Communicating Sequential Processes for Distributed Constraint Satisfaction ⋆

Ilias Sakellariou [a,*], Ioannis Vlahavas [a], Ivan Futo [b], Zoltan Pasztor [b], Janos Szeredi [b]

[a] *Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, 54124, Greece*

[b] *ML Consulting and Computing Ltd, ML Kft, H-1011 Budapest, Gyorskocsi u. 5-7., Hungary.*

Abstract

CSPCONS is a programming language that supports program execution over multiple Prolog processes with constraints. The language is an extended version of CSP-II, a version of Prolog that supports channel-based communicating processes and TCP/IP communication, that is based on the CSP model introduced by Hoare. CSPCONS inherits all the advanced features of CSP-II and extends it by introducing constraint solving capabilities to the processes. In CSPCONS each Prolog process has one or more solvers attached and each solver is independent from the others, following the original CSP-II model, thus resulting to a communicating sequential constraint logic programming system. Such a model can facilitate greatly the implementation of distributed CLP applications. This paper describes the original CSP-II system along with details of the extensions that resulted to the CSPCONS system and presents an example demonstrating the applicability of the system to distributed constraint satisfaction problems.

Key words: Distributed Constraint Logic Programming, Communicating Sequential Processes, Prolog

⋆ This work was supported by the Bilateral Cooperation Program Greece-Hungary 2000-2002

* Corresponding Author.

Email addresses: iliass@csd.auth.gr (Ilias Sakellariou), vlahavas@csd.auth.gr (Ioannis Vlahavas), futo@ml-cons.hu (Ivan Futo), pasztor@ml-cons.hu (Zoltan Pasztor), szeredi@ml-cons.hu (Janos Szeredi).

1 Introduction

A large number of problems appearing in diverse areas such artificial intelligence, engineering, operational research, scheduling, planning and resource allocation can be modeled as constraint satisfaction problems (CSP). Informally, such a problem consists of finding a valid assignment of values to variables under a set of constraints on the variables. Despite its simple statement, the problem belongs to the NP-complete class of problems, thus its solution demands sophisticated techniques. This led to the emergence of constraint programming (CP), a new area that aims to provide the necessary support to efficiently describe and solve large constraint satisfaction problems. CP is based on a strong theoretical foundation and includes extensions to a variety of programming languages that belong to different programming paradigms, such as logic programming, object oriented programming, etc.

Historically constraint logic programming (CLP) was the first CP paradigm, mainly due to the fact that its declarative nature offered a natural way to state constraints. CLP is an extension of traditional logic programming languages such as Prolog, in which the standard unification mechanism is replaced by constraint satisfaction. In the past decade, CLP has reported a number of successful real-world applications and has been classified by the ACM as one of the "strategic directions" in computer science.

The availability of a large number of machines connected by some local network nowadays, naturally led to the idea of solving CSP distributively. In this direction, Distributed Constraint Programming investigates how multiple processing units, called workers or (recently) agents, cooperate to solve a CSP problem. Usually each agent "owns" a subproblem of the original problem, i.e a subset of the original variables and the constraints on them. Since it is hardly ever the case that the original problem is decomposed in such a way that there are no constraints on variables that belong to different agents, in such a setting agents need to communicate to achieve a consistent global solution while ensuring that the constraints of their own subproblem are not violated. The distributed approach is encouraged not only by the possible gains regarding efficiency that might emerge by such problem partitioning, but also by the fact that in many real-world applications problem information is distributed among different departments, thus if the problem is solved centrally, costly mechanisms for data communication and synchronization are required.

CSPCONS is a constraint logic programming language for building such DCSP applications. It is an extension of the Communicating Sequential Prolog II (CSP-II) a version of Prolog that is based on the notion of communicating sequential processes [1]. CSPCONS supports independent CLP processes, each having its own constraint store that communicate through message exchange

over channels and events. Communication is possible both between processes that reside in the same host and on different hosts over TCP/IP networks. Constraint solving programming facilities in CSPCONS are implemented in the form of C libraries, thus permitting the incorporation of new constraint systems or algorithms just by the addition of the appropriate library. The current version includes a library for constraint satisfaction over finite domains (FD) and reals.

The combination of the channel based communication and constraint satisfaction, all under the logic programming framework, offers a powerful platform for rapid prototyping and implementation of any distributed CSP application and an excellent testbed for DCSP algorithms. This paper extends our previous work described in [2] by providing a more detailed view of the facilities of the platform, changes to the finite domain solver and a more complex application demonstrating the advantages of our approach.

The rest of the paper is organized as follows. Section 2 briefly presents related work in the field of distributed constraint satisfaction. An overview of the features of the CSP-II language is presented in Section 3, considered necessary since all these are inherited to the CSPCONS language. The necessary extensions for the support of constraints together with the description of the implementation of the FD solver that form the CSPCONS language is given in Section 4. A simple distributed constraint version of the job-shop scheduling problem is presented in 5, to demonstrate the CSPCONS ability to easily implement such algorithms. Finally conclusions and future work are stated in section 6.

2 Distributed Constraint Satisfaction

A constraint satisfaction problem (CSP) consists of finding an assignment of values from a given domain to a set of variables, such that a set of constraints on the variables is satisfied. More formally a constraint satisfaction problem consists of:

- a set of variables X $x_1, x_2, ..., x_n$
- a set of domains D each associated with a variable $D_1, D_2, ..., D_n$, i.e. $x_i \in D_i$
- a set of constraints C that impose restrictions on the values that the variables can take. Each constraint $C_k(x_{k1}, ..., x_{km}), m <= n$ is a predicate on the Cartesian product $D_{k1} \times D_{k2} \times \ldots \times D_{km}$ that is true on a subset of the product, indicating that a value assignment that belongs to the subset is valid under the constraint. Constraints can be unary, binary or higher order depending on their arity.

In distributed constraint satisfaction problem (DCSP) variables and/or constraints are distributed over some network of agents, which are constraint solvers that co-operate to solve the original problem. A number of approaches have been reported to the literature that address the issue of building distributed constraint programming applications. The rest of this section provides a brief overview of languages and algorithms proposed for tackling DCSP problems.

2.1 DCSP Languages

Quite a few approaches to designing a suitable language for the implementation of distributed constraint applications have been reported. In the sequel we will restrict our attention to those that belong to the logic programming paradigm, as these are more related to our work.

CIAO is a logic programming system based on Prolog, extended with constraints, parallelism and concurrency [3,4]. Distributed execution in CIAO involves a number of workers (processing units), whose communication is based on the Linda library, a blackboard architecture, and the use of attributed variables[5]. The CSPCONS language presented in this paper has a completely different approach to communication: Prolog applications communicate via channels, thus allows a more robust and flexible communication schema.

A different approach to solving CSP problems in parallel has been proposed by Tong and Leung in [6]. Their model, called Firebird, is based on an extension of the Andorra principle and is an attempt to build a concurrent constraint logic programming system on a massively parallel SIMD computer, that will exploit OR-Parallelism. In Firebird execution interleaves between *indeterministic derivation steps* that consist of guard tests, commitment and spawning in the same manner as committed-choice languages and *non-deterministic derivation steps* which consist of setting up a choice point on a domain variable and attempting all the alternative values in its domain in an OR-parallel manner. However in Firebird parallelization is automatic, i.e. the language follows the spirit of "classic" parallel logic programming, whereas in CSPCONS the programmer has the freedom (and of course the burden) to define a cooperation/distribution schema custom to the application developed.

Probably the most successful distributed constraint language is Mozart [7,8] an implementation of the distributed OZ language[9]. Oz is a multi-paradigm language [10] that offers powerful constraint solving facilities and has been used in a number of projects. OZ (and Mozart) supports the logic programming paradigm along with concurrent object oriented functional programming and, to the opinion of the authors, presents a new powerful programming paradigm,

different than that CSPCONS follows, i.e extending a logic programming language to support distribution and constraints while maintaining the standard Prolog syntax. It has to be noted however that the average Prolog programmer might require some time to adopt to the syntax and programming philosophy of OZ, whereas in CSPCONS such a training period is not required.

Two of the most important representatives of the CLP paradigm are SICtus and ECLiPSe. Both platforms offer powerful constraint solving capabilities by providing a number of efficient solvers [11–14]. In both platforms communication between two applications over TCP/IP networks can be achieved through sockets that establish streams between two machines; the set of predicates offered maps directly to BSD-style socket functions. Of course, someone could argue that since both languages provide interface facilities to foreign languages, such as C/C++ and JAVA communication could be implemented via these, however such a solution would require from the programmer more effort to manage the development of the application.

In CSPCONS however, the programmer has more control over TCP/IP communication, than that provided by sockets, since a detailed mechanism for configuring connections to fellow applications is provided. But the extended communication facilities do not end here; the TCP/IP network interface provides an alert mechanism that informs the application about changes in the status of partners participating the community. Thus an agent in the community knows everything it needs to know for successfully cooperating with other agents in the community. The latter allows constructing a reliable communication infrastructure for the participating agents. In addition there can be a clear separation of the code handling communication with the rest of the code by assigning all communication related operations to a separate process.

