition the window shown in figure 5.14 is called. A condition statement can either be in the form of a predicate (e.g. $available(T)$) or in the form of an equality (e.g. $S = [H|T]$). This can be determined by choosing the appropriate option on the top of the window. Right below that, the user can specify the term in the case of a predicate condition. We have already discussed the window which is to used to build terms 5.9. At the lower part of the display the both sides of the equality can be specified. The available options are the usual apart from the right hand side, which can be a newly defined list. The list can be specified with the relevant window shown in 5.11. Note that according to the initial type selection, only the corresponding part of the display can be used.

Considering the design approach of the forms presented so far, it is rather easy to guess how the forms used for replacing unspecified functions and arguments look like. The first one allows the user to select the unspecified function and specify it as an existing one of as a new by specifying its name. In the arguments case, the selected argument can be specified either as an existing argument or in a way identical to the case of building arguments (figure 5.10). The selection of the unspecified statement can be done as in the definition window in both cases.

It is not worth discussing the rest of the forms called from the *Protocol* submenu of the main window in detail, since their design and functionality are rather intuitive. We would just like to mention that there is a window for reordering the arguments of an argument list and one which allows the user to select a definition in order to replace it with an unspecified one. The first one is useful as the application of patterns does not ensure that the argument lists of two occurrences of the same role will be in the same ordering. The validation process checks the number of arguments but the

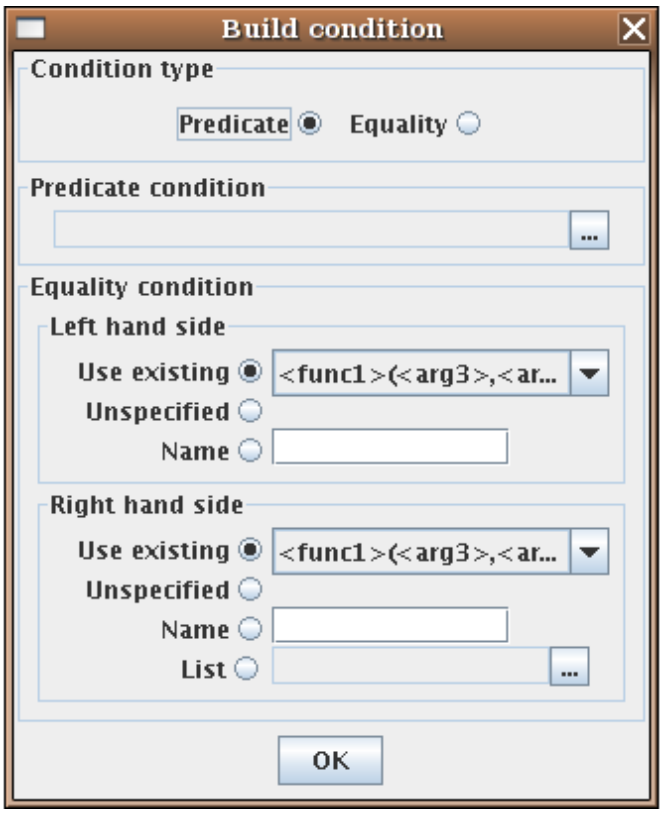

Figure 5.14: The window for replacing unspecified conditions

ordering is not checked anywhere. It is up to the engineer to ensure the consistency of the argument list as far as the ordering is concerned. The second window is a tool for correcting errors. The user can select a faulty definition and replace it with an unspecified one, which can then be defined again correctly. Finally, the validation feature and the functionality for deleting the selected clause do not require separate forms and are implemented directly on the main window. In the first case, the relevant method of the protocol object is called and the result (a list of errors) is shown on the bottom of the main display. The deletion of a clause is done by deleting the highlighted clause in the list on the upper left part of the main window.

The last part of the graphical interface is concerned with the functionality supported by the the *Patterns* submenu of the main window. The saving of a set of clauses as a pattern and the application of a pattern are not really interesting to discuss from the interface point of view. In the first case, a simple window to select the clauses to be included into the pattern is initially used. The next step is to select the file, in which the pattern is stored, using a common file chooser. For the second case, the file is initially selected and then the pattern is applied to the protocol after being shown in

Figure 5.15: The window for mapping the argument list of the pattern to the one of the existing clause

a preview window. The case which interesting to discuss is the use of a pattern to refine a skeleton. The first two steps of this process are similar to the ones of applying skeletons. The only difference is that in this case the user is required the clause to which the pattern will be applied. After the confirmation in the preview window, the display shown in figure 5.15 is presented to the user. The role of this window is to provide a mapping between the head of the existing clause and the one of the pattern clause. In particular, the argument lists of the two agent roles must be merged in order to provide the argument list of the resulting role. The drop down list at the top of the display includes all the arguments of the role in the pattern. The selected argument from this list can be mapped either to an argument from the argument list of the existing clause or to a new argument, which will be added to the list. Note that we do not require that all elements of the argument list are mapped. This choice allows the partial application of a pattern and it can be useful when trying to apply a pattern which is not perfectly suitable to the specific case.

After confirming the choices in the window shown in 5.15, another window is required to complete the process of applying the pattern (figure 5.16). This one provides the functionality for replacing unspecified definitions from the existing clause with definitions from the pattern. The left part of the display allows the selection of a definition from the pattern. The drop down list includes all the ”Definition” objects of the pattern. The preview below that shows the full textual representation of the selected definition (there is not enough space at the list). Similarly, the right part is used to

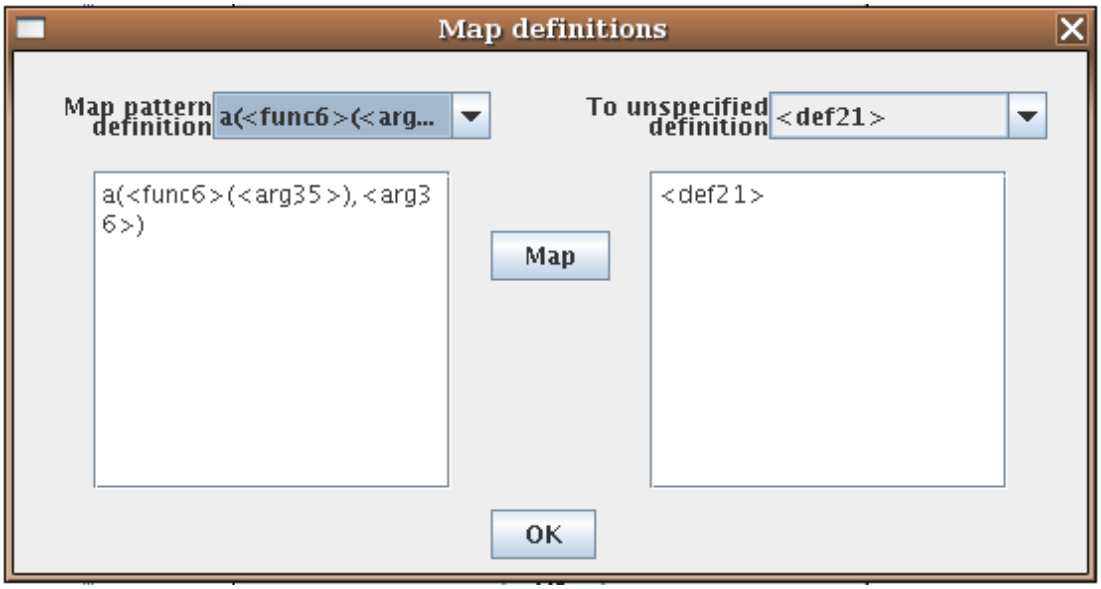

Figure 5.16: The window for replacing unspecified definitions from the existing clause with definitions from the pattern

select the definition to be replaced. As in the previous window, it is not required to use all the definitions from the pattern, for the reasons described above.

After describing the most interesting of the forms, we discuss some general issues about the interface in whole. The approach followed throughout the design of the GUI is to create a wizard-like interface. The way in which forms are called from others in order to perform specific tasks, results in a sequence of steps which gradually achieves the initial objective. The decision to follow such an approach was made for two reasons. Firstly, we believe that the division of a task into steps is particularly suitable to the incremental method for building LCC protocols. It quite natural to think the method as a series of actions which incrementally define parts of the protocol and the interface should not violate this natural way of thinking. The second reason is the familiarity of users with wizard-like interfaces. Although the editor does not target novice users, learnability is considered as a factor which plays some role for the value of the outcome. One of the objectives of the graphical editor is to allow the building of LCC protocols at a higher level than simply writing the protocol in LCC. In that sense, it can be said that it aims to assist engineers to get familiar with LCC more quickly. This can be achieved by a graphical interface which requires less time to learn than the process of building protocols by hand. We expect that an interface which is structured in way familiar to most users, will perform better in that sense.

5.3.2 Using class structure to provide the editing operations

Another interesting implementation issue is the integration of the internal structure with the interface. We describe this by example. The correspondence between the editing operations and internal procedures can easily generalize to other cases. The definition of a message provides a good scenario, as it involves more operations than other statements. We wish to use the scenario described in 4.3.2 for this purpose. A suitable case is the replacement of <def8> with an unconditional incoming message. The protocol at this stage is shown in figure A.22. We can see why <def8> should be replaced by a message, by comparing the current state of the protocol with the target clause shown in 4.1. In the next few paragraphs, we discuss how interface operations lead to changes in the internal representation of the protocol when trying to build the required message definition.

The first step is to use the menu of the main window, in order to show the display in figure A.24. The drop down list with the definitions is filled by asking the protocol to return all of its unspecified definitions, using the mechanism described in 5.2.2. This is the way in which all the drop down lists are filled with the suitable statements. After choosing the <def8> as the definition to be replaced and the message as the type of the new definition, the ”OK” button should be pressed to confirm the choices. As we said when describing the display, the window used for building messages will then appear (figure 5.8).

