EXTENDING A PARALLEL CLP LANGUAGE TO SUPPORT THE DEVELOPMENT OF MULTI-AGENT SYSTEMS

Panagiotis Stamatopoulos **Dimitris Margaritis** **Constantin Halatsis**
University of Athens, Department of Informatics

Abstract

An extension of the parallel constraint logic programming language ElipSys is presented. This extension is directed towards the development of multi-agent systems which have to deal with large combinatorial problems that are distributed in nature. Problems of this kind, after being decomposed into subproblems, may be tackled efficiently by individual agents using ElipSys' powerful mechanisms, such as parallelism and constraint satisfaction techniques. The proposed extension supports the communication requirements of the agents, in order to have them cooperate and solve the original combinatorially intensive problem. The communication scheme among the agents is viewed as a three-layered model. The first layer is socket oriented, the second realizes a blackboard architecture and the third supports virtual point-to-point interaction among the agents.

Keywords

multi-agent systems, distributed execution, parallel CLP

Introduction

Multi-agent systems is a major research area of Distributed Artificial Intelligence (DAI) [9, 2], in which agents of various types and capabilities cooperate in problem solving. Besides its very challenging nature, the research in this area is motivated by the need for large scale programming to attack intensive problems, as far as the sizes of the required knowledge and computation are concerned. What is more is that advances in parallel computing are exploited, both at the machine and the programming levels.

The approach that has to be followed for the development of a multi-agent system is to build a separate subsystem for each problem domain and, then, make these subsystems cooperate. There are many advantages in a procedure like that. Firstly, the modularity achieved reduces the complexity of the whole system, which, in addition, is more reliable, in the sense that it can continue to operate even if part of it fails. Moreover, there might be a speedup in execution, since the subsystems can operate in parallel. Another advantage is that the subsystems are reusable. Finally, the knowledge acquisition procedure is facilitated, since it is easier to find experts in narrower domains.

In the multi-agent systems research area, the main problems at the theoretical level are the development of representation techniques for the formal specification of agents, cooperation etc. At the pragmatic level, open problems are related to the required computational models and the programming environments. The choice of a programming environment depends on the capabilities of the adopted system in conjunction with the nature of the application. Search-based distributed applications are computationally intensive and require a powerful underlying system to cope with the combinatorial nature of the problems involved. In addition, this system should offer the appropriate features to handle the distributivity of the application. In this paper, staying at the pragmatic level, a framework for the development of search oriented multi-agent systems in the ElipSys language [1, 6] is presented. ElipSys is a parallel constraint logic programming system developed at the European Computer-Industry Research Centre (ECRC).

The framework which is proposed is an extension of the ElipSys language as a three-layered communication scheme that facilitates the interaction among ElipSys agents. The flow of information among the agents is carried out, at the lower layer, through stream sockets over a TCP/IP network. Various Prolog systems, such as SICStus Prolog [11], Quintus Prolog [10], ECL^{i}PS^e [5] etc., provide the socket facility. However, much programming effort is required to build multi-agent systems directly on top of socket I/O. SICStus Prolog goes a step further, since it provides also a library with some higher level blackboard-based communication facilities, which are of the client/server variety. A client program may connect to the server and, then, read and write arbitrary terms. The work which is presented in this paper adopts, as a second layer, the blackboard approach, which forms the basis of a third layer for virtual point-to-point agent communication. This layer provides the higher level facilities, which may be used by an ElipSys programmer for the development of multi-agent systems.

In the following, a brief introduction to the ElipSys language is

given. Next, the communication oriented extension of the language is presented in detail and, finally, the proposed approach is discussed and compared to related systems.

The ElipSys Language

ElipSys is a parallel logic programming language which incorporates various powerful execution mechanisms. The language supports OR-parallelism, AND-parallelism, data-parallelism, data driven computation and constraint satisfaction techniques over finite domains. All these features make the language ideal for stating and describing combinatorial problems.

OR-parallelism aims at the concurrent exploration of the various alternative clauses that define an ElipSys procedure. AND-parallelism results from the concurrent execution of two goals in conjunction and data-parallelism is a kind of parallelism arising from the concurrent treatment of the elements of a set of data. In an ElipSys session, the user supplies the number of the desired workers, which are mapped, during parallel execution, to real processing elements, if available. In addition, ElipSys provides a data driven computation rule on top of the usual depth-first left-to-right execution strategy of Prolog. This rule modifies the reduction order of goals according to instantiations of variables. The language also supports constraint satisfaction techniques over finite domains [12], which lead to a priori pruning of the search space of a problem and thus result in more optimized execution. Finally, an interface with the C language is offered, which allows the programmer to define customized built-in predicates and link them with the core system.