Although a mixture of C/C++ and some solver library can equally serve in the case of building a distributed constraint application, the approach requires encoding everything in a lower level language thus increase development and maintenance times.

Finally, a prototype implementation of a multi-threaded version of SICStus Prolog with basic send/receive operations between threads is described in [15], which is close to the programming model of CSPCONS. However as shown in the following sections CSPCONS offers a richer set of programming primitives when it comes to communicating sequential processes programming. To our knowledge no language that combines communicating sequential processes, to the extent that CSPCONS does, with constraints has been proposed in the literature till now.

A number of algorithms have been proposed that address the issue of distributed constraint satisfaction. A class of these involves distributed/parallel versions of local consistency algorithms, i.e. they are distributed propagation algorithms. For example earlier work in the field involved two massively parallel versions of the AC-4 algorithm proposed by P. Cooper and M. Swain[16]. In [17] Y. Zhang and A. Mackworth present three parallel and distributed algorithms for computing consistency by formulating a CSP as a *dual network*, that were tested on a transputer based machine. The *DisAC-4* algorithm [18], is a coarse-grain distributed version of the AC4 algorithm, based on a message passing scheme for communicating domain reductions. In the same spirit an alternative distributed arc consistency algorithm, named *DisAC-9* with minimal message passing which is based on the variation of the AC-6 [19] consistency algorithm is presented in [20]. More recently a distributed constraint propagation algorithm based on chaotic iteration [21] was proposed in [22,23]. Finally, a distributed version of the singleton arc consistency algorithm, that is also based on a message passing scheme for the communication of domain inconsistencies can be found in [24].

The second class of algorithms performs distributed search using a set of agents that communicate with a message passing scheme.For example in [25,26] authors propose an asynchronous backtracking algorithm and its modification, the asynchronous weak-commitment search, that efficiently solves distributed constraint satisfaction problems. Other approaches involve algorithms for distributed backtracking algorithm (DIBT) [27], distributed forward checking (DIFC) [28], distributed dynamic backtracking (DisDB) [29] that combines ideas from ABT and DIBT, and an algorithm [30] that integrates a distributed bounds consistency algorithm, called DHC, with a distributed search technique called Asynchronous Aggregation Search [31]. A common characteristic of almost all distributed search algorithms mentioned above is information exchange is implemented through message passing, making the CSPCONS platform an ideal platform for their implementation using logic programing.

3 The CSP-II Prolog

Communicating Sequential Prolog II (CSP-II) [32,33] is a distributed logic programming system that follows the standard ISO Prolog syntax [1], but is further enriched with features like modularity, multitasking, real-time programming and network communication. CSP-II supports the communicating

[1] ISO/IEC 13211-1

sequential process programming methodology [1] in a Prolog environment. Thus the main concept of Csp-ii is a Prolog *process*. Processes run in parallel and communication between them is achieved through message passing over channels and events, when they reside to the same host (same application).

Channel-based communication is extended over TCP/IP networks, providing the ability to establish connections between different Csp-ii applications across the Internet. Furthermore, under this schema Csp-ii also provides communication with foreign (non CS-Prolog) applications, an interface to relational data base systems, real-time programming methods like cyclic behavior, reaction to predefined events, timed interrupts, etc.

The system consists of three main components: a compiler, a linker and a runtime system. The Prolog source is compiled into a binary format file containing the byte code. The compilation follows the Warren Abstact Machine (WAM) approach [34], probably the most widely adopted standard for compiling Prolog programs. It should be noted that the Csp-ii byte code is in some aspects different than the WAM code, however a complete description of these differences is outside the scope of this paper. The Csp-ii code is interpreted by a "byte code interpreter" when executing the CS-Prolog runtime system. Among other things the system includes a pre-processor similar to what is found in C compilers and an integrated development environment with a multi-window trace utility.

3.1 Csp-ii *Processes*

A Csp-ii process is defined as the execution flow of a Prolog goal that has its own Prolog execution environment and dynamic database. Its progress is independent of the execution of other processes. The separation dynamic databases ensures that processes may influence each other only by the supported communication techniques, i.e. channels, events and interrupts, or through external objects like files. A Csp-ii application consists of any number of processes as shown in Figure 1, that are identified by a unique system-wide symbolic name.

Two types of processes are supported:

- self-driven or normal processes, which is the most usual kind, and
- event-driven or real time processes.

A *self driven* process is characterized by its (Prolog) goal; after its creation, it will begin the execution of this goal. The non-fatal termination of a self-driven process is determined by the termination of its goal, after which it disappears from the Csp-ii system and will never reappear. Definition of self

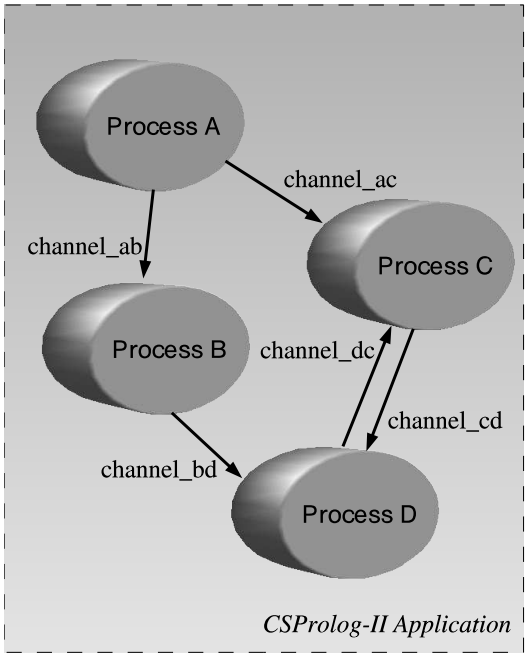

Fig. 1. Processes in a CSP-II application

driven processes is performed via the `new/3` built-in predicate; for instance the code below defines a self driven process named `processA`, with the Prolog goal `queens(Solution)`.

```
%%% Definition of a self-driven process
...
new(processA, queens(Solution)),
...
```

A *real time* process is characterized by one (Prolog) goal for the initialization, one goal for the event handling and by the description of the events that trigger its execution. The initialization goal is executed once and provides the means for performing any necessary setup actions; after its successful termination the process switches to a cyclic behavior. From that moment on it is controlled by the incoming events. For every real time process, the incoming events are gathered in a separate first-in-first-out input queue, from which the process consumes them by initiating its event-handling goal. The number of events that real time processes can be triggered for is unlimited. Successful termination of a process is signaled by the failure of its event-handling goal. Such termination is considered as regular; it does not affect the overall success or failure of the application. A real time process is defined by the built-in predicate `new_rt/5`, as shown below:

```
%%% Definition of a real time process
...
new_rt(processC,respond,pInit(Data),[stop,find,get],idle(600)),
...
```

The predicate in the example above defines a real-time process named `processC`, for which a predicate `respond` handles all incoming events, the predicate `pInit/1` performs all necessary initialization operations, the process responds to the events `stop`, `find` and `get`, and `idle(600)` defines that a timer event should be generated for this process if another explicit event is not generated for 6 seconds.

3.2 Inter-Process Communication

Inter-process communication is achieved by synchronous messages or by event passing. In the former case messages are passed through *communication channels*. A *message* can be any Prolog term except a single unbound variable, however compound terms containing unbound variables are allowed. Figure 2 presents an example of the code required for channel communication between processes A and B of figure 1. Communication channels act as system-wide available resources, identified by unique names and may appear and disappear dynamically during the program's lifetime. A channel implements an one way communication between two processes; in such a connection one process has the sending end of the channel and the other the receiving end. The total number of channels in the system and the number of the channels a process can be connected to are unlimited.

```
%%% Process A
   ...
   open_channel_for_send(channel_ab),
   send(sendChan, address(john_smith,Addr)),
   close_channel(channel_ab),
   ...
```

```
%%% Process B
...
open_channel_for_receive(channel_ab),
receive(sendChan,Message),
close_channel(channel_ab),
...
```

Fig. 2. Send Receive Operation between two Csp-ii processes

Events serve for triggering real time processes and are also identified by system-wide unique names. Event generation can occur either explicitly by the built-in predicate `generate_event/2` or implicitly by the internal clock of the Csp-ii scheduler. The latter allows to invoke execution of the real-time process in specific time intervals or when the process is idle for a certain amount of time (see example in section 3.1). The number of the available

events in a program is limited only by the system's resources and every occurrence of an event may have an optional data argument that carries additional information. As in the case of channel messages this data argument is an arbitrary Prolog term, except the case of a single unbound variable. For example the following predicate generates an event named `find` with the optional data argument `address(johnSmith)`.

```
...
generate_event(find,address(johnSmith)),
...
```

Finally, it should be noted that processes can backtrack, however communication is not backtrackable.

3.2.1 The Dining Philosophers' Problem in CSP-II

To illustrate further the the notions of processes and synchronized communication, this section presents a CSP-II implementation of the well known Dining Philosophers' problem. The problem concerns process resource allocation and was originally proposed by E.W.Dijkstra [35].