At this point we have to define the ”Agent” object. By pressing the relevant button, the window for specifying agents will appear (figure 5.7). The role of the agent should be the same as the one in the outgoing message of the protocol (according to the target clause in 4.1 the agent receiving the outgoing message is the one sending the incoming). In this case we have to define an appropriate ”Term” object by calling the relevant window (figure A.25). We can specify that the function is the existing <func3> and the argument is <arg15> (same as the role of the agent receiving the outgoing message). When the ”OK” button is pressed, the window returns a new ”Term” object whose function member is set to be the object corresponding to <func2> and its argument member is the object corresponding to <arg15>. Note that we are not cloning the objects. The same objects are referenced by both roles.

After the term window is closed, the agent window displays the textual representation of the returned ”Term” object in the relevant text box (figure A.26). The ID of the agent can be specified using the drop down list since it is also the same as the one in

the outgoing message. A new ”Agent” object will be created using the ”Term” object returned before and the selected existing ”Argument” object as members. The final result after the agent window is closed is shown in the relevant text box of the message window.

The second part of defining the message is to specify its content. The content of a message is defined as a term and therefore the term window is shown (figure A.27). The function of the term we want to build is ”inform”, but we can also use an unspecified one at this point, and its argument is another term. This means that we have to specify the argument list of the term by pressing the relevant button. In the display for building arguments (figure A.28) appearing next, we specify that the argument will be a term. Another instance of the term display, allowing us to define the new term, is shown (figure A.29). This time, the function should be named ”located” and the argument is a list containing `<arg20>` and `<arg23>`. In order to specify its argument, the arguments window is shown again (figure A.30. This time the choice is an argument list and the window shown in figure A.31 appears.

After adding the two existing arguments to the list (`<arg20>` and `<arg23>`), the window returns an ”ArgList” object containing them back to the arguments window. After confirming the choice in the argument display the same object is returned back to term window and a new ”Term” object is created (this is the term `located(<arg20>, <arg23>)`). The new ”Term” object is returned to the first instance of the arguments display and after pressing the ”OK” button it is returned back to the first instance of the term display. At this point a new ”Term” is created using the returned ”Term” object as its argument (`<func2>(located(<arg20>,<arg23>))` assuming we have used an unspecified function instead of ”inform”). Finally, this last ”Term” object is returned to the message window.

After specifying the agent and the content of the message, we set the message to be incoming and unconditional (figure A.32). This is possible using the relevant options on the message display. The confirmation of the message window will return a ”Message” object to the definitions window. This object contains the ”Agent” and ”Term” objects created in message window as members (the ”Term” object is the content). The final step is to confirm the definitions window. The window will call the protocol’s method for replacing definitions, passing to it the definition `<def8>` and the newly constructed message as arguments of the call. The protocol will ask from its clauses to replace the given definition (`<def8>`) and the process of replacement described in 5.2.2 will result in the state shown in A.33. Finally, the definitions window

will ask the main window to refresh the displayed clauses. Although the procedure described above does not involve the interface in whole, it is adequate to demostrate how basic operations are used from the graphical interface in general.

Chapter 6

Evaluation and discussion

In the previous two chapters we have described a method for building LCC protocols and a graphical tool implementing the method. The task of evaluating them is not an easy one, for several different reasons. The nature of the language, in which the method is based, is one of them. We have already mentioned that LCC is not just a logic programming language, but mainly a process calculus. Although several efforts with similar objectives have been made for techniques editing in Prolog, it is clear that a direct comparison with the LCC case is probably not meaningful. There are specific aspects of the different methods, which can be used as a basis for a comparison, but it is probably not possible to say that our editor is clearly better or worse. The difference in the groups of people that the editors are supposed to help (we focus on assisting engineers while the other two editors focus on teaching novices) is also expected make the comparison even more difficult.

Another issue about evaluation is the difficulty of measuring any aspect of the performance. A useful approach to estimate the efficiency would be to experiment on the usability features of the editor with human subjects. An experiment like that could measure the time needed for constructing protocols with and without the editor. The number of errors in the constructed protocols would also be an interesting metric. A reasonable hypothesis is that the editor, and therefore the method, can help engineers to build LCC protocols faster, while making fewer mistakes. Even an experiment like that would not be perfectly appropriate for evaluating our work at this stage. Although the editor has some value from the engineering point of view as it is, its practical use for protocol building is subject to many improvements. At this point, the graphical tool is mainly for demonstration purposes in order to investigate the use of patterns in LCC. Moreover, the described experiment is not feasible at this point, due to the lack

of time and the difficulty of finding adequate number of knowledge engineers familiar with LCC. Given these facts, there is no obvious way to obtain some metrics about the efficiency of the method and this is a major problem for the evaluation.

Considering these difficulties, there are not many alternatives left for the evaluation of the project. The approach we have followed is to theoretically discuss how and in what degree the expectations are met by the outcome. A comparison with the editors listed in 3 will also contribute to the evaluation task. Note that this comparison will be limited to the factors in which it can be meaningful, given the differences of the editors. Finally, a list of improvements for the method and the editor will be presented. These improvements will help the method to achieve its goals at a higher degree, while enabling us to evaluate our work in measurable terms.

6.1 Comparing outcome with project objectives

In this first part of the evaluation we will try to assess the project outcome with regard to its initial goals. In order to do that, we will have to clearly identify and state the objectives and the factors affecting them. As it has been said before, the two main objectives of our work are:

1. The investigation of how patterns can be used in LCC protocols and whether *techniques editing* can be suitable for LCC

2. The development of a tool which is expected to make the LCC approach for the coordination of multi-agent systems, more accessible to knowledge engineers

These two objectives are illustrative of our motivation for this project. However, they are too general to be used for evaluation. The main factors involved in satisfying them have to be determined, in order to have meaningful discussion about how the outcome meets these demands.

From the researchers' point of view, an ideal method achieving the first objective would allow us to define and combine any kind of patterns at any level of abstraction. Although it is not clear how this kind of freedom would affect usability (it is not sure if such a method would be efficient for the users), it is interesting for the researchers to be able to capture all kinds of common programming techniques. Such a method should also allow the building of protocols by using only patterns and without requiring manual refinement. Another interesting feature is to be able to determine the suitability of a pattern for a given case. The building of protocols could then be done by entering

high level goals to the editor, which would be able to propose combinations of patterns achieving the goals. Given that our understanding of patterns in LCC is not complete at the moment, a reasonable expectation is probably to approach the most important of the above features.

The range of protocols supported by our editor are adequate to include the most commonly used patterns, which are the skeletal ones due to their higher abstraction. By studying the example protocols in [9], [10] and [8], we can see that suitable skeletons for all the clauses involved can be defined and applied using the editor in a convenient way. Moreover, as we have shown in the scenario in 4.3.2, a number or detailed patterns can be supported and a mechanism for applying them to skeletons exists. However, interesting improvements can be made in order to increase the information and the constraints included in the detailed patterns (e.g. we can require that an undefined definition includes specific statements). This will also make the range of such patterns wider. The existing structure allows these modifications (e.g. in order to support the constraint described above only an appropriate extension to the definition class is required) and therefore it is argued that we have worked towards that direction.

Finally, the mechanism for combining patterns can accept even more improvements. In particular, a more detailed mapping between the statements in the pattern and the ones in the already constructed skeleton. An initial approach to this is illustrated by the way that the heads of two clauses are combined when applying a detailed pattern. Hence, we believe that this issue has also been studied during the project and there is a straightforward way to extend the functionality in this way. An aspect in which we did not focus during this project, is the feature of judging the suitability of patterns and proposing relevant patterns based on that. The automated combination of patterns, in order to accomplish high level goals, was also not a feature we have worked on. These two are considered as advanced features requiring in depth research on LCC protocols and considerable amount of time to implement. Our editor can only be used to experiment with patterns, in order to obtain the information required for building such a system. Although these features are extremely interesting from the researcher's point of view, we believe that a tool for building LCC protocols can be adequately efficient without supporting them. The performance of the editor with regard to patterns in general is considered to be good, given the available resources for the current project. Many of the missing features can be easily added and are better described as future work than as shortcomings. The development of a more intelligent editor requires much more work on understanding how patterns work in LCC and

therefore it would not be reasonable to expect that so early.

Apart from understanding and applying protocols in LCC, which is interesting from the researcher's point of view, another objective of the project is to build a tool for supporting knowledge engineers. This second goal has mostly practical value and should be considered together with the first. We could probably put more effort working on patterns, if we did not have to ensure that the outcome of the project is useful from the practical point of view. The usefulness of the tool in that sense can be measured as the time needed to get familiar with the process of building protocols, the time needed to construct a given protocol and the quality of the protocols in terms of the number of errors in the final result. As we have argued before, we do not have the resources to run experiments with human subjects on the above factors. What we are going to do is to evaluate the usability and efficiency features of our editor by comparing them with the desired ones.

A tool able to construct LCC protocols using high level descriptions as input, would probably be the most efficient and convenient way to build protocols. Moreover, the ideal editor should be able to build the full range of protocols supported by the language, since it does not aim to be a demonstration or teaching tool (in that case a reasonable subset would be adequate), but a practical one used for general purpose protocol building. Especially this last requirement can not be ignored by any tool claiming to have practical applications. It is obvious that this combination of requirements is not just difficult to implement but probably even infeasible. We believe that the most decent way to build a useful editor is to prioritize the different requirements involved.

