CSPCONS: A Communicating Sequential Prolog with Constraints

* Ioannis Vlahavas[1], Ilias Sakellariou[1], Ivan Futo[2], Zoltan Pasztor[2], and Janos Szeredi[2]

[1] Department of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki Greece
{vlahavas, iliass}@csd.auth.gr
[2] ML Consulting and Computing Ltd, ML Kft, H-1011 Budapest, Gyorskocsi u. 5-7., Hungary.
{futo, pasztor, szeredi}@ml-cons.hu

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, among other features, channel-based communicating processes and TCP/IP communication and 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. Currently CSPCONS offers a finite domain constraint solver, but the addition of new solvers is supported as they can be integrated in the system in the form of linkable C libraries. This paper briefly describes the original CSP-II system along with the extensions that resulted to the CSPCONS system.

1 Introduction

In the past decade, constraint programming has proven to be a suitable platform for tackling large combinatorial problems with significant applications in industry, like scheduling, resource allocation, etc. However even with the most advanced techniques, solving such problems is both space and time costly.

The introduction of new sophisticated sequential algorithms for constraint satisfaction is one way to overcome the problem. However the availability of a large number of machines connected by some local network, naturally led to the approach of distributing the problem to multiple processing units, often called agents or workers, that cooperate to solve the problem more efficiently.

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

CSPCONS is a logic programming language for building such systems. The language 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. CSPCONS supports independent CLP processes each having its own constraint store that communicate through message exchange over channels. Communication is possible both between processes that reside in the same host and on different hosts over TCP/IP networks. Constraint facilities in CSPCONS are implemented as C libraries, thus permitting the incorporation of new constraint just by the addition of the appropriate library. The current version includes a library for constraint satisfaction over finite domains (FD).

The combination of the channel based communication and constraint satisfaction, all under the logic programming framework, offers a powerful platform for the rapid implementation of any distributed CSP application.

This 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 its features 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. Section 5 shows an example of a distributed implementation of the N-queens problem along with some experimental results. Finally conclusions and future work are stated in section 6.

2 Distributed Constraint Satisfaction Problems

Informally, 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. Constraints are imposed on a subset of the domain variables and restrict the values which can be simultaneously assigned to them.

A distributed constraint satisfaction problem is a CSP in which the variables/constraints are distributed over some network of agents. Agents are constraint solvers which co-operate to solve the original problem. The need for distributed constraint programming applications derives mainly from two facts: a) more efficient implementations, in terms of execution time, can be achieved by decomposing the original problem into subproblems and b) representing problems that are naturally distributed is significantly facilitated, as for example production planning in a factory in which independent departments must meet their local constraints and at the same time co-operate to achieve global constraints.

A number of approaches have been reported to the literature that address the issue of building distributed constraint programming applications. In the sequel we will restrict our presentation to systems that belong to the logic programming framework and also present some algorithms proposed.

The approach followed for the implementation of the distributed capabilities of the CIAO language[5] is described in [1]. CIAO is a system based in Prolog extended with constraints, parallelism and concurrency. The distributed execution capabilities are based on the Linda library for implementing communication between processing units (referred to as workers), i.e. it adopts a blackboard architecture and the use of attributed variables[6].

A different approach to solving CSP problems in parallel has been proposed by Tong and Leung in [12]. 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.

Apart from the above systems a number of algorithms have been proposed that address the issue of distributed constraint satisfaction. A class of such algorithms performs distributed arc consistency, as for example a distributed version of the AC4 algorithm, based on an message passing communication model [9]. In [16] Zhang and Mackworth present parallel and distributed algorithms for computing consistency by formulating a CSP as a *dual network*, in which constraints correspond to nodes and variables to arcs. These algorithms were tested on a transputer based machine.

In [14, 15] authors propose an asynchronous backtracking algorithm and its modification, the asynchronous weak-commitment search, that efficiently solves distributed constraint satisfaction problems. In the proposed algorithm a problem variable is assigned to each agent who instantiates it and communicates its value through messages to other agents. Upon the detection of an inconsistency, agents exchange appropriate messages in order to backtrack and achieve a consistent assignment of values.

In the distributed backtracking algorithm (DIBT) introduced in [4], a different approach is followed. Agents compute their position in a total ordering of the network, each having a set of parent and child agents. Upon variable instantiation the agent's children are informed of the chosen value and failure to determine a consistent value in this set initiates backtracking to the parent agents. The algorithm employs message passing communication.

Finally, an algorithm that integrates distributed consistency techniques into asynchronous backtracking is presented in [11]. The proposed algorithm combines a distributed bounds consistency algorithm, called DHC, with a distributed search technique called Asynchronous Aggregation Search [10]. Agents communicate information by message exchange as in the previous algorithm.