The problem is as follows: Five philosophers spend their lives thinking and eating. Eating takes place around a circular table surrounded by five chairs, one for each philosopher. In the middle of the table there is a large bowl of spaghetti and each philosopher has on its left a fork (figure 3). Each fork can only be picked up by one philosopher at a time. In order to eat, each philosopher has to sit down in his own chair and pick both forks on his left and right. After eating for while the philosopher puts down both his forks and gets up from his chair to continue thinking.

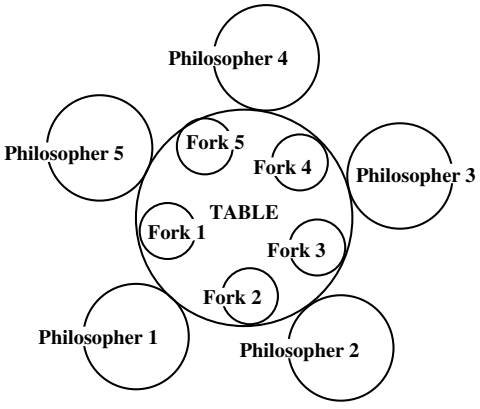

Fig. 3. The Dining Philosophers Problem

The solution to the problem presented in this section is the one proposed by Hoare in [36]. Each philosopher is modelled as a process, which executes the loop *sit down, pick up fork on the left, pick up fork on the right, eat, put*

down fork on the left, put down fork on the right and *get up*. Each fork is also modelled by a process that accepts messages (events in the original problem solution) from the two philosophers in its neighborhood. There are two types of messages that can be accepted by a fork process: a *pick_up_fork* and a *put_down_fork* message that correspond to the two philosopher actions. In the specific solution, deadlock avoidance relies on the existence of a "footman" i.e. a process that allows at most four philosophers to sit down. A detailed specification of the solution can be found in [36].

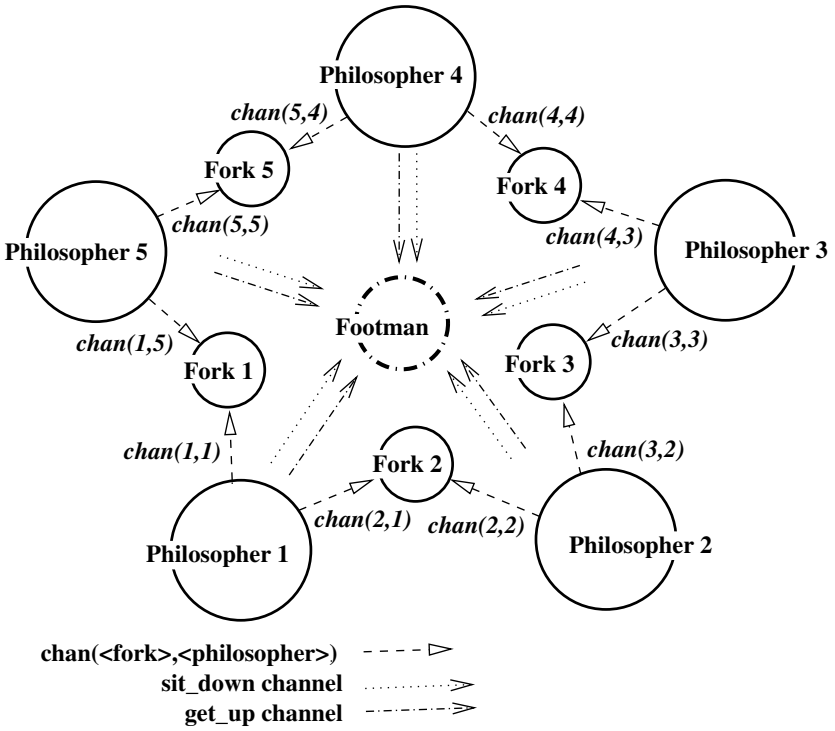

Fig. 4. CSP-II processes and channels for the Dining Philosophers' Problem

The communicating sequential processes solution can be directly mapped to a CSP-II Prolog implementation. Each "philosopher" and "fork" process becomes a CSP-II self-driven process and all processes communicate over synchronized channels, as shown in figure 4. For example, the `philosopher 1` process sends messages (events) to `fork 2` over the channel `chan(2,1)`. Arrows in the figure indicate the direction of messages in the channel; for instance, in the previous example `philosopher 1` holds the sending end of the `chan(2,1)` channel, while `fork 2` the receiver end of the same channel. All `chan(<fork>,<philosopher>)` channels are fixed, in the sense that both their ends are owned by the same processes during the execution of the program. The previous does not hold for the `sit_down` and `get_up` channels: although the receiving end of both these channels is always owned by the `footman` process, the sending end belongs to a different `philosopher` process each time, for reasons that will be explained in the following.

```
%%% Dining Philosophers Implementation in CSP
main_goal(_):-   new(footman,footman_top_level), % footman process
                 define_forks(5),define_philosophers(5),
                 start_processes.

%%% Definition of fork processes
define_forks(0).
define_forks(N):- N>0,NN is N-1,
                  new(fork_proc(N),fork_top_level(N)),
                  define_forks(NN).

%%% Definition of philosopher processes
define_philosophers(0).
define_philosophers(N):-
                  N>0,NN is N-1,
                  new(philosopher_proc(N),philosopher_top_level(N)),
                  define_philosophers(NN).
```

Fig. 5. Dining Philosophers in Csp-ii : Process definition

The Csp-ii code for the processes definition for the dining philosophers' problem is given in figure 5. The predicate `main_goal/1` is invoked by the Csp-ii runtime system. The predicate defines recursively all necessary processes by calling the `new/2` predicate described in section 3.1. For example the call `new(fork_proc(N),fork_top_level(N))` (where N is an integer between 1 and 5) defines a process `fork_proc(N)` with the top level goal `fork_top_level(N)`. Finally `main_goal/1` invokes process execution by calling the `start_processes` builtin.

```
%%% Top level predicate for philosopher process
fork_top_level(N):-
              P is 5 - ((1-N) mod 5),
              open_channel_for_receive(chan(N,N)),% right phil.
              open_channel_for_receive(chan(N,P)),% left phil.
              fork([chan(N,N),chan(N,P)]),
              close_channel(chan(N,N)), close_channel(chan(N,P)).

%%% Main loop (recursion) for the fork process
fork(CHANS):- receive(CHANS,MSG_UP,WINNER), MSG_UP=pick_up_fork,
              receive(WINNER,MSG_DOWN), MSG_DOWN=put_down_fork,
              !,fork(CHANS).
```

Fig. 6. Dining Philosophers in Csp-ii : Fork Process

Figure 6 presents the implementation of the fork process. There are five such processes in total and each such process owns the receiver end of two channels that correspond to the two philosophers, between which the fork is placed. Both channels are opened for receive by the `fork_top_level/1` predicate and are passed as an argument (list) to the `fork/1` predicate. The latter blocks

until a pick_up_fork message is received from either of the two channels. Upon the reception of the first message, the third argument of the receive/3 predicate is unified with the name of the *winner* channel, i.e. the channel over which the first message arrived. Then the process blocks, waiting for a put_down_fork message from the same (WINNER) channel, i.e. from the same "philosopher". Finally, the process loops by calling itself recursively.

```
%%% Top level predicate for philosopher process
philosopher_top_level(N):-
              F is (N mod 5) + 1,
              open_channel_for_send(chan(N,N)), %% LEFT FORK
              open_channel_for_send(chan(F,N)), %% RIGHT FORK
              philosopher(N,[chan(N,N),chan(F,N)]),
              close_channel(chan(N,N)), close_channel(chan(F,N)).

%%% Main loop (recursion) for the philosopher process
philosopher(N,[LEFT,RIGHT]):-
              sit_down_at_table(N),
              pick_up_fork(LEFT), pick_up_fork(RIGHT),
              eat(N),
              put_down_fork(LEFT), put_down_fork(RIGHT),
              get_up_from_table(N),
              !, philosopher(N,[LEFT,RIGHT]).

%%% Philosopher's actions
sit_down_at_table(N):-
              open_channel_for_send(sit_down,scheduled),
              send(sit_down,N), close_channel(sit_down).

pick_up_fork(CHAN):- send(CHAN,pick_up_fork).

put_down_fork(CHAN):- send(CHAN,put_down_fork).

eat(N):- format('Philosopher ~w is eating ~n',[N]).

get_up_from_table(N):-
              open_channel_for_send(get_up,scheduled),
              send(get_up,N), close_channel(get_up).
```

Fig. 7. Dining Philosophers in CSP-II :Philosopher Process

The CSP-II code for the philosopher process is shown in figure 7. The philosopher_top_level/1 process opens two channels for sending messages, each corresponding to a fork on the philosopher's side, after which the main loop of the philosopher process is executed (predicate philosopher/2). Most of the philosopher's actions are self explained, for instance picking up a fork is actually sending a pick_up_fork message to the corresponding channel (predicate pick_up_fork/1).