As said before, the most important requirement is to support everything that can be written using the LCC language. This was the basis for the development of our editor. The incremental method can build any possible statement of LCC and it can be said to be equivalent with writing protocols by hand in terms of expressiveness. This feature ensures the suitability of the outcome for use in general purpose protocol building. The second factor is the level of description required. Apart from the knowledge about what the protocol should do, writing protocols also requires detailed knowledge of the syntax of LCC. Our editor lifts the level of knowledge required by the engineer to a higher one. The user of our editor is able to focus on the semantic level of the protocol by determining which the content of a statement should be. We do not claim that no knowledge of the syntax is required, but the editor itself can avoid many potential errors in protocol structuring. Moreover, errors at this level are not expected to occur since

the statements are guaranteed to be syntactically correct. The semantic consistency of the protocol is also partially supported, mainly by the validation feature we have described before. However, more work is required in order to claim that the tool is adequate to ensure the semantic consistency of the protocols.

We have argued that the editor is valuable in the sense that it requires less knowledge from the engineer than manual building. This has mainly to do with allowing engineers to become familiar with LCC more quickly. The existence of a graphical user interface can also help in that sense. The issue of efficiency either in terms of time needed for constructing protocols, or in terms of quality of the outcome, is more difficult to evaluate without having metrics. We can argue that it is probably the case that the quality of the protocols is considerably better, given the lack of syntactical errors and the mechanism for avoiding common errors at the semantic level. However, the claim is not well supported without having discrete measurements. The speed up in the process of building protocols is even harder, since any estimation would be very inaccurate and therefore of limited importance. Roughly speaking, even if the incremental method does not contribute at the reduction of time at all, the use of patterns (especially the skeletal ones which can be widely used with very low effort) is expected to speed up the process. In general, the editor is expected to be more efficient than writing the protocol by hand, but it is left to be determined when more evidence will be available, probably by running some experiments.

The conclusion of this part of the evaluation is that the work done so far is towards the right direction. Although there is a difference between the state of our work and the ideal case, which varies from issue to issue, we can argue that the approach to the problem was at least a fair one. We are going to see later that interesting future work can be based upon our effort and this fact is adding more value to the current outcome.

6.2 Comparing outcome with related work

We have already discussed in 3 two *techniques editing* based Prolog editors. We have also argued that these efforts are similar to ours due to the approach they follow for structured development of Prolog programs. However, we have also highlighted the differences between LCC and Prolog. It is not meaningful to consider all three editors as alternative solutions to the same problem. We will try to compare the approaches with regard to the aspects in which a comparison may lead to interesting conclusions.

An interesting issue is the representation and use of patterns. Robertson's editor

([6]) has a richer representation of patterns comparing to ours. It follows the same skeleton-addition approach, but the patterns also include information about how arguments will be mapped to existing clauses. Moreover, the editor can assess the suitability of a specific pattern in a given case. The whole procedure of applying patterns is much more controlled than in our case. Ted ([1]) on the other hand, has some similar features for mapping the arguments and checking the suitability, but the range of patterns it can support is limited to relationships between the head and the recursive arguments of recursive clauses. Given the fact that both of them aim to help novices learn Prolog, it is quite obvious why this controlled approach to applying patterns is desirable. We do not argue that such features would not be interesting in our case, but it is probably the case that they are not of the same importance. In our case, it is reasonable to require that engineers know how to use the supplied patterns in order to accomplish their objectives.

By taking a closer look to the way that the three editors view patterns, we realize that there is a major difference between our editor and the others. We mostly view patterns as reusable parts of LCC code, which can be applied when constructing protocols mostly in order to save effort. No information about what these patterns do at the semantic level is available. Prolog editors view patterns as primitive operations, the combinations of which can produce an impressive range of Prolog programs. They aim to help novices understand these primitive techniques and learn how to combine them to build working programs. This second approach would probably be desirable in LCC case as well. The reason we did not work towards that direction is mainly that LCC identical to Prolog. In Prolog quite impressive things can be done with a reasonably small number of patterns. This is the case mostly because Prolog programs are made by using the language's simple syntax in complex ways. Some of these ways can be captured and formalized as patterns.

Judging from what we have seen so far in the LCC protocols we have studied, LCC is somehow different. Although it is similar to Prolog in some sense, the patterns which can do amazing things in Prolog, are not that useful for building LCC clauses. This is not unexpected since LCC is a process specification language aiming to express totally different things. It is not clear yet if LCC clauses can be seen as combinations of a reasonably small set of patterns. Even if this is proved to be the case, these patterns will definitely not be same as in Prolog. Considering the practical value that we would like the outcome to have, we have chosen a approach which is more pattern-assisted then pattern-based. At this initial stage of research in LCC patterns, we do not wish

to see clauses as combinations of patterns. What we want is to provide mechanisms for capturing common parts of LCC code and reuse it when possible. By studying the repeatability of those patterns and the factors which may affect it negatively, we can improve our method and probably reach the level of pattern usage in Prolog.

Another interesting aspect, in which we can compare our editor with the others, is usability. Robertson's editor can suggest modifications to the program being constructed. It provides guidance to the user and it is clear that much effort has been put on the usability of the editor. Although the interface is not graphical, it is obvious that it is good enough to allow novices construct Prolog programs efficiently. Ted does not seem to provided such a guided approach to the construction of programs, but the level of usability is good due to the controlled way of applying patterns. In general, both editors seem to have better performance than ours from the usability point of view. Once more, this is expected since they both target to novices. In our case, it is probably more important to do things fast than to provide explanations and guidance. Of course, the combination of efficiency and usability would be even better, but much effort is probably required to reach such a result. We have focused on allowing engineers to express anything that can be expressed in LCC, using a robust method which aims to reduce the effort. Future work can be done on the graphical interface in order improve usability without violating its efficiency and therefore degrading its value from the engineering point of view.

The conclusion after this brief comparison is ambiguous. The differences in the languages supported by the editors and the goals of the projects do not allow us to adequately evaluate our work with regard to related work in Prolog. There are some aspects in which discussion is meaningful, but even in those cases there are arguments to support that these differences do not allow safe conclusions.

6.3 Future directions

We have mentioned several aspects of our editor so far, in which we believe that interesting improvements can be made. The research in pattern-based building of LCC protocols is at a very early stage and therefore the list of possible improvements to the method and the editor can be very long. We have discussed some of the long-term directions in the previous sections. In particular, the investigation of how LCC clauses can be entire ly described as combinations of common patterns and the building of protocols by providing higher level descriptions of what the protocol is expected to do, are

already said to be long-term goals. Such improvements require a better understanding of LCC patterns and should be attempted at later stages. At this point we will try to focus on a reasonably small set of improvements which are more or less straightforward to apply. A list of those is the following:

- A more sophisticated way to merge patterns by allowing the detailed mapping of the definitions included in the pattern to existing ones.
- Mechanisms allowing constraints to be placed on unspecified statements (e.g. requiring that an unspecified definition should include somewhere a given predicate or argument).
- Refinement of the class structure to precisely reflect the syntactical constraints of LCC
- Work on the usability and efficiency of the graphical interface. Adapting the interface to the natural way of thinking when building protocols.
- Using the tool to experiment on patterns of variable detail, in order to obtain better understanding of their role in LCC

We have already described why the mechanism for applying detailed patterns on a skeleton should be further refined. We have also seen what such a mechanism should look like, since the mapping of arguments included in the heads of the clauses illustrates an approach to tackle the problem. The extension of the constraints to allow their application on unspecified statements is also interesting. At this moment, the user has to specify the position of a statement in a definition or condition. This means that the definition or condition has to be defined at some level of abstraction. This is restrictive since all alternative cases have to be defined explicitly. By allowing constraints on unspecified statements it becomes possible to require that another statement should be included somewhere in the unspecified one, without specifying its exact position (this is useful when we actually do not care about the position). As far as the class structure is concerned, we have already mentioned that we have chosen not to refine it as much as we could, in order to keep it simple. As the understanding of our actual requirements for the method is becoming more elaborate, the class structure should be refined to precisely reflect the syntax of LCC.

Usability and efficiency are important factors, since the editor aims to have some practical value. A study of the way that the engineers think when building protocols

would have a major contribution. A user interface adapted to a more natural way of thinking might reduce the effort and time needed to build protocols, while increasing the learnability of the interface at the same time. Finally, the exploitation of the outcome to further investigate the role of patterns in LCC is a major part of the future work. The conclusions that will be drawn by experimenting with patterns using the editor, will lead to the further improvement of the method and the tool, towards the long-term goals we have described before.

Chapter 7

Conclusion

We have proposed a method for the structured building of LCC protocols. We have also described how this method can be used to build a prototypical editor for this purpose. We argue that this tool has both a practical value, since it can be used for general purpose protocol building, and research value, since it can serve as the basis for further investigation of LCC patterns. Interesting future work can be based upon the results presented here and therefore we consider that our effort has considerable contribution to the whole LCC approach.

We have developed a tool, which seems to improve the process of building LCC protocols, by requiring less knowledge and effort from the engineer. Moreover, there are some straightforward things to be done that are expected to improve this situation even more. The practical value was a primary consideration when designing the method and implementing the editor and this has resulted in a tool which is expected to make LCC protocol building both easier and faster. The validity of these expectations are left to be confirmed by experimenting on the usability and efficiency of the tool, since we did not have the resources to evaluate our method by running experiments during this project. Nevertheless, the current tool does allow large segments of protocol to be added rapidly, in skilled hands, it is likely to be faster than a generic editor.