ElipSys has been designed for maximum portability between different parallel platforms. It is currently available on various machines, namely Sun workstations (in sequential and pseudo-parallel mode), Sequent Symmetry, Sparc based multiprocessors from Sun and ICL and the KSR-1 distributed memory machine.

A Three-layered Communication Scheme in ElipSys

The main features of the ElipSys language aim to support the efficient solution of problems that may be mapped to a search of OR-trees. The parallelism offered may serve as a means for the concurrent exploration of branches of these trees and the constraint satisfaction mechanism may be used to prune the OR-trees and reduce the search space. However, this computational environment is not sufficient to support the development of multi-agent systems, because a framework is needed to allow the agents to exchange information. In any case, the searching facilities of ElipSys are indispensable at the intra-agent level, especially if someone has to cope with combinatorial problems.

For this reason, an appropriate extension of ElipSys is needed that can be integrated smoothly with the core system and is able to support the interaction requirements of multi-agent systems. The aim is to allow an agent to be viewed as an ElipSys program and to be identified by a name. For every agent, there has to be a way to send requests to other agents and to get answers to these requests. The coupling of a request with the respective answer is necessary, since an agent may send more than one request to another agent and get the answers at any time. It is also required that the requests and the answers have to be arbitrary ElipSys terms. Moreover, a desirable feature is to be able to have more than one agent of the same type for the purpose of load sharing.

Following the above requirements, a set of predicates has been designed which is sufficient to support transparent point-to-point communication among agents in ElipSys. To this end, a low-level communication framework has been adopted, namely stream sockets in the Internet domain, on which these predicates are based.

The direct implementation in C of the required predicates via socket oriented system calls is very complicated and difficult to debug and maintain. Thus, it has been decided to implement the appropriate primitives in C, through the operating system low-level calls, and enhance ElipSys, via the ElipSys to C interface, with the corresponding built-ins. Actually, these built-ins form a low layer of communication, where ElipSys processes running on different, or even the same, machine(s) may be connected and exchange messages. These messages are simple strings.

Unfortunately, the implementation of the required predicates for point-to-point agent interaction directly on top of the low socket layer presents many disadvantages, such as:

- There may be many connections among the agents which can increase super-linearly with the number of agents of the same type.
- An application based on such a framework has irregular structure which is not easy to understand and maintain.
- It is difficult to have more than one agent of the same type and to vary the number of the agents dynamically.
- It is necessary for all agents to know the network location of the agents that they want to cooperate with.
- A predefined order of agent initiation is required, so that deadlocks will not occur during the startup of the system.

Having in mind the above, it has been decided to introduce a medium layer between the low-level built-ins and the high-level predicates. This layer realizes a blackboard structure where clients may connect and then read and write information (actually arbitrary ElipSys terms). The format of these terms is irrelevant and is not examined by the blackboard. The functionality provided is similar to that of the client/server library of SICStus Prolog.

The blackboard is an ElipSys program which may be started via a specific predicate and begin listening for connections at a user supplied port number. It has been implemented in ElipSys using the low-level socket built-ins. The clients are agents, that is ElipSys programs, which are supported by a set of predicates to handle connections, disconnections and I/O with the blackboard. These predicates are also implemented in ElipSys via the low-level

built-ins. A client may connect to the blackboard, if it knows the hostname of the machine where the blackboard is running and the port number where it listens for connections. The blackboard functions as a central network-wide communication area. It uses its ElipSys workspace to assert and retract information according to the requests coming from the clients. I/O is unification based, rather than simple pattern matching. Another interesting characteristic of the blackboard is that it may also function as a broadcast medium. This latter feature is in fact exploited by the agents to advertise work to be done. This work can be assigned to be executed by any agent requesting this kind of work.

The predicates required to support the development of multi-agent systems form a high layer which has been implemented in ElipSys on top of the medium blackboard layer. These are the high-level predicates that may be used by a programmer and they realize a virtual point-to-point interaction framework. In this way, all the disadvantages of the implementation of these predicates directly via the low-level built-ins have been eliminated. The only requirements for each agent are to know the network location of the blackboard and to begin execution after the blackboard has started listening to connections. The agents may be started in any order.