A point that requires a bit more clarification is the `sit_down_at_table/1` predicate, that provides synchronization with the `footman` process. The sending end of the channel `sit_down` is owned by different philosopher processes during the execution of the program. To obtain this end, i.e. permission to sit down, the channel must not be owned by another (philosopher) process. If the channel at the time of the call is owned by a different process, then the call to `open_channel_for_send/2` blocks until the channel is released by the process that owns it (i.e. a `close_channel/1` call by that process). This blocking behavior is indicated by the value `scheduled` to the second argument of the `open_channel_for_send/2` predicate. Since there is only one such channel in the system, only one philosopher process can ask assistance from the footman to sit at the table, and thus the latter can decide to grant permission depending on the number of guests at the table. Obviously, the `get_up_from_table/1` predicate behaves in a similar manner.

```
%%% Top level predicate for footman process
footman_top_level:- open_channel_for_receive(sit_down),
                    open_channel_for_receive(get_up),
                    footman(0),
                    close_channel(sit_down), close_channel(get_up).

%%% Main loop for the footman process
footman(0):- receive(sit_down,_N), footman(1).

footman(4):- receive(get_up,_N), footman(3).

footman(Philos):- N =< 3, N >= 1,
             receive([sit_down,get_up],_,WINNER),
             on_table(WINNER,Philos,NewPhilos),
             footman(NewPhilos).

%%% Defines the number of philosophers that sitting at the table
on_table(sit_down,Philos,NewPhilos):- NewPhilos is Philos + 1.

on_table(get_up,Philos,NewPhilos):- NewPhilos is Philos - 1.
```

Fig. 8. Dining Philosophers in Csp-ii : Footman process

Finally, figure 8 presents the Csp-ii code for the footman process. The argument of the `footman/1` predicate (main process loop) represents the philosophers that are currently sat on the table. Its operation is rather simple: when there are no philosophers sitting down (clause `footman(0)`), the process can only accept a message from the `sit_down` channel, i.e. a philosopher can only sit at the table, whereas when there four philosophers sitting down, it can only accept a message from the `get_up` channel (clause `footman(4)`), i.e. a philosopher can only leave the table. In any other case ($N \in [1, 2, 3]$) it can receive messages from any of the two channels, increasing or decreasing the number of philosophers on the table accordingly.

As a natural extension of the original inter-process channel concept, the external communication conceptually consists of message streams. In order to facilitate speed-up of external communication, asynchronous message passing is introduced as an option. The *send* operation in this case still remains blocking but the condition for continuing execution is the availability of sufficient buffer space instead of the commencement of the matching *receive* operation. For the Prolog programmer the communication environment appears as

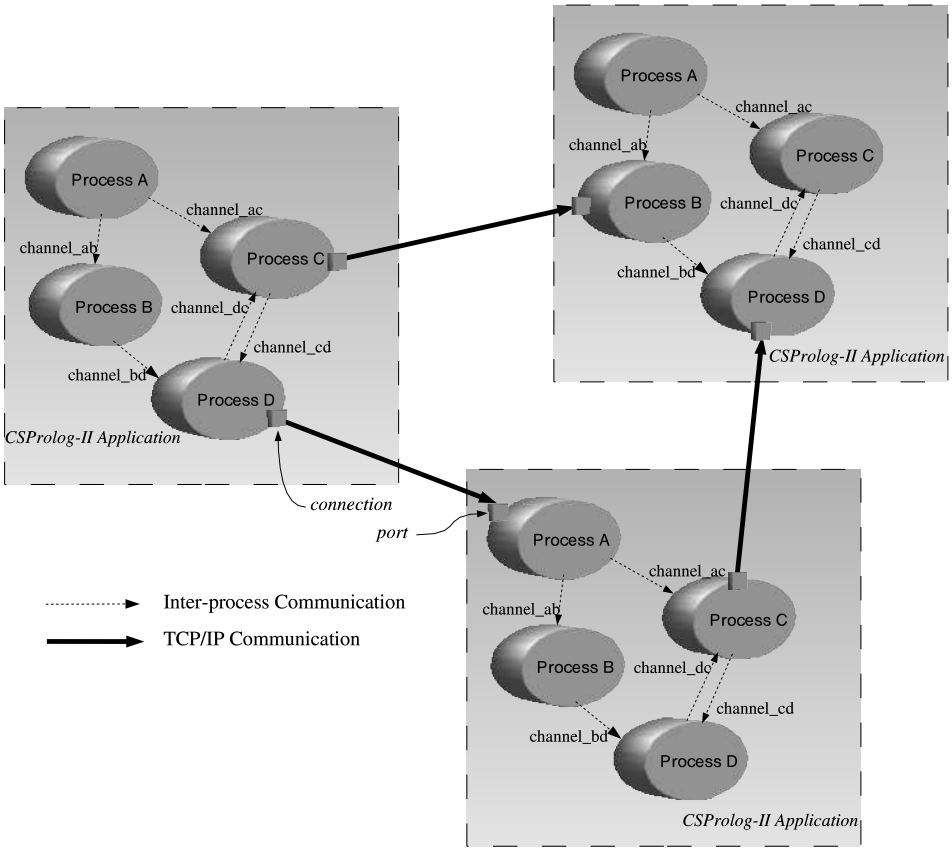

Fig. 9. A Csp-ii community

a homogeneous address space (community) in which all fellow applications (partners) are accessed via channel messages (Figure 9). The mechanism for connecting channels to other Csp-ii applications introduces two new notions: the port and the connection.

A *port* represents an incoming message substream. This entity should not be confused with the well known TCP/IP port. A Csp-ii port is the entry point of all incoming messages for the local application. It is explicitly created by a corresponding predicate and has a local channel associated with it at the time of its creation. The application receives all messages through that channel, as

shown in Figure 10. A parameter set during port creation determines the size of the message buffer so that asynchronous communication can occur.

A *connection* is the representation of an outgoing message stream. It is also explicitly created by the programmer and is associated with a partner's port to where it forwards all outgoing messages that it receives from a specific local channel of the sender application (Figure 10). All previous information is defined at the creation of the connection, including a parameter indicating the number of messages stored in the connection buffer. In order to be able to

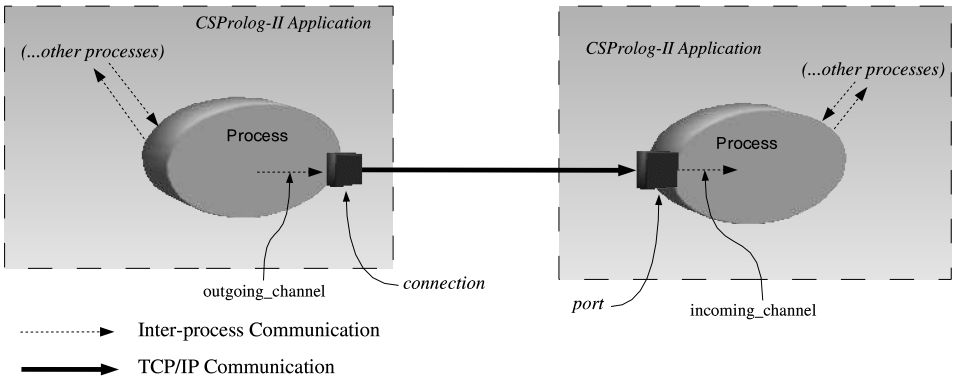

Fig. 10. Connection Port Mechanism in Csp-ii

communicate with a partner, a configuration process has to be performed using a special built-in predicate. Though this, all necessary network information of the partner is defined, i.e. its name, port, IP address or hostname, IP port it listens to, etc. Although this operation requires detailed knowledge of the partner's network information, it provides a more versatile connection schema. We are currently considering the idea to introduce some sort of naming service in a future version, however this will not require modifications of the current communication model, since it will be added in the form of a simple Prolog library.

An important feature of the Csp-ii TCP/IP communication is that changes in the status of a partner is immediately known to all other partners via the generation of specific network related events. Thus the user can define a real time process that captures all such events and adjusts the behavior of its application.

Applications can also establish communication with a non-Prolog application through an appropriate mediator, that handles all data and protocol conversions. Currently Csp-ii supports an ASCII mediator for plain text communication and one for communication with a specific network management platform (HNMS).

Csp-ii has been successfully employed in the development of a distributed expert system for the management of a TCP/IP based WAN [37,38].

4 CSPCONS: Extending the CSP-II Framework for Constraint Programming

Although CSP-II offered an adequate set of networking facilities for implementing distributed applications, it lacked programming facilities for CP, that would extend significantly its scope of applications. This fact was the main motivation behind extending appropriately the original system, an effort that resulted to the Communicating Sequential Prolog with Constraints (CSPCONS) system. The latter inherits all the advanced networking features, presented in section 3 as well as providing appropriate libraries for constraint programming.