Judging the outcome from the researchers' point of view, we can say that although some interesting issues have been addressed, the research field is quite new and therefore there are lots of things to be done yet. The approaches used in Prolog editors can give useful ideas for how a similar method for LCC should be. However, a direct application of those approaches in LCC is not straightforward, because of the differences between the languages. LCC clauses does not seem to be described in the way that

Prolog *techniques* attempt to describe Prolog clauses, mostly because LCC is oriented to specifying processes. An adaptation of *techniques editing* approach is probably possible, but we have chosen a method that is more straightforwardly based on structural patterns, in order to ensure the practical value of the outcome. Our proposed method is not so ambitious as *techniques editing*, since it does try to describe LCC clauses as they are composed by a set of *techniques*. Our aim was to enable capturing and reusing common patterns, when it is possible and useful according the engineer's judgment. Since our editor can be used to write patterns, it can be used in order to improve our understanding of the role of patterns in LCC, enabling the design of more ambitious approaches similar to *techniques editing*. Some interesting future directions, that aim to the development of more sophisticated methodologies for builing protocols using patterns, are presented in 6.3.

Appendix A

Using the interface to construct an example clause

The following figures show the process of constructing the example clause shown in 4.1 as a series of screenshots taken from the editor. It supplementary to the description of the process given in 4.3.2.