Thus, the communication scheme adopted to support distributed applications in ElipSys is a three-layered one. In a nutshell, these layers consist of the following:

Low-level built-ins: These built-ins implement all the required primitives to support the communication among processes through stream sockets in the Internet domain. They are implemented in C and connected to the core language through the ElipSys to C interface.

Medium-level predicates: These can be used to pass arbitrary terms between a blackboard server and various clients. This set of predicates is implemented in ElipSys on top of the low-level built-ins and supports the upper layer of the communication scheme.

High-level predicates: These predicates are tailored to be used for virtual point-to-point message passing, actually through the blackboard, by multi-agent applications. They are implemented in ElipSys on top of the medium layer.

In the following sections, the built-ins and the predicates that support the three layers are presented in more detail.

Socket-based Interprocess Communication

As was stated above, the low-level built-ins are very similar to actual system calls adapted to Prolog. With these built-ins one can communicate strings to another network entity. The format of the strings sent should be known by the other party in any case. The adopted format is a 4-byte integer followed by this number of ASCII characters which contain the actual string itself. Short descriptions of the low-level built-ins follow:

- `socket_create(`*−Socket*`)`:
  Create a stream *Socket* in the Internet domain.
- `socket_bind(`*+Socket, +Address*`)`:
  Bind a *Socket* to a name. *Address* has the form *Host*:*Port*.
- `socket_listen(`*+Socket, +Length*`)`:
  Establish a maximum backlog queue of *Length* number of pending connections.
- `current_host(`*?HostName*`)`:
  HostName is unified with the name of the host the ElipSys system is running on.
- `socket_connect(`*+Socket, +Address*`)`:
  Connect the socket *Socket* to the specified address *Address*.
- `socket_accept(`*+Socket, −NewSock, ?FromHost*`)`:
  Accept the first pending request for connection on socket *Socket*. The new socket available for I/O is returned in *NewSock*, while *Socket* is still waiting for new connections. The hostname and port from which the remote connection is being attempted is unified with *FromHost*.
- `socket_select(`*+WtSocks, ?RdSocks, +TimeOut*`)`:
  Return the list of socket descriptors that will not block upon reading in *RdSocks*, from the list that is input in *WtSocks*. *TimeOut* is of the form *Secs*:*Usecs*. The call will return after *Secs* seconds and *Usecs* microseconds if no descriptor becomes available and *RdSocks* will be `[]`. The call will wait indefinitely if *TimeOut* is instantiated to the atom `off`. If one of the sockets in *WtSocks* has been executed a `socket_listen/2` call upon, an appearance of this socket descriptor in *RdSocks* means that a subsequent `socket_accept/2` call on this socket will not block.
- `socket_close(`*+Socket*`)`:
  Close the specified *Socket*.
- `socket_write(`*+Socket, +String*`)`:
  Write the string *String* to the specified *Socket*.
- `socket_read(`*+Socket, −String*`)`:
  Read the string *String* from the specified *Socket*.

Blackboard and Clients

The medium-level predicates are used by the blackboard and a client. These predicates are written in ElipSys and use the low-level built-ins described previously. They can be used to communicate arbitrary ElipSys terms, limited only by the length of an ElipSys string (64K). The terms that are passed by the client medium-level predicates to the low-level built-ins (to be subsequently converted to strings and sent to the blackboard) have a certain format. So do the responses returned from the blackboard. They both include, in addition to the terms involved, some information in their functor as to what was the medium-level predicate that was called. This information dictates the behaviour of the blackboard in response to the call, but do not affect in any way the terms, which are not interpreted at this level.