There were two design choices toward this extension: either extend the system to support attributed variables that will enable the implementation of the constraint solver, or extend the C language interface appropriately and provide constraint solvers as C/C++ libraries. The latter approach was adopted for two reasons. The first concerns the fact that an excellent foreign language interface already existed in the CSP-II system, that facilitated greatly the development of the new solver interface, and the second is that under this schema the system could be easily integrated existing industrial strength constraint solvers available in the market. The latter plays a significant role in the adoption of CSPCONS in real world applications.

The rest of this section provides a brief overview of the extensions that were considered necessary for the development of the CSPCONS system.

4.1 Overview of the CSPCONS *System*

The CSPCONS system consists of two main subsystems: the *solver* and the *core*. The solver is responsible for maintaining the constraint store and performing any constraint related tasks, i.e. is responsible for storing domain variables and the set of constraints as well as for constraint propagation. The core is the extended CSP-II system that keeps track of the active instances of the different solvers, dispatches requests originated by the Prolog program to the appropriate solver instance, and performs other system-related tasks, including all normal Prolog predicate calls.

In general, each CSPCONS process can have active instances of several different solvers, as for example an FD and a Linear solver. However the set of constraints and domain variables maintained by instances of a solver that belong to different processes are independent of each other, resulting to a communicating sequential CLP system.

In order to support the above model, CSPCONS introduced to the original

CSP-II system a new set of built-in predicates, an appropriate C interface between the core and the solver and a new variable type, named constraint variable.

The CLP-related predicates that are defined in the new built-in predicate set are classified into three groups. The first group is concerned with the term type system extension, i.e. their use is the identification of constraint variables. The second group consists of the solver-independent predicates used for obtaining information about the installed solvers and selecting a particular solver. The third group consists of the "normal" interface predicates used for the introduction of new domain variables, constraints and for labeling. The predicates in the third group require cooperation between the core subsystem and the particular solver that is currently selected.

For achieving the solver-core cooperation a dedicated C language interface was implemented as an extension to the original CSP-II system. Under this schema, solvers are implemented in the form of linkable C libraries. Each solver exposes to the core a table containing pointers to specific functions (entry points). These entry points are mainly implementations of the normal interface predicates, i.e. a CLP related predicate call corresponds to an entry point. For example the `clp_constraint/1` predicate used to introduce new constraints in a program corresponds to the `constraint()` entry point function. However the implementation of the entry points depends on the use of a set of functions provided by the core, called *callback* functions, that provide various services such as constraint variable creation and removal, introduction of new trail points in the backtrack stack, etc.

Finally, constrained variables are introduced as a new term type in the original set of term-types. They are always associated with a corresponding internal variable of the solver. Their creation and removal is the responsibility of the solver, who requests it by appropriate callback function calls from the core. Upon unification of a constraint variable to a term in a Prolog program three cases can occur, depending on the state of the variable:

- If the unification involves a constraint and a normal unbound variable then it simply succeeds and the latter simply refers to the former in the computations that follow.
- If the variable is fixed to a specific value then unification is handled by the core. The solver in this case is called by a special entry point only to inform the solver about the status of the variable and its value if it is fixed.
- If the variable is being unified with another constraint variable or any other term then the unification is the responsibility of the solver who treats it as a newly introduced equality constraint. The solver in this case is called via an appropriate entry point and must either add the new constraint to the store if it is consistent or simply reject it, yielding a unification failure.

The solver subsystem is initialized when the first constraint predicate call is issued by the user program in the process. The solver instance starts with an empty system of constraints and during forward execution, new constraints are incrementally added to the model. The solver evaluates the resulting constraint set and if it is consistent, it accepts the additions and the call succeeds, otherwise rejects them, i.e. the call fails. If the predicate, which passes the new constraint succeeds, then all unbound variables occurring in the passed constraints become constrained variables and their behavior during unification is determined through a solver-core cooperation.

If the Prolog program backtracks over a CLP-predicate call or a unification of a constraint variable, the solver must revert to the state that was in effect before that call. Thus the state of the constraint store maintained by a solver instance must be synchronized with the state of the evaluation stack of the Prolog host process. Any change in the constraints store caused by the evaluation of a CLP-predicate or a unification involving constrained variables must be "undone" when the interpretation backtracks over the predicate that originated the change.

In the CSPCONS system there are two trail stacks: the core and the solver trail. The first is used by the Prolog interpreter itself for registering normal variable bindings that should be undone during backtracking, while the solver trail is used for registering changes in the constraint store. To achieve synchronization between these two areas the interface offers the ability to introduce identifiers of the solver trail to the core trail. On backtracking a special entry point function (`backtrack()`) is invoked and an identifier is passed back to the solver as argument to this function. The identifier indicates the appropriate stack level that the solver should backtrack to. Any necessary actions for restoring the state of the constraint store are organized based on this information.

The model offers independence of the code concerning the constraint handling and provides the means to easily extend the system to support any constraint domain. Currently CSPCONS supports a finite domain solver while there also exists an experimental linear equations-disequations solver. The Finite Domain solver is presented in more detail in the next section, since this constraint system has attracted more interest in the CP community mainly due to its numerous industrial applications.

The original version of the finite domain solver was based on the Macworth's AC-3 algorithm [39] for simplicity for handling binary constraints. Since the latter is not considered state of the art in consistency techniques, in the current version it was replaced by a constraint-based version of the AC-2000 proposed in [40]. Higher order constraints are handled by a bounds consistency algorithm, for efficiency reasons.

The FD solver supports constraints of the form: $x \in \{n_1, n_2, .., n_m\}$ and $exp_1\ R\ exp_2$ where $\{n_1, n_2, .., n_m\}$ is a set of natural numbers, $R \in \{=, \neq, <, >, \geq, \leq\}$ and exp_1, exp_2 are linear expressions on constraint variables. Constraints can be posted through the `clp_constraint/1` predicate as shown in the following examples:

```
...
clp_constraint([X in [1..10], Y in [1..10]]),
clp_constraint([3*X < 2*Y +10]),
...
```

In order to provide syntactic compatibility with well known CLP languages such as SICStus and ECLiPSe constraints can be defined in the using the infix operators $\#=, \#<, \#<=, \#>, \#>=$ and $::/2$, the latter for domain creation. As usual the solver provides a library of predicates for labeling based on heuristics, optimization, etc. For example, the code that follows defines a very simple problem of four variables and one equation on them:

```
...
X :: [1..10], Y :: [1,2,4],
Z :: [1..30], W :: [1,2,3..10],
X+3*Y#=Z-W, labeling([X,Y,Z,W]),
...
```

The implementation has been tested on a variety of benchmarks, including the well-known cryptarithmentic and alpha problems, golomb rulers, N-Queens, etc. and has performed adequately. It should be noted however the system performance cannot be compared with systems such as ECLiPSe or SICStus that employ far more sophisticated constraint handling algorithms. The advantage of the CSPCONS system is that it offers an excellent platform to model and test distributed constraint based applications, as it is going to be demonstrated in the next section.

5 Distributed Job-shop Scheduling in CSPCONS

Job-shop scheduling (JSS) is one of the most popular scheduling problems and has attracted significant attention in the past decades. It has been long used as a benchmark for the evaluation of numerous and diverse optimization techniques.

More formally, the $n \times m$ job-shop problem involves n jobs $J_1, J_2, \ldots J_n$ and m machines or processors $M_1, M_2, \ldots M_m$. Each job $J_k, 1 \leq k \leq n$ consists of l uninterrupted operations $O_{k1}, O_{k2}, \ldots O_{kl}$, upon which a precedence constraint holds, i.e. they have to be scheduled in a predetermined order (O_{k1} before O_{k2}, O_{k2} before O_{k3}, etc). Each operation O_{ki} has a non-negative duration $d(O_{ki})$ and requires processing by a specific machine for its completion. The machines obey a capacity constraint limiting the operations they can simultaneously process to one. The problem's objective is to find the start time of each operation $start(O_{ki})$, such that all constraints are satisfied and the *makespan* of the total schedule is minimum. The latter is defined as the maximum completion time of all jobs, i.e:

$$makespan = max(start(O_{kl}) + d(O_{kl})), 0 \leq k \leq n$$

A great number of techniques, such as constraint programming, genetic algorithms, simulated annealing, local search, etc have been employed to tackle the problem yielding different results in terms of time or solution quality. A review of the techniques employed to solve the JSS problem can be found in [41].

The aim of this section is not to provide a novel approach to the solution of the JSS problem, comparable to the already proposed techniques, but to demonstrate the the suitability of the CSPCONS system in rapid prototyping and testing of distributed algorithms. In that direction, the rest of this section presents details of two distributed algorithms for solving (sub-optimally) the problem. The two algorithms differ in the way agents cooperate for available time slots to schedule their operations.

5.1 Agent Based Version

In the first algorithm, we have chosen to model the JSS problem as a collection of cooperating agents, each responsible for a job in the problem, that negotiate on the use of resources (Agent Based Version - ABV). The main idea behind the above modeling is rather simple: agents enforce precedence constraints on the operations they are responsible, generate proposals on machine utilization and then resolve between them potential conflicts on the use of resources.