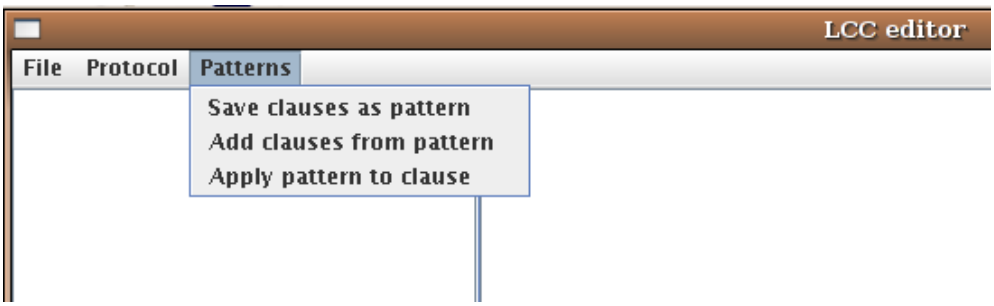

Figure A.1: Add a clause from pattern

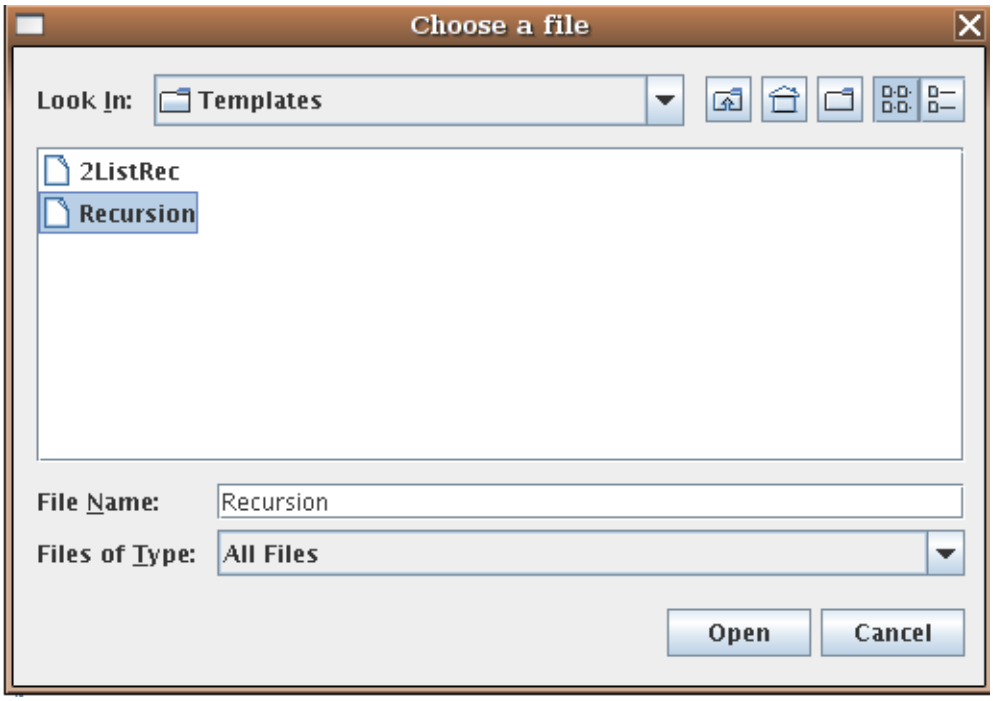

Figure A.2: Select the file

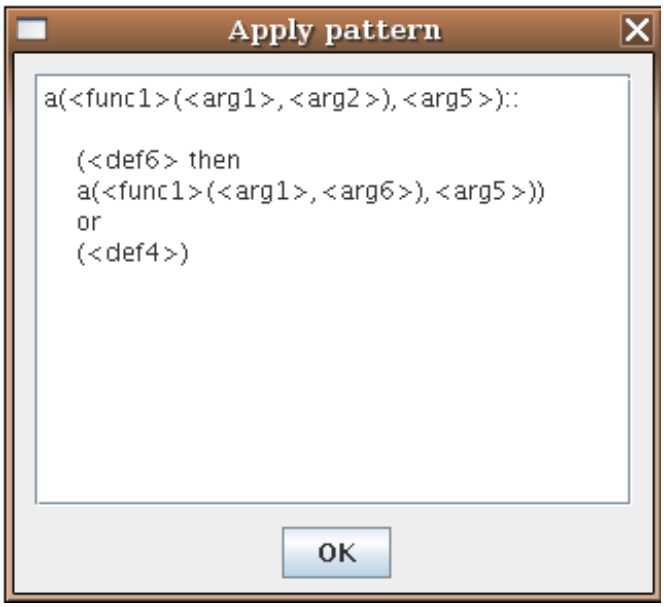

Figure A.3: Preview the pattern stored in the file

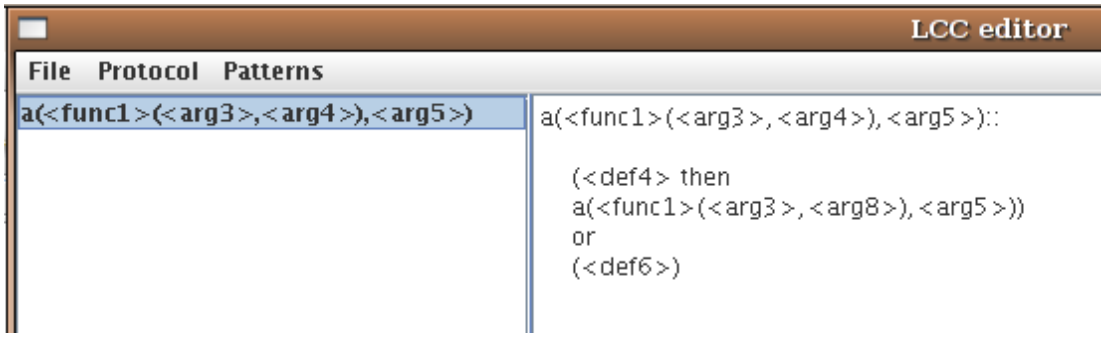

Figure A.4: The new clause

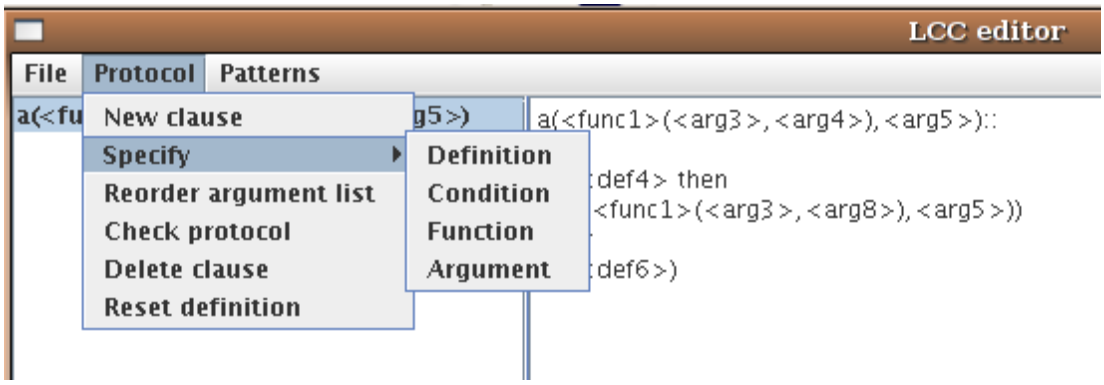

Figure A.5: Specify the definition

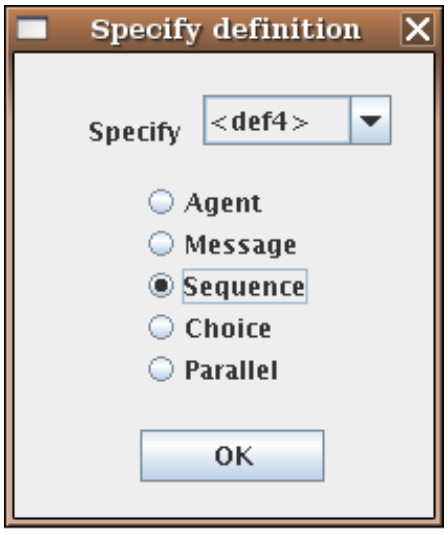

Figure A.6: Specify it as a sequence

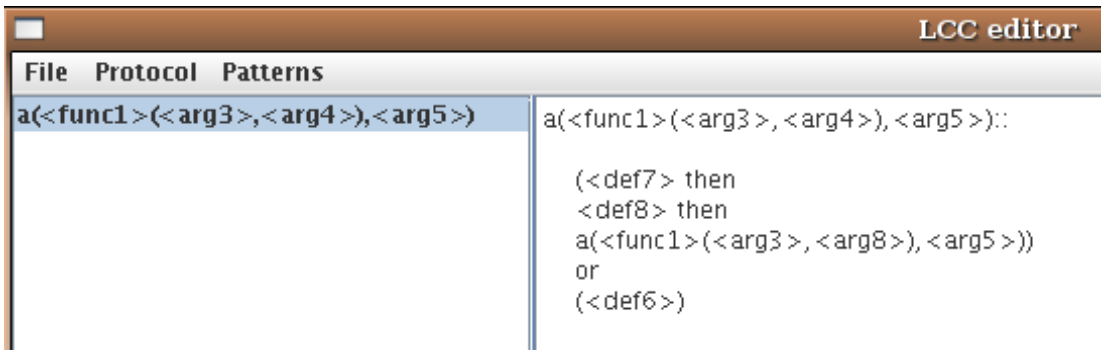

Figure A.7: The resulting clause

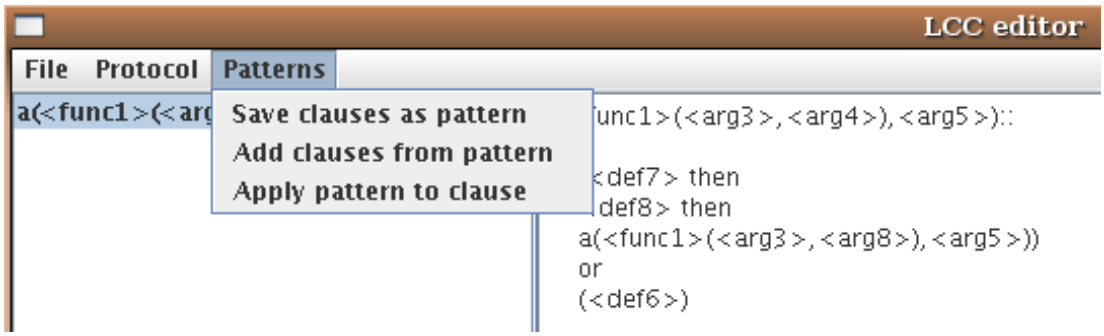