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.

3 The Csp-ii Prolog

The Csp-ii distributed Prolog system is being developed since 1995 [2],[3]. The syntax and the built-in procedures of the language follow those of the standard Prolog (ISO/IEC 13211-1); furthermore the language is extended with features like modularity, multitasking, real-time programming and network communication.

The main feature of the Csp-ii system is that it supports the communicating sequential process [7] programming methodology in a Prolog environment. Processes run in parallel and communication between them is achieved through message passing over channels. This process-based model allows the implementation of parallel and distributed algorithms.

The channel-based communication has been extended with networking capabilities over the TCP/IP protocol, thus 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 containing the WAM code, although in some points different. This 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

Csp-ii processes are defined as the execution flow of a Prolog goal and every process has its own Prolog execution environment and dynamic database. Thus the progress of a process is independent of the execution of other processes. This separation of dynamic databases ensures that Csp-ii processes may have influence on each other only by the Csp-ii provided communication techniques, i.e. channels, events and interrupts, or through external objects like files. On a single processor machine a time-sharing scheduler controls the concurrent processes.

Processes are identified by a unique system-wide symbolic name. Two kinds of processes are provided:

- self-driven or normal processes, which is the most usual kind.
- 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. At the moment of its termination the process disappears from the Csp-ii system and will never reappear.

A *real time* process is characterized by one 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 the successful termination of the initializing goal 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. The 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.

Inter-process communication is achieved by synchronous messages or by event passing. 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. 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.

As stated *events* serve for triggering real time processes and are also identified by system-wide unique names. They can be generated explicitly by built-in predicates 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. The number of the available events in a program is unlimited. It should be noted that every occurrence of an event may have an optional data argument that can be used to provide some additional information. The event data is an arbitrary Prolog term, except the case of a single unbound variable.

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

3.2 TCP/IP Communication

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 a homogeneous address space (community) in which all fellow applications (partners) are accessed via channel messages. A separate mechanism is introduced for connecting channels to other Csp-ii applications. Two notions are introduced in this mechanism: the port and the connection.

A *port* represents an incoming message substream. This entity should not be confused with the normal 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 a local channel is associated with it at the time of its creation. The application receives all messages through that channel. A parameter set during port creation determines the size of the message buffer so that asynchronous communication can take place.

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. 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 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.

A Csp-ii application 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 [13].

4 Extending the Csp-ii Framework for Constraint Programming

Cspcons is an extension of the Csp-ii system that inherits all its advanced features and at the same time provides constraint solving capabilities.

The 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. It should be noted that several solvers are allowed to each program. 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, called constraint variable.

The CLP-related predicates that are defined in the new built-in predicate set can be divided 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. This cooperation is achieved through a dedicated for the purpose C language interface.

Solvers are implemented in the form of linkable C libraries. Each solver must expose for 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.

4.1 The CSPCONS Execution Model

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. 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.

4.2 The Finite Domain Solver

Since our main aim was to test the ideas and the extension model, the implementation of the FD solver had to be kept as simple as possible. Thus the solver was based on the AC-3 [8] algorithm. Although the latter is not considered state of the art, it was selected due to its simplicity.