For each Operation O_i **in** Job **do**
1. Generate a proposed booking for O_i on machine m
2. Broadcast booking to other agents.

while there is not a final booking for O_i **do**

3. Collect bookings (proposed+final) from other agents on machine m.
4. Insert any constraints derived from final bookings

on the machine.

5. Based on the received proposed bookings decide whether to commit to a start time for O_i or to generate a new proposed booking.
   taking into account the constraints derived from the new bookings.
6. Broadcast the new booking.

endWhile

endFor

Fig. 11. Agent Scheduling Loop

Under the above schema, all agents impose precedence constraints on the operations of their jobs, using the CSPCONS constraint primitives. Intra-agent conflicts, i.e. overlapping requests for scheduling an operation on the same machine are resolved by message exchange and a dynamic priority schema that is based on the current minimum makespan of the agents' jobs.

Agents communicate by exchanging *booking* messages (or simply *bookings*), i.e. allocation of an operation to a machine at a specific time slot. Bookings contain information about a specific operation, such as agent and operation IDs, start time, duration, machine required, status etc. and are classified according to their status to proposed and final. *Proposed* bookings represent requests for scheduling an operation on a specific machine starting at the time indicated by the booking. *Final* bookings on the other hand indicate commitment, i.e that the corresponding operation is scheduled as indicated. Bookings are broadcasted to all agents in the application, thus each agent is aware of the proposals and commitments of all other agents.

The execution cycle of each agent is shown in Figure 11. Each agent generates initially a proposed booking indicating to all agents in the application not only its desired minimum start time and machine for operation O_i but also the minimum makespan for the job if operation O_i is scheduled as requested. In the step that follows it collects all available information (bookings) on the utilization of the specific machine. Final bookings on the machine are used to produce new constraints, i.e. the unscheduled operation O_i has to be performed after the end of any operations already booked. Proposed bookings are used to decide whether to commit to a start time or simply generate a new proposal. This decision is based on a dynamic priority decision strategy: priority is given to operations with the maximum job makespan, reflecting the simple (polite) rule "grant priority to agents (jobs) that end later". Thus the agent that has the greater makespan has priority over all other agents

and generates a final booking, that it broadcasts to the community. All other agents simply take into account existing bookings and proceed by generating a new proposal. The application terminates when all agents have found valid schedules for their operations.

5.1.1 Implementation

A straightforward approach in implementing the algorithm would be to model each agent as a *normal* CSPCONS process, that communicates over channels with other agents. However, intra-process channel based communication is synchronous, which is prone to deadlocks, thus requires a more sophisticated design. To avoid such situations we had to introduce asynchronous communication between processes, by simply adding a real-time process to each agent, as shown in figure 12. The latter is responsible for handling all communications to and from the agent. The introduction of the real-time communication

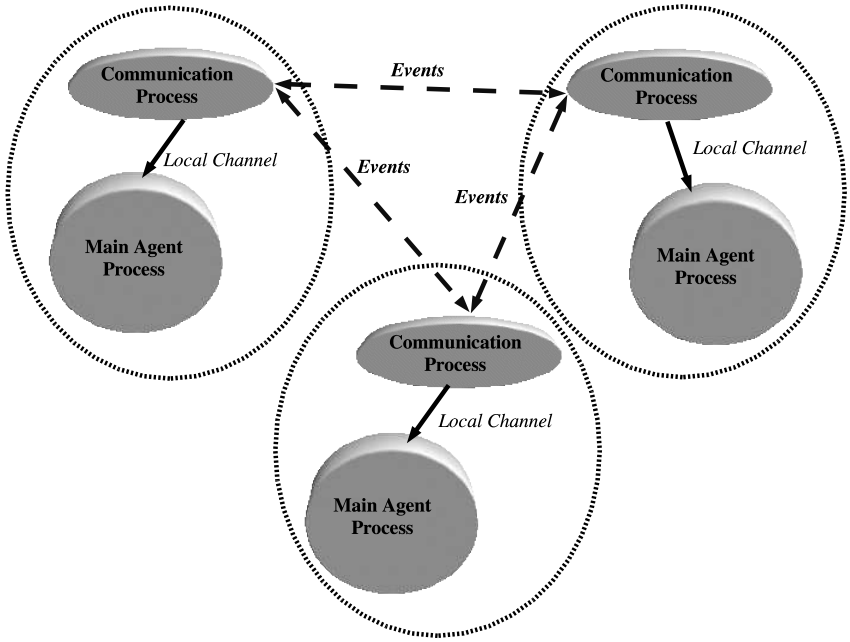

Fig. 12. Agents in the Job-shop scheduling Application

process facilitated greatly the implementation of the algorithm. Under this new schema, agents communicate by generating events that carry booking information. The role of communication process is twofold: a) it is responsible for accepting all agent bookings and broadcasting locally generated bookings and b) maintaining all machine utilization information based on the received booking messages. The main agent process that is responsible for scheduling the job, simply collects from its corresponding communication process, all necessary bookings using channel based communication. The latter is again initiated by appropriate event generation, this time by the main agent process.

In the second version, the utilization of each machine is controlled by a machine agent (Machine Based Version - MBV). In this case job agents do not exchange bookings, but instead they submit requests for scheduling an operation to the corresponding machine agent (figure 13). The latter collects all pending requests at the specific time, decides upon winning operations, i.e. operations to be scheduled next and informs the interested agents. Non-winning agents are just informed that they have to reschedule their operations after the end of the winner operations and of course to re-submit a booking request taking into account the new constraint.

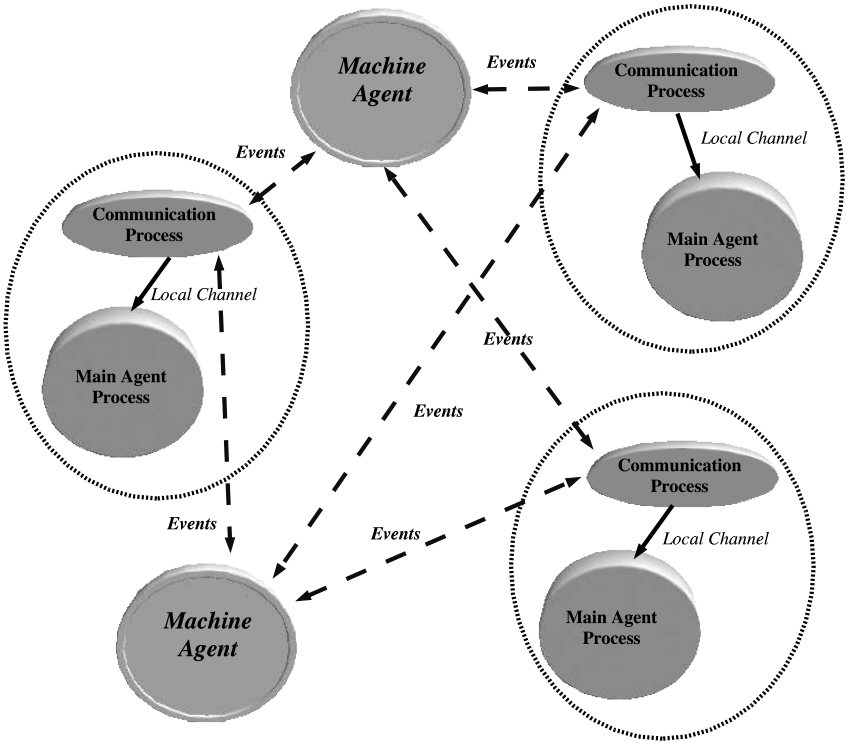

Fig. 13. Job and Machine Agents in the Distributed JSS Application

Machine agents maintain information about machine utilization on which base their decisions. An improvement concerning the first version is that the machine agent initially tries to schedule any request in available idle intervals (i.e. gaps in the schedule) of the machine. From the set of requests that cannot be satisfied in the previous step, a winner operation is selected, using the same rule described in section 5.1: "grant priority to agents jobs that end later". One interesting aspect of the second version is that job agents are unaware of the overall schedule, thus all job information remains private.

5.2.1 Implementation

The machine agent is modeled as a real-time process that communicates with job agents via events, that carry all necessary information (figure 13). A point worth mentioning is when does the agent decide that it has received all currently pending request messages concerning its resource utilization. To avoid complex algorithms, the agent simply assumes that it has received all pending requests when there are no more messages over a certain period of time. The length of this time period affects the overall performance of the algorithm and possibly the solution quality, i.e. waiting for longer intervals introduces significant delays but allows to process more requests at a time. However, it does not affect its soundness since all requests will eventually be processed.