Figure A.8: Apply a pattern to the existing clause

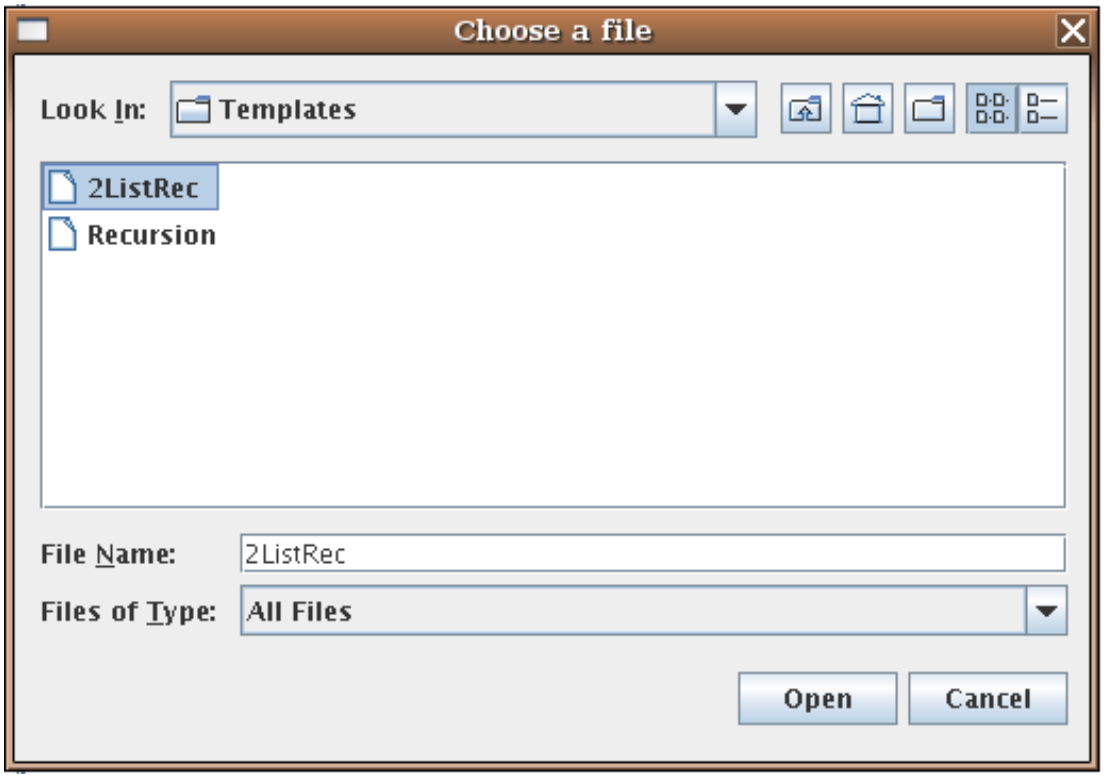

Figure A.9: Select the file

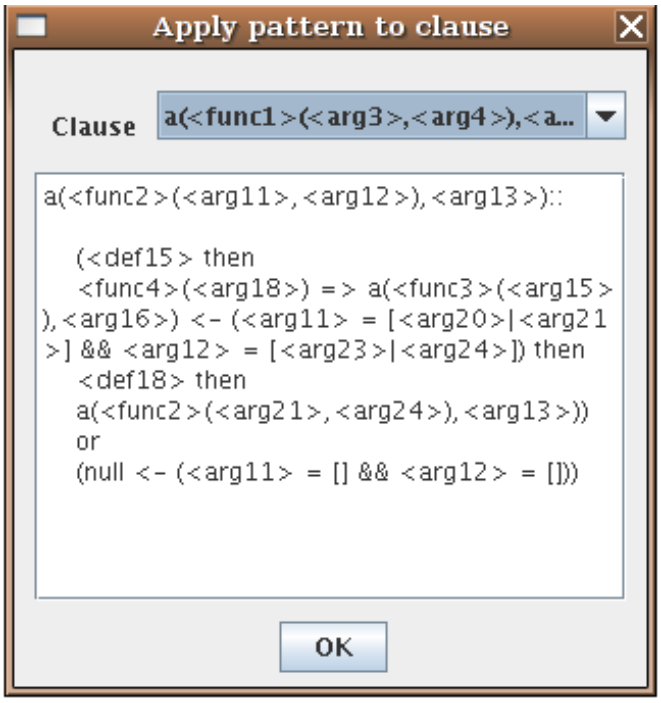

Figure A.10: Preview the pattern with the new numbering

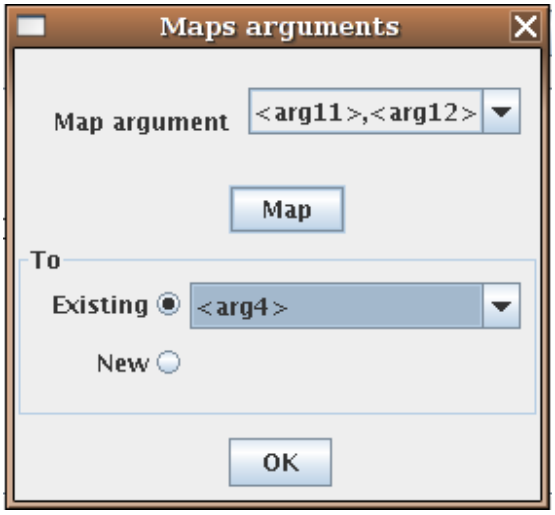

Figure A.11: Map arg11,arg12 to arg4 of the existing clause

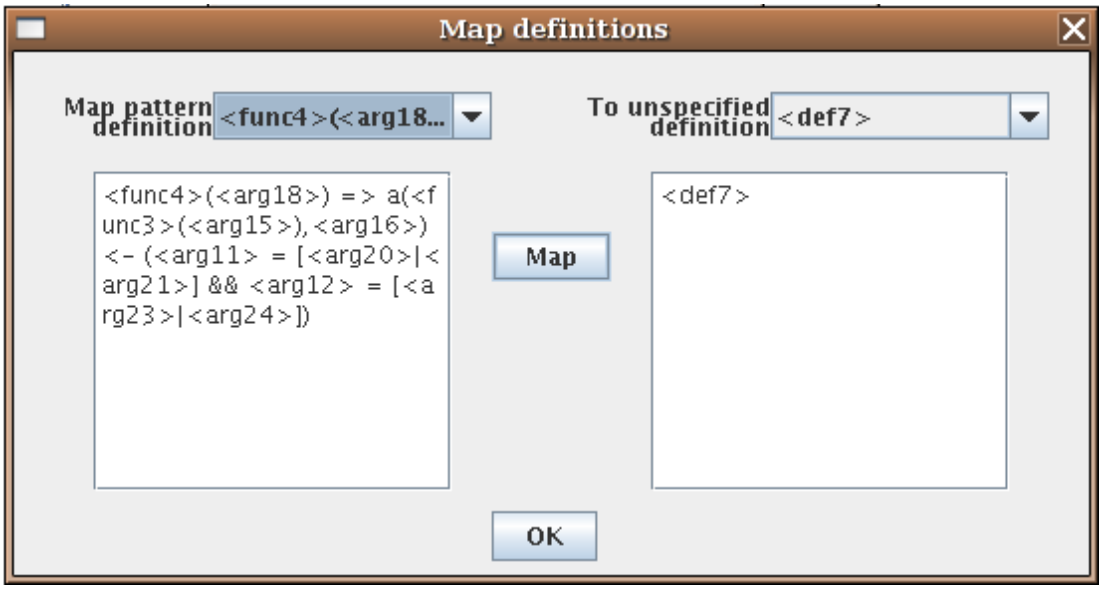

Figure A.12: Map the definition representing the conditional message to def7

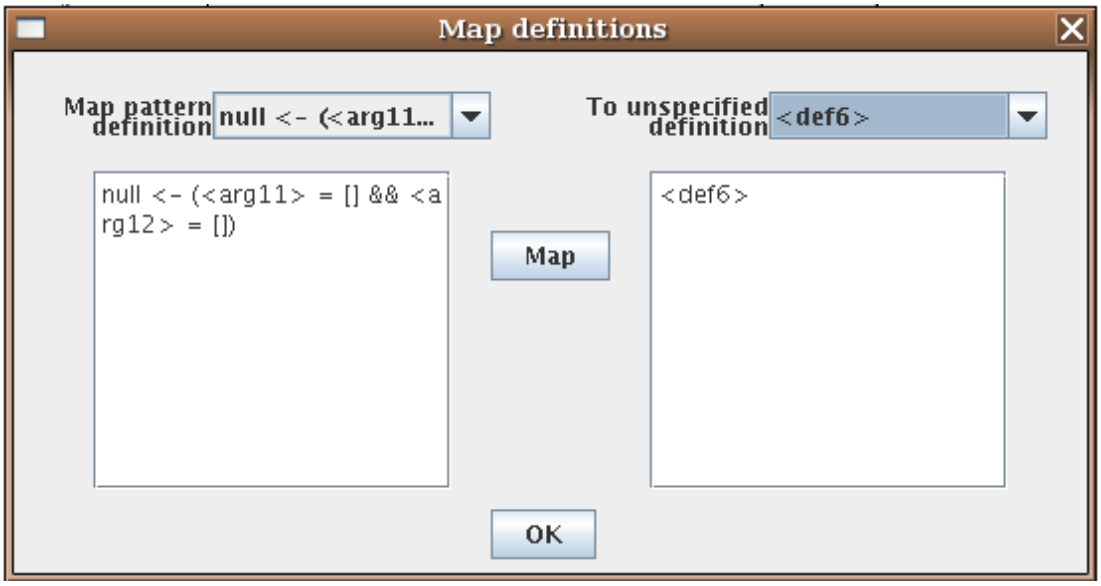

Figure A.13: Map the definition representing the terminating condition of the recursion to def6