The blackboard and client medium-level predicates are:

- `blackboard(+Port)`:
  Start the blackboard server on the current machine. The blackboard starts listening to *Port* for incoming connections.
- `blackboard_connect(+Address, -Socket)`:
  Connect to the blackboard server which resides at *Address*. *Address* is of the format *Host:Port*.
- `blackboard_disconnect(+Socket)`:
  Disconnect from the blackboard server.
- `out(+Socket, +Term)`:
  Send the term *Term* to the blackboard server.
- `in(+Socket, ?Term)`:
  Read and retract term *Term* from the blackboard server. Blocks if *Term* is unavailable until it is written on the blackboard by another client.
- `in_noblock(+Socket, ?Term)`:
  Like `in/2` but return immediately. Fail if *Term* is not on the blackboard.
- `in(+Socket, +TermList, ?Term)`:
  Any one of the terms in *TermList* is removed and returned when available on the blackboard after being unified with *Term*.
- `rd(+Socket, ?Term)`:
  Wait until term *Term* appears on the blackboard and return it when it does. Do not remove it from the blackboard.
- `rd_noblock(+Socket, ?Term)`:
  Like `rd/2` but return immediately. Fail if *Term* is not on the blackboard.
- `rd(+Socket, +TermList, ?Term)`:
  Any one of the terms in *TermList* is returned when available on the blackboard after being unified with *Term*.
- `unique_id(+Socket, +Counter, ?Value)`:
  Generate a blackboard-wide unique value for the specified counter. *Counter* can be any number or atom. Counters do not need initialization. A counter that is used for the first time by a client is automatically initialized to 0.

Virtual Point-to-point Agent Interaction

In order to support the development of multi-agent systems in the ElipSys language, in a somehow transparent way, a third layer consisting of some high-level predicates has been implemented on top of the previous medium-level client/server predicates. A concept used at this level is that of an agent which is actually a program running under ElipSys. This third layer provides a virtual point-to-point communication model among the agents. The relevant predicates are implemented in ElipSys and their design is based on the requirement that an agent may need to send a request to another agent and, in most cases, it has to get an answer to its request. The request is identified via the blackboard by a unique identifier, so as to couple it with the expected answer.

Special care is taken for the high-level predicates to work correctly in a parallel environment. To this end, a dynamic connection approach has been adopted, which means that for every communication transaction, an agent connects to the blackboard, sends or receives data and then disconnects. In this way, multiple connections from an agent to the blackboard may be active at any time, actually from parallel ElipSys threads.

What is sent to the blackboard by an agent, both for requests and for answers to requests, is an ElipSys term that encodes the names of the sending and the receiving agents, the message itself and the unique identifier of the message.

The high-level predicates that realize the agent-oriented communication model are the following:

- `send_request(?Agent, +Request, -Id)`:
  Send the request *Request* to the agent *Agent*. The unique identifier of the request is returned as *Id*.
- `send_answer(?Agent, +Answer, +Id)`:
  Send the answer *Answer* to the agent *Agent*. This answer corresponds to the request whose identifier is *Id*.
- `get_message(?Agent, ?Message, ?Id)`:
  Get either a request or an answer *Message*, whose identifier is *Id*, from agent *Agent*. Block till a message is available.
- `get_message_noblock(?Agent, ?Message, ?Id)`:
  Similar to `get_message/3`, but fail if message is not available and return immediately, rather than block.

These predicates assume that every agent has a unique name which is defined in the agent's code by a specific fact of the form `myself(AgentName)` where *AgentName* is the name of the agent. Another assumption is that every agent knows where the blackboard is running. Thus, a fact of the form `blackboard_address(Host:Post)` has to be defined in the code of each agent, where *Host* is the name of the machine where the blackboard is running and *Port* is the port where it listens for connections.

Discussion

The three-layered communication scheme which was presented in the previous sections, as an extension of the parallel CLP language ElipSys, is a modular framework that supports the solution of combinatorial problems by decomposing them into mutually dependent subproblems. In this way, a multi-agent system may be designed, where each agent tackles a specific subproblem by using the search oriented features offered by the language. The extension proposed in this paper is responsible for handling the communication issues among the agents.

Taking into consideration the above, the extended ElipSys language is unique for the specific kind of applications mentioned previously. What is more is that even for applications where the language is partially exploitable, there still exists the possibility to design them using a multi-agent architecture, where some agents may be developed under different environments. These