This simple approach is elegantly implemented by *timer events*, that "wake" the process up when no other event has occurred for a certain amount of time (see Section 3.1). To demonstrate the simplicity of implementing such behavior in CSPCONS, the top level predicate of the machine agent is shown in Figure 14. In that the `get_event/2` built-in collects any events from the queue of the process.

```
%% Updates the internal DB about job agent requests.
%% Message interception.
machine_agent(N):-
   get_event(machine(N),BOOKING),
   updateDB(BOOKING).

%% No more messages to receive so try to do something usefull.
machine_agent(_):-
   get_event(idle(_)),
   existRequests, % check if there are any requests pending.
   answerRequests.

%% At idle times, where there is nothing to answer do nothing.
machine_agent(_):-
   get_event(idle(_)).

%% Stop event received - Terminating Execution.
machine_agent(N):-
  get_event(endM(N)),
  !,fail.
```

Fig. 14. Top-level Predicate for Machine Agent

The implementation of job agents does not differ significantly from those in the ABV version. The only change is that communication is limited to the machine agents and that the decisions about the operation allocation is delegated to the corresponding machine agent.

We have tested the implementation on some well known instances of the job-shop scheduling problem, namely the Fisher - Thomson 6x6, 10x10 and 20x5 [42], also known as mt06, mt10, mt20.

The results indicated in table 1 present the makespan (in time units) and the time consumption of the schedules produced by the two versions of our algorithms compared to the best known solutions. Column BKS presents the makespan of the best known solution reported in the literature, where as columns ABV and MBV present the makespan of the Agent Based Version and the Machine Based Version respectively. For each problem the average execution time is listed in parentheses in seconds. The table also provides details about the number of jobs and machines in each of the problems.

In all cases the solution obtained by the algorithms were not optimal, however this was not our aim. The algorithms could be extended in a number of ways to achieve an optimal solution; for example a local search algorithm could be applied after the first solution to increase the quality of the latter, search could be re-initiated with an upper bound on the solution quality, etc. The difference in the solution quality between the Agent Based version and the Machine based version is due to the fact that the latter tries to schedule operations in available idle intervals of the machine.

Table 1
Makespan of the schedules produced by the two approaches (in time units). The table also lists the average execution time of the algorithms for each problem.

Problem	Jobs	Machines	BKS	ABV	MBV
mt06	6	6	55	76 (2.2 secs)	60 (1.7 secs)
mt10	10	10	930	1245 (410 secs)	1186 (390 secs)
mt20	20	5	1165	1675 (930 secs)	1670 (870 secs)

The time consumption of the algorithm is rather high when compared with local search techniques, or dedicated constraints (such as the cumulative constraint) used in current CLP systems. However, providing a time efficient solution to the problem was not among the aims of this work; the algorithm is rather naive and its presentation servers only as an example of how distributed CLP applications can be implemented easily given the programming facilities of Csp-ii . Multiple approaches can be followed to improve this aspect of the algorithm. For example a combination of constraint solving and local search similar to the one reported in [43] could improve significantly the efficiency and the solution quality of the algorithm. However, the execution time of the current algorithm could be reduced by providing a version that runs on multiple hosts as it will be shown in the following section.

To stress the suitability of the CSPCONS platform for developing and testing such applications it should be noted that the complete programs of both versions are no more than 400 lines of Prolog code each.

5.4 Distributed Machine Based Version

The observant reader would have noticed that all the agents in the two versions of the problem are processes of a single CSPCONS application running on a single host. To test futher the platform and possible gains in efficiency obtained by a parallel/distributed version of this simple program we have developed a multi-host version of the machine based version of the algorithm and run a number of experiments.

Deriving the multi-host distributed version from the version presented above was straightforward, since it simply required minor modifications to the communication processes so that instead of generating an event, they simply broadcast a message over an established TCP/IP channel. Such a modification, did not affect the implementation of the agents themselves; it simply involved adding a few necessary predicates to the real-time processes. This presents another advantage of the CSPCONS platform: distributed algorithms can be developed and tested in a single host and latter easily ported to a network environment.

In the distributed implementation each agent and machine is an independent CSP-II application, thus, depending on the problem there are 12, 20 and 25 agents running in parallel to solve the problem. All communication takes place over TCP/IP channels. We have tested the new version on a network of four hosts each running a number of CSP-II process. In our experiments the processes were evenly distributed to the hosts and all measurements concern wall time. Our experiments involved running the same problem on up to four hosts with varying the delay (timeout) that each machine agent waits for collecting messages (see section 5.2) from 10 to 200 msecs. The latter provides an evaluation of the impact of this parameter to the total execution time of the algorithm.

The results obtained for the ft06 problem are presented in figure 15. As shown from the graph, the execution time of the algorithm does not improve greatly as the number processors increases, mainly due to the reason that network delays introduced by the TCP/IP message passing introduce significant overhead for the specific problem. An important observation here is that the algorithm's performance is greatly influenced by the timeout, since the later is comparable to the execution time of the algorithm.

A similar set of experiments was conducted for the ft10 problem. The results

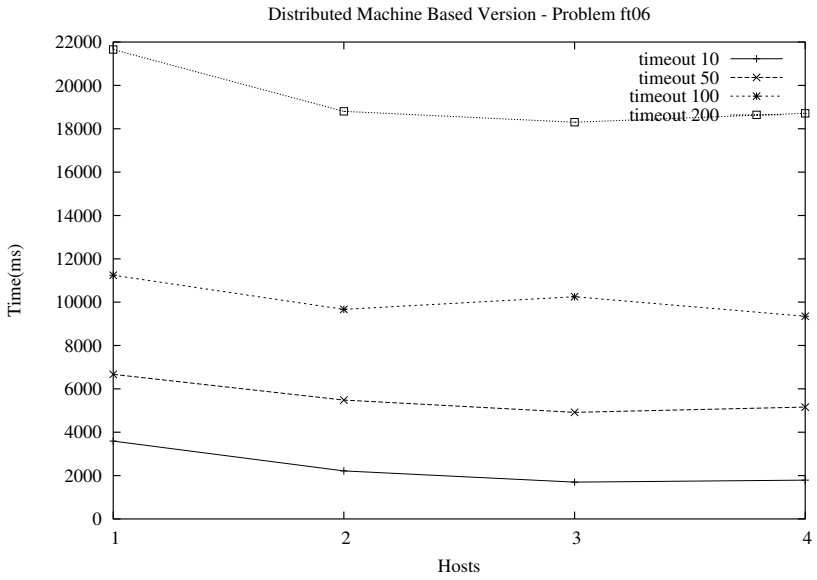

Fig. 15. Experimental results for the ft06 Problem.

are shown in figure 16. As it can be seen from the graph in this case the speedup obtained is substantial and the algorithm has a good scale up. In this problem, the timeout parameter does not have a significant impact on the overall performance, since the timeout is far less than the execution time of the distributed system. Similar to this results were obtained for the ft20 problem.

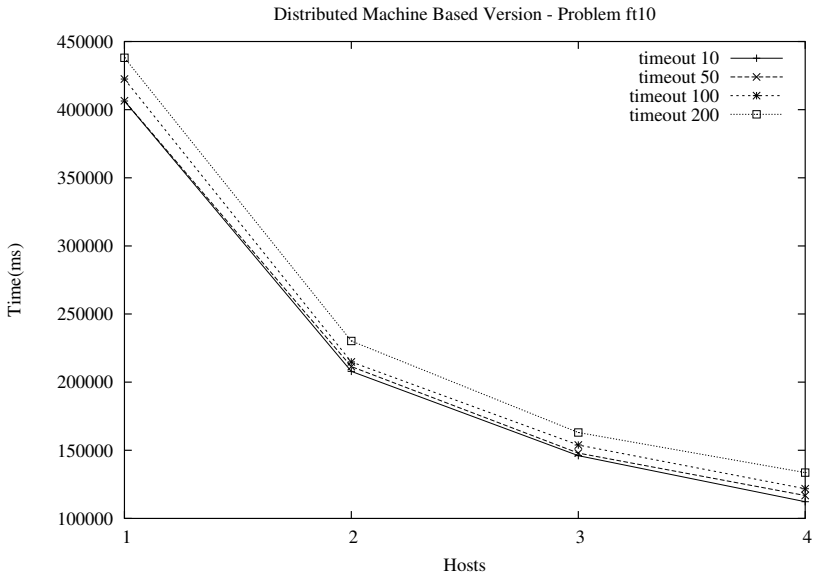

Fig. 16. Experimental results for the ft10 Problem.

6 Conclusions and Future Work

The CSPCONS language, presented in this paper, offers a suitable platform for rapid prototyping, testing and developing of any DCSP application. Programming through the use of communicating sequential processes and constraints in a logic programming environment can successfully address the issues of easily developing applications that require agent based program distribution and communication. As demonstrated in the job-shop scheduling examples, in such an application each agent can be a collection of independent CSPCONS processes that exchanges messages with other agents in order to achieve a global consistency.

Different constraint solvers can be added in the form of C (or C++) linkable libraries. This results to an extensible language that can be tailored to the application requirements and also a great platform for testing new constraint algorithms.

We are currently investigating the implementation of some DCSP algorithms as for example those reported in [26,27] and in [30]. Such implementation might require both further development of the constraint solver or the introduction of new programming facilities. One of the main issues that has to be addressed is the definition of necessary components, which would allow the programmer to exploit these algorithms in an application without having to implement them again. Our ambition is to develop a framework that will relieve the programmer of the burden to explicitly encode all the above and just concentrate on the program development.