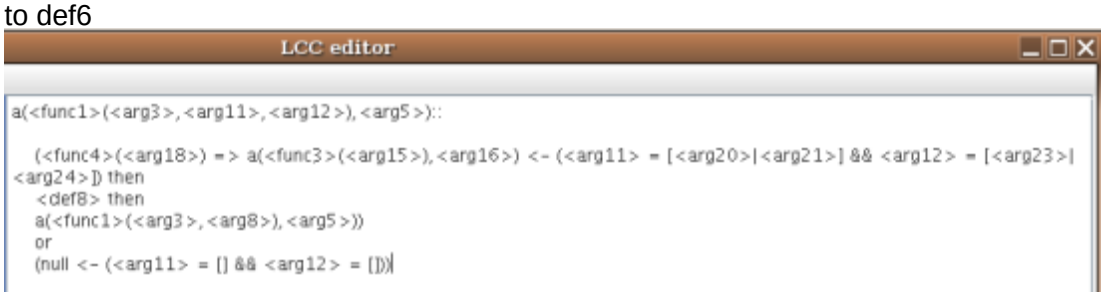

Figure A.14: The clause after applying the pattern

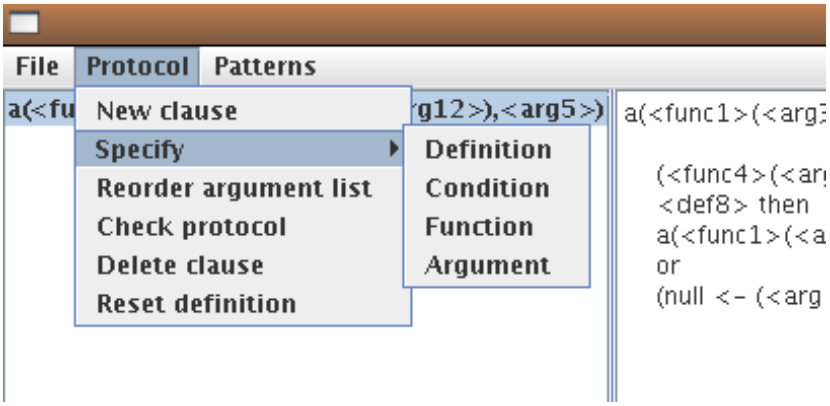

Figure A.15: Specify an argument

Figure A.16: Specify arg8 as an argument list

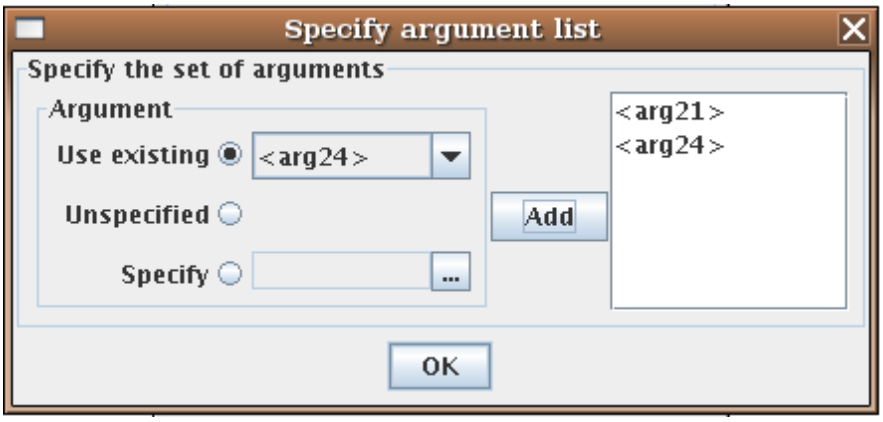

Figure A.17: Build an argument list with arg21 and arg24 as elements

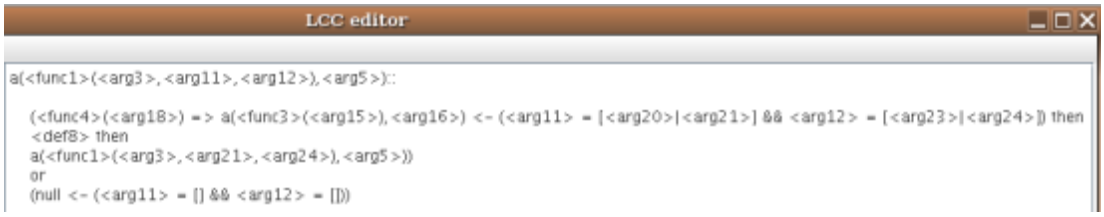

Figure A.18: The result after completing the application of the detailed pattern

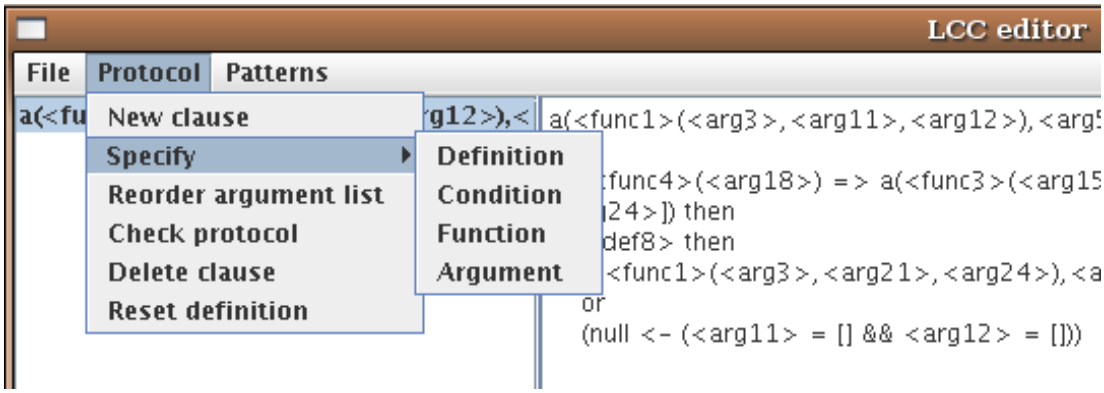

Figure A.19: Specify the argument arg8

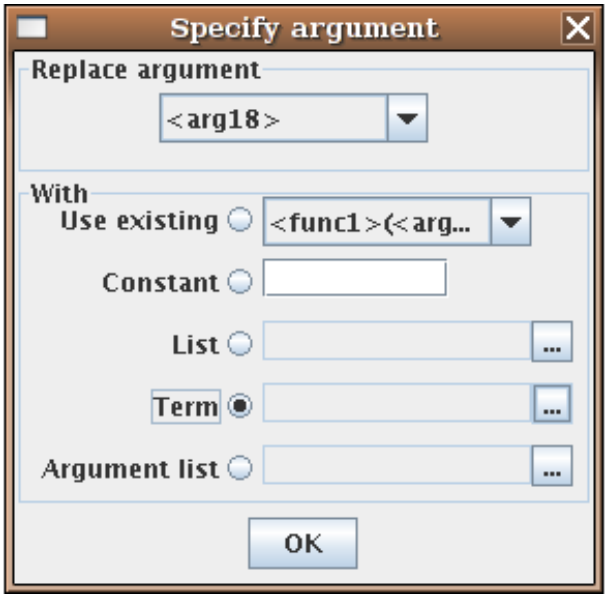

Figure A.20: Replace it with a term

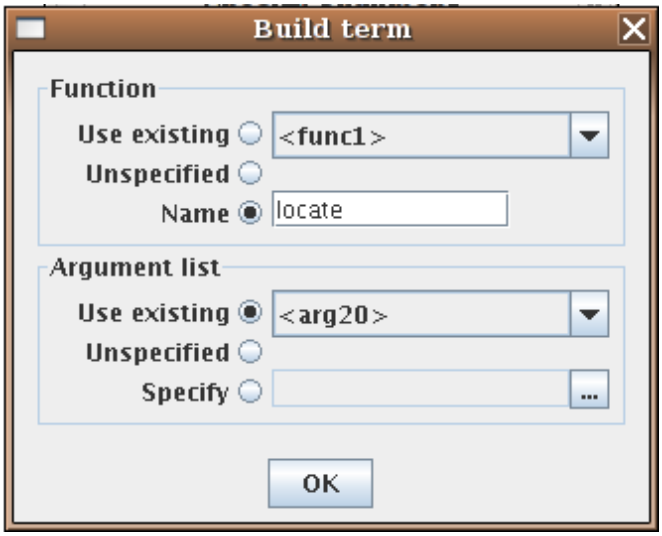

Figure A.21: Specify the term

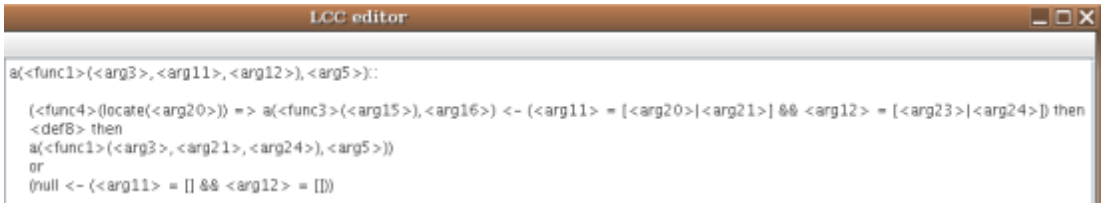

Figure A.22: The clause after specifying arg8

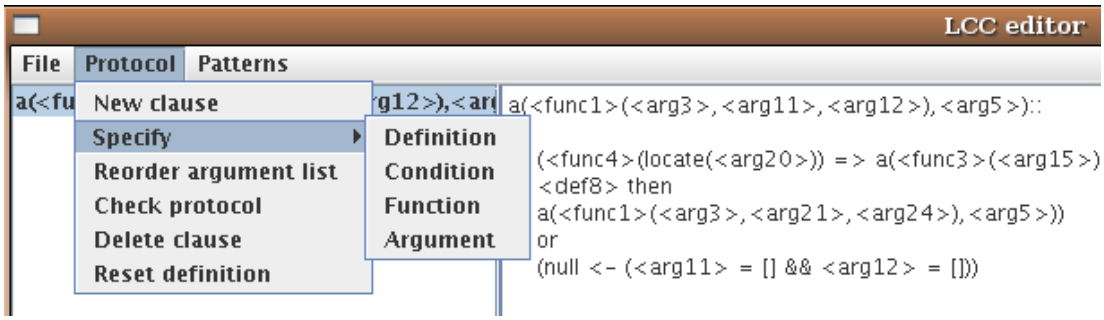

Figure A.23: Specify the definition def8

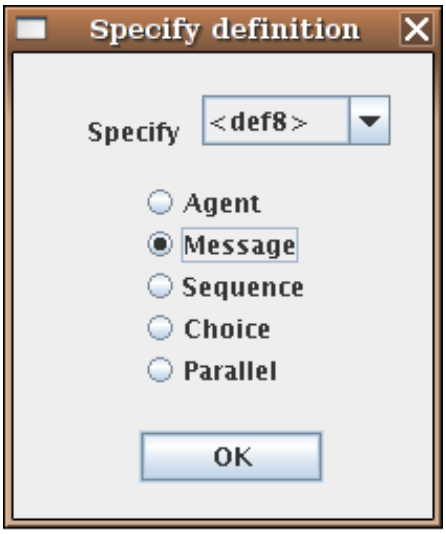

Figure A.24: Replace it with a message

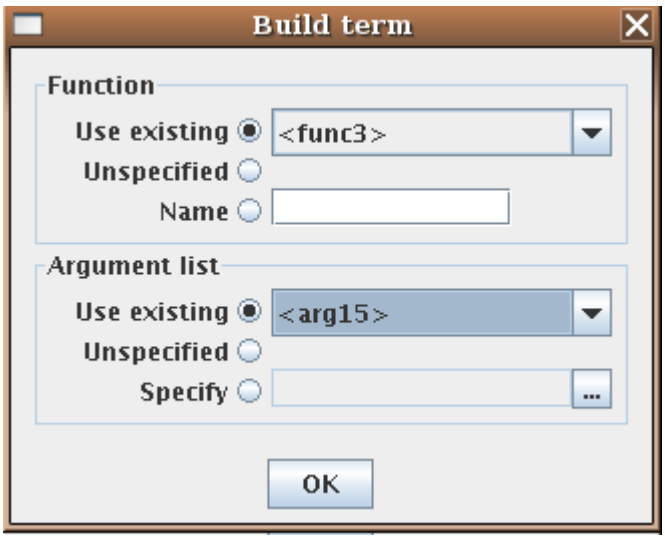

Figure A.25: The role of the agent sending the message is the same with one receiving the outgoing message

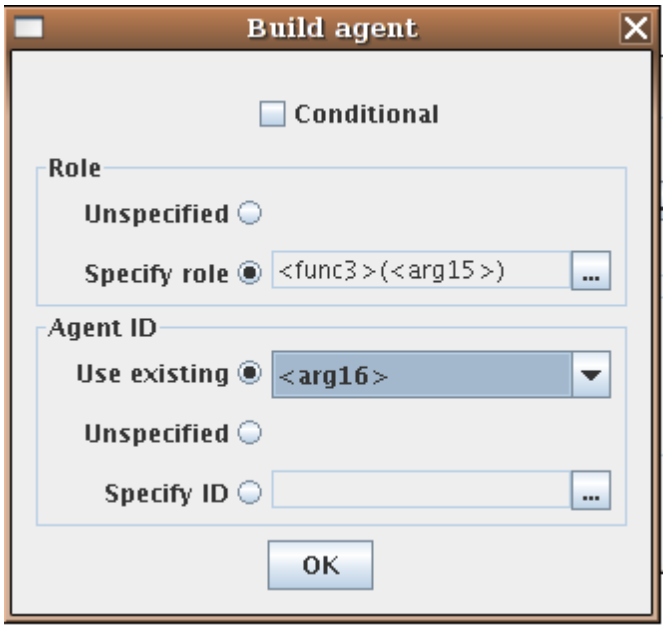

Figure A.26: The identifier of the agent is also the same

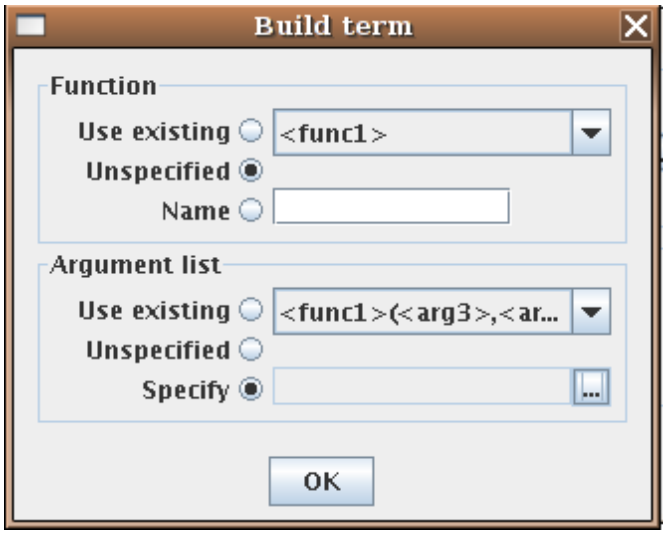

Figure A.27: The term representing the message

Figure A.28: The content of the message is another term

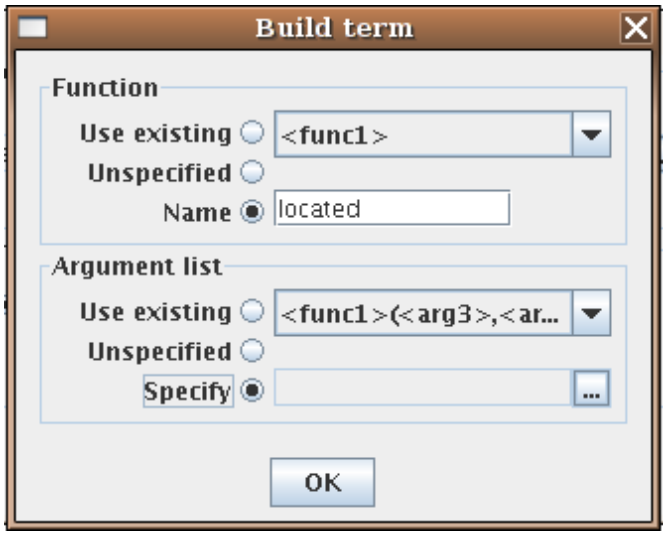

Figure A.29: Define the term representing the content

Figure A.30: Specify its argument as an argument list

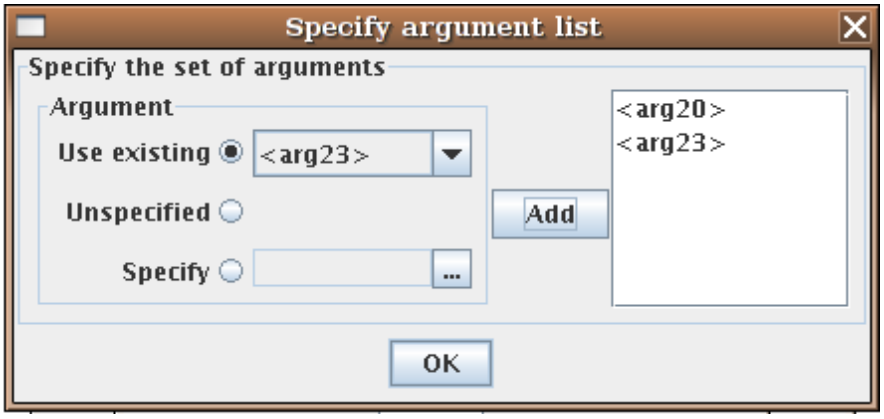

Figure A.31: The argument list consists of the remainders of the two lists in the condition of the outgoing message (arg20 and arg23)