Currently the 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. All constraints are posted through the `clp_constaint/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]),
```

All unary and binary constraints are handled internally by the consistency algorithm. Higher arity constraints are delayed until they become ground and are then handled as unary constraints. The solver also provides a set of predicates for labeling including one that uses the fail-first principle.

As mentioned in a previous paragraph backtracking involves synchronizing the solver and the core trail. The solver trail stack contains entries that belong to four types. Two of them concern variable creation and constraint addition, and the third type concerns value removal, while the last type records constraint variable unification with an integer. Upon value removal only a pointer to the specific value is recorded. This pointer is sufficient for restoring the value since what is required is flipping the `valid` field of the structure that stores the value. Each trail entry has an identifier associated with the core trail entry according to the extension model described above. Multiple solver trail entries can share the same identifier value since they belong to the same choice point and thus the core trail is not overtaxed with entries.

It should be noted that the implementation has been tested on a variety of benchmarks, including the well-known cryptarithmentic and alpha problems and some artificial ones and has performed adequately. However the system performance cannot be compared with systems such as ECLIPSE or SICStus that employ far more sophisticated constraint handling algorithms.

5 Solving the N-Queens Problem

To show the suitability of the proposed system for the implementation of any distributed CSP program, we have implemented a single process and two multi-process versions of the well-known N-Queens problem. The single process version is in fact the standard implementation of the problem but without using the first-fail principle.

The multi-process versions consist of two independent processes each having its own store. Both versions divide the problem of N Queens in half, assigning N/2 Queens to each process. On each subset of these variables local constraints are applied, stating the relations between N/2 Queens in a (N/2)xN chessboard. A priority is set between the two processes having one of them assigning values first and passing them via a channel to the second process. Messages are passed via inter-process channels, since the program is executed in the same host, however the implementation of TCP/IP communication between processes of different hosts is straightforward.

The two versions implement different search algorithms between the two independent processes. The first version employs synchronous backtracking (SB), as that is described in [14], to solve the problem. Under the synchronous backtracking algorithm the first process instantiates its variables to consistent values according to the local constraints and communicates them to the second process. The latter upon reception of this partial solution, introduces to the store new constraints based on the set of values received and searches for a solution. If such a solution is found then the program terminates with success otherwise a

backtrack message is passed back to the first process. The above loop continues until a solution is found.

The second version is an enhancement of the synchronous backtracking algorithm (ESB). In this algorithm the sender process communicates the value of a variable as soon as it is instantiated, i.e. at each step of the labeling phase. To achieve early pruning of inconsistent values, the sender process remains blocked after the transmission of the message, until the receiver process responds with an acceptance or rejection of the value. In order to provide such a response the receiver process introduces to the store all constraints that derive from the received value.

When all variables of the first process are instantiated an `end` message is sent to the second process which in turn searches for a solution. If such is found then the program terminates with success, otherwise it sends a `backtrack` message to the sender process and backtracks itself to the last choice point. However

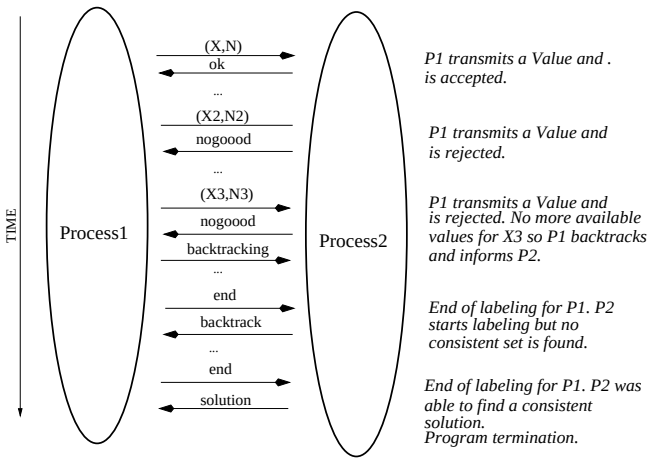

Fig. 1. Message exchange in the Multi-process version.

since the sender process might backtrack not only over the last choice point but also over previous points, the receiver has to be notified so that it can it turn remove any constraints from the store that were introduced because of the previous values transmitted. This extra synchronization is achieved by an appropriate `backtracking` message sent by the first process to the second. If for some reason no solution can be found, the first process sends a failure message to the second indicating that no valid values were possible to be found. The types of messages that are exchanged under in the ESB algorithm are listed in Table 1. Message exchange is shown in Figure 1. We have run several tests for the above versions for various number of queens from N=8 to 28. Speedups for various N compared with the single process version are shown in Figure 2.

Table 1. Types of Messages

Message	Description
`value(X)`	Transmission of a value X, to which a variable was instantiated.
`ok`	Acceptance of a value.
`nogood`	Rejection of a value.
`backtrack`	No labeling found for local variable set. Backtracking of sender process is forced.
`backtracking`	Sender process informs backtracking over a previous choice point.
`end`	The first process has finished with the assignment of values.
`solution`	Reporting that a consistent solution was found.
`failure`	No solution was found. Program termination.

As shown in the figure the multi-process versions are less efficient for a small number of queens justified by the fact that the communication overhead for these cases is comparable to the actual time of computing the solution. However as the number of queens grows the situation is reversed. The speedup obtained is justified by the fact that each process has to solve an easier problem compared to the full N queens problem.

As expected the ESB version performs significantly better that the simple synchronous version, since communicating each value as soon as it is instantiated allows early detection of inconsistencies.

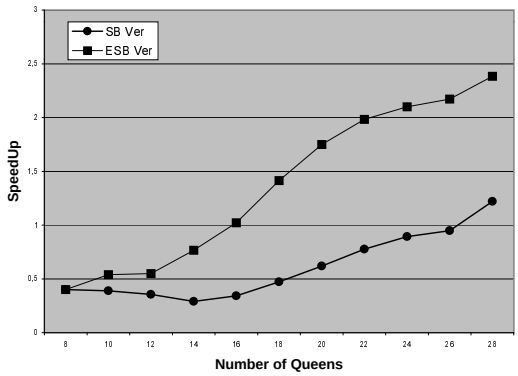

Fig. 2. Speed Up of the Multi-process versions.

6 Conclusions and Future Work

The CSPCONS language, presented in this paper, offers a suitable platform