Possible application areas of the CSPCONS system include distributed planning and scheduling. Our immediate plans also include the development of a distributed scheduling application for university course scheduling, that will fully test the potential of the current implementation of the language.

References

[1] C. A. R. Hoare, Communicating Sequential Processes, Communications of the ACM 21 (8) (1978) 666–677.

[2] I. P. Vlahavas, I. Sakellariou, I. Futo, Z. Pasztor, J. Szeredi, CSPCONS: A Communicating Sequential Prolog with constraints, in: Methods and Applications of Artificial Intelligence, Procs of the 2nd Hellenic Conference on AI, SETN 2002, Vol. 2308 of Lecture Notes in Computer Science, Springer, 2002, pp. 72–84.

[3] M. Hermenegildo, F. Bueno, D. Cabeza, M. G. de la Banda, P. Lopez, G. Puebla, The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems, in: I. de Castro Dutra, M. Carro, V. S. Costa, G. Gupta, E. Pontelli, F. Silva (Eds.), Parallelism and Implementation of Logic and Constraint Logic Programming, Nova Science, 1999, pp. 65–85.

[4] D. Cabeza, M.Hermenegildo, Distributed Concurrent Constraint Execution in the CIAO System, in: Proceedings of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies, U. Utrecht / T.U. Madrid, 1995.

[5] M. Hermenegildo, D. Cabeza, M. Carro, Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems, in: L. Sterling (Ed.), Proceedings of the 12th International Conference on Logic Programming, MIT Press, Cambridge, 1995, pp. 631–646.

[6] B.-M. Tong, H.-F. Leung, Data-parallel concurrent constraint programming, The Journal of Logic Programming 35 (1998) 103–150.

[7] P. Van Roy, S. Haridi, Mozart: A programming system for agent applications, in: International Workshop on Distributed and Internet Programming with Logic and Constraint Languages, 1999, part of International Conference on Logic Programming (ICLP 99).

[8] S. Haridi, P. V. Roy, P. Brand, M. Mehl, R. Scheidhauer, G. Smolka, Efficient logic variables for distributed computing, ACM Transactions on Programming Languages and Systems 21 (3) (1999) 569–626.

[9] S. Haridi, P. Van Roy, G. Smolka, An overview of the design of Distributed Oz, in: Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97), ACM Press, Maui, Hawaii, USA, 1997, pp. 176–187.

[10] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, C. Schulte, Logic programming in the context of multiparadigm programming: the Oz experience, Theory and Practice of Logic Programming To appear.

[11] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, M. G. Wallace, ECLiPSe: An introduction, Tech. Rep. IC-Parc-03-1, IC-Parc, Imperial College London (2003).

[12] H. Simonis, Developing applications with ECLiPSe, Tech. Rep. IC-Parc-03-2, IC-Parc, Imperial College London (2003).

[13] M. Carlsson, Finite domain constraints in sicstus prolog, CICLOPS Workshop at CP'2001, Int. Conf. on Principles and Practice of Constraint Programming, 2001. Invited talk.

[14] M. Carlsson, G. Ottosson, B. Carlson, An open-ended finite domain constraint solver, in: Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97), Vol. 1292, Springer-Verlag, Berlin, 1997, pp. 191–206.

[15] J. Eskilson, M. Carlsson, SICStus MT — A multithreaded execution environment for SICStus prolog, Lecture Notes in Computer Science 1490 (1998) 36–53.

[16] P. R. Cooper, M. J. Swain, Arc consistency: Parallelism and domain dependence, Artificial Intelligence 58 (1–3) (1992) 207–235.

[17] Y. Zhang, A. K. Mackworth, Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments, Tech. Rep. TR-92-30, Department of Computer Science, University of British Columbia (Nov. 1992).

[18] T. Nguyen, Y. Deville, A distributed arc-consistency algorithm, Science of Computer Programming 30 (1–2) (1998) 227–250, concurrent constraint programming (Venice, 1995).

[19] C. Bessière, Arc-consistency and arc-consistency again, Artificial Intelligence 65 (1) (1994) 179–190.

[20] Y. Hamadi, Optimal distributed arc-consistency, Constraints 7 (3) (2002) 367–385.

[21] K. Apt, From Chaotic Iteration to Constraint Propagation, in: Proceedings of 24th International Colloquium on Automata, Languages and Programming, ICALP'97, Vol. 1256, Springer-Verlag, 1997, pp. 36–55.

[22] E. Monfroy, J.-H. Réty, Chaotic iteration for distributed constraint propagation, in: Proceedings of The 14th ACM Symposium on Applied Computing, SAC'99, Artificial Intelligence and Computational Logic Track, San Antonio, Texas, USA, 1999, pp. 19–24.

[23] E. Monfroy, Control-driven constraint propagation, Applied Artificial Intelligence 15 (1) (2001) 79–103.

[24] I. Sakellariou, I. Vlahavas, Distributed singleton consistency, Journal of Experimental and Theoretical Artificial Intelligence 16 (2) (2004) 107–124.

[25] M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara, The Distributed Constraint Satisfaction Problem: Formalization and Algorithms, IEEE Trans. on Knowledge and Data Engineering 10 (5) (1998) 673–685.

[26] M. Yokoo, K. Hirayama, Algorithms for Distributed Constraint Satisfaction: A Review, Autonomous Agents and Multi-Agent Systems 3 (2) (2000) 185–207.

[27] Y. Hamadi, C. Bessière, J. Quinqueton, Backtracking in Distributed Constraint Networks, in: H. Prade (Ed.), Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98), John Wiley & Sons, Chichester, 1998, pp. 219–223.

[28] P. Meseguer, M. A. Jimenez, Distributed forward checking, in: Proceedings of the CP 2000 International Workshop on Distributed Constraint Satisfaction, Singapore, 2000.

[29] C. Bessière, A. Maestre, P. Meseguer, Distributed dynamic backtracking, in: M. Silaghi (Ed.), Proceedings of the IJCAI'01 Workshop on Distributed Constraint Reasoning, Seattle WA, 2001, pp. 9–16.

[30] M. Silaghi, D. Sam-Haroud, B. Faltings, Maintaining hierachical distributed consistency, in: EPFL (Ed.), Proceedings of the CP2000 Workshop on Distributed Constraint Satisfaction, Tech. Report # 00/338, 2000.

[31] M. C. Silaghi, D. Sam-Haroud, B. Faltings, Asynchronous Search with Aggregations, in: Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00), AAAI Press, Menlo Park, CA, 2000, pp. 917–922.

[32] I. Futo, Prolog with Communicating Processes: From T-Prolog to CSR-Prolog, in: D. Warren (Ed.), Proceedings of the 10th International Conference on Logic Programming, The MIT Press, 1993, pp. 3–17.

[33] I. Futo, A Distributed Network Prolog System, in: Proceedings of the 20th International Conference on Information Technology Interfaces, ITI 99, 1998, pp. 613–618.

[34] D. H. D. Warren, An abstract Prolog instruction set, Tech. Rep. 309, SRI (1983).

[35] E. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971) 115–138.

[36] C. A. R. Hoare, Communicating Sequential Processes, Prentice/Hall International, 1985.

[37] I. Vlahavas, N. Bassiliades, I. Sakellariou, M. Molina, S. Ossowski, I. Futo, Z. Pasztor, J. Szeredi, I. Velbitskiy, S. Yershov, S. Golub, I. Netesin, System Architecture of a Distributed Expert System for the Management of a National Data Network, in: F. Giunchiglia (Ed.), Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA-98), Vol. 1480 of LNAI, Springer, Berlin, 1998, pp. 438–451.

[38] I. Vlahavas, N. Bassiliades, I. Sakellariou, M. Molina, S. Ossowskia, I. Futo, J. S. Zoltan Pasztor, I. Velbitskiyi, S. Yershov, I. Netesin, ExperNet: An Intelligent Multi-Agent System for WAN Management, IEEE Intelligent Systems 17 (1) (2002) 62–72.

[39] A. K. Mackworth, Consistency in Networks of Relations, Artificial Intelligence 8 (1) (1977) 99–118.

[40] C. Bessière, J.-C. Régin, Refining the Basic Constraint Propagation Algorithm, in: B. Nebel (Ed.), Proceeding of IJCAI-01, 17th International Joint Conference on Artificial Intelligence, Seattle, US, 2001, pp. 309–315.

[41] A. Jain, S. Meeran, A state-of-the-art review of job-shop scheduling techniques, Tech. rep., Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, Dundee, Scotland (1998).

[42] H. Fisher, G.L.Thompson, Probabilistic learning combinations of local job-shop scheduling rules, in: J. Muth, G. Thomson (Eds.), Industrial Scheduling, Prentic Hall, Englewood Cliffs, 1963, pp. 225–251.

[43] K. Chatzikokolakis, G. Boukeas, P. Stamatopoulos, Construction and repair: A hybrid approach to search in csps, in: Proceedings of the 3rd Hellenic Conference on Artificial Intelligence SETN-2004, Samos, Vol. 3025 of LNAI, 2004, pp. 342–351.