environments may be different languages or even different hardware platforms. The only requirement is to adapt the three-layered communication scheme to the other language and/or machine and respect the communication protocol of each layer. This is the case of an application on tourism, called MaTourA (Multi-agent Tourist Advisor) [7, 8], which is currently under development in the ESPRIT III 6708 APPLAUSE Project (Application & Assessment of Parallel Programming Using Logic). MaTourA comprises a set of autonomous agents (Tour Generation Agent, Activity Agent, Event Agent, Site Agent, Accommodation Agent, Transportation Agent, Ticketing Agent etc.). Each agent reflects procedures involved in a tourist advisory environment, accomplishes specific functionality and manipulates specific knowledge. These agents may either answer queries on their own or coordinate their knowledge and communicate with each other, in case joint action is required. The most complex problem that MaTourA has to solve is the construction of personalized tours that satisfy the tourists' wishes. A tour is a journey through various sites where the tourist may visit activities and attend events. Accommodation, transportation and ticketing information is also included in a tour's description. Most of the MaTourA agents are being developed in the extended ElipSys under UNIX on Sun workstations. Some other agents are going to be implemented in ECL'PS^e in the same environment and the User Interface Agent is currently being designed to run under MS windows on PC platforms.

As far as related work is concerned, there are various logic programming systems that provide some kind of support for the development of multi-agent systems. Apart from many commercial Prolog languages that provide socket-based features, more advanced systems are IC Prolog II [4] and Shared Prolog [3]. IC Prolog II is a multi-threaded Prolog system which includes also a Parlog subsystem, high-level communication primitives and an object-oriented extension. The high-level communication primitives support pipes for interaction among threads as well as stream and datagram sockets for interaction among various Prolog processes. The concept of mailboxes is also provided as a means for a more abstract communication model. Shared Prolog is a concurrent language which is useful for coordinating communication and synchronization of agents via a blackboard structure. An agent is a Prolog program extended with a guard mechanism which governs the message exchange with the blackboard. There are no explicit primitives for communication. However, these languages as well as others with similar features, such as Delta Prolog, CS-Prolog, PMS-Prolog, Multi-Prolog and Linda Prolog, do not provide the necessary facilities to tackle combinatorial problems in a distributed environment, as the extended ElipSys language does.

Conclusions

A three-layered communication scheme was presented as an extension of the parallel constraint logic programming language ElipSys. This communication scheme is based at the lower level on stream sockets which are used for message passing among ElipSys processes over a TCP/IP network. A blackboard/client architecture was built on top of the previous level and, finally, virtual point-to-point interaction is supported at the higher level.

The proposed framework is suitable for developing multi-agent systems in ElipSys, where the agents themselves may exploit the parallelism and the constraint satisfaction techniques offered by the language. The whole system, i.e. the extended ElipSys language, is extremely useful for search-based applications that are distributed in nature.

Acknowledgements

This work is partially funded by the ESPRIT Programme of the Commission of the European Communities as part of the APPLAUSE ESPRIT Project 6708. The authors would like to thank the APPLAUSE Team of the Athens University for the fruitful discussions they had on the presented system. Special thanks are also directed to ECRC Gmbh (Munich, Germany) for providing the ElipSys language.

References

[1] U. Baron, S. Bescos, S. A. Delgado-Rannauro, P. Heuzé, M. Dorochevsky, M. Ibáñez-Espiga, K. Schuerman, M. Ratcliffe, A. Véron, and J. Xu. The ElipSys logic programming language. Technical Report DPS-81, ECRC, December 1990.

[2] A. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[3] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[4] D. Chu and K. Clark. IC Prolog II: A multi-threaded Prolog system. In *Proceedings of the ICLP'93 Post Conference Workshop on Concurrent, Distributed & Parallel Implementations of Logic Programming Systems*, pages 115–141, 1993.

[5] *ECL'PS^e: User Manual*, March 1993.

[6] *ElipSys User Manual for Release Version* 0.6, April 1993.

[7] C. Halatsis, P. Stamatopoulos, I. Karali, C. Mourlas, D. Gouscos, D. Margaritis, C. Fouskakis, A. Kolokouris, P. Xinos, M. Reeve, A. Véron, K. Schuerman, and L.-L. Li. MaTourA: Multi-Agent Tourist Advisor. In *Proceedings of the International Conference on Information and Communications Technology in Tourism*. Springer Verlag, 1994.

[8] C. Halatsis, P. Stamatopoulos, D. Margaritis, I. Karali, C. Mourlas, D. Gouscos, and C. Fouskakis. Tool assessment. APPLAUSE Deliverable D.WP3.4, University of Athens, May 1993.

[9] M. Huhns, editor. *Distributed Artificial Intelligence.* Pitman, London, 1987.

[10] *Manual for Quintus Prolog Release 3.1*, 1991.

[11] *SICStus Prolog User's Manual*, August 1992.

[12] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1989.