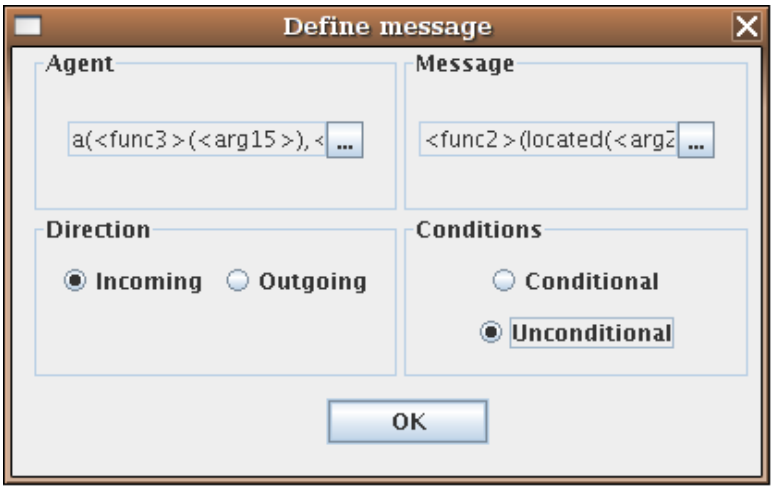

Figure A.32: The message as we have defined it

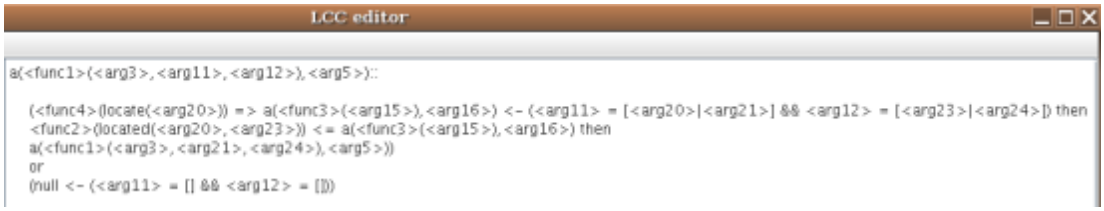

Figure A.33: The result after the addition of the message

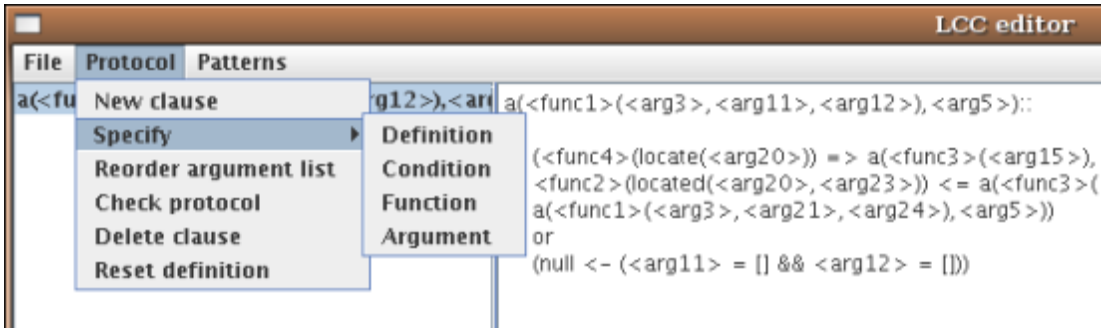

Figure A.34: Specify function

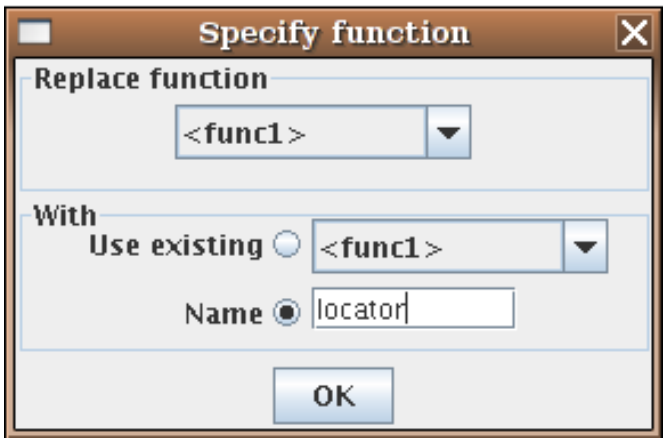

Figure A.35: Replace the functions with named ones

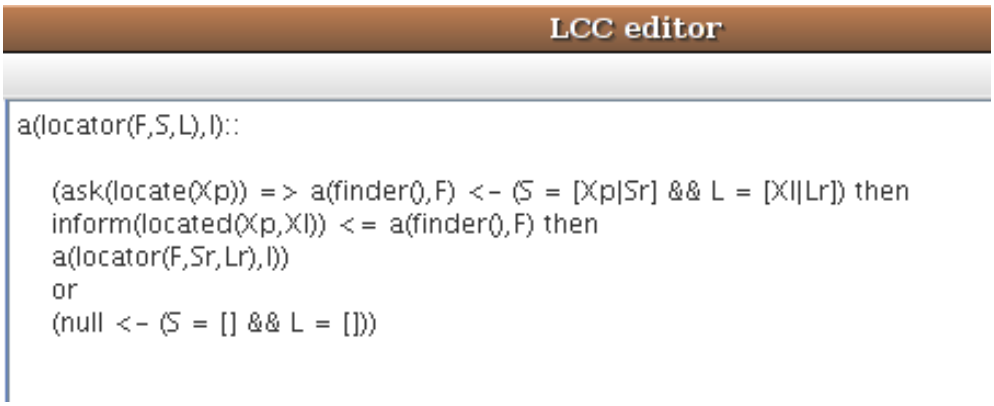

Figure A.36: The final result

Bibliography

[1] Bowles A. A techniques editor for prolog novices. Internal software report, available by the author, 1994.

[2] Bowles A., Robertson D., Vasconcelos W., Vargas-Vera M., and Bental D. Applying prolog programming techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, 1994.

[3] Walton C. Model checking agent dialogues. In *Proceedings of the 2004 Workshop on Declarative Agent Languages and Technologies (DALT)*, 2004.

[4] Walton C. Model checking multi-agent web services. In *Proceedings of AAAI Spring Symposium on Semantic Web*, 2004.

[5] Walton C. and Barker A. An agent-based e-science experiment builder. In *Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*, 2004.

[6] Robertson D. A simple prolog techniques editor for novice users. In *Proceedings of 3rd UK Annual Conference on Logic Programming*, pages 190–205. Springer-Verlag, 1991.

[7] Robertson D. A lightweight coordination calculus for agent social norms. In *AAMAS Workshop on Declarative Agent Languages and Technologies*, 2004.

[8] Robertson D. A lightweight coordination calculus for agent systems. Technical report, Informatics, University of Edinburgh, 2004.

[9] Robertson D. A lightweight method for coordination of agent oriented web services. In *Proceedings of AAAI Spring Symposium on Semantic Web Services*, 2004.

[10] Robertson D. Multi-agent coordination as distributed logic programming. In *"Logic programming" 20th International Conference, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 416–430, 2004.

[11] Robertson D., Correa da Silva F., Agusti J., and Vasconcelos W. A lightweight capability communication mechanism. In *Proceedings of the 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2000.

[12] Walton C. D. and Robertson D. Flexible multi-agent protocols. Technical Report EDI-INF-RR-0164, University of Edinburgh, 2002.

[13] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] FIPA Foundation for Intelligent Physical Agents. *FIPA Specification Part 2 - Agent Communication Language*, 1999.

[15] McGinnis J., Robertson D., and Walton C. Using distributed protocols as an implementation of dialogue games. In *Proceedings of the First European workshop on Multi-Agent Systems*, 2003.

[16] Austin J. L. *How to Do Things With Words*. Oxford University Press, 1962.

[17] Esteva M., Rodriguez J. A., Sierra C., Garcia P., and Arcos J. L. On the formal specification of electronic institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, number 1991 in Lecture Notes in Artificial Intelligence, pages 126–147. 2001.

[18] Greaves M., Holmback H., and Bradshaw J. What is a conversation policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies*, Autonomous Agents '99.

[19] Kirschenbaum M., Lakhotia A., and Sterling L. Skeletons and techniques for prolog programming. Technical Report 89-170, Case Western Reserve University, 1989.

[20] Brna P., Bundy A., Dodd T., Eisenstadt M., Looi C.K., Pain H.and Robertson D., Smith B., and Van Someren M. Prolog programming techniques. *Instructional science*, 20(2/3):111–134, 1991.

[21] Coalition D. S. DAML-S: Semantic markup for web services. Technical report, DARPA Agent Markup Language group, 2003.

[22] Finin T., Fritzson R., McKay D., and McEntire R. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management CIKM'94*. ACM Press, 1994.

[23] Vasconcelos W. Designing prolog programming techniques. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Tranformation*. Springer Verlag, 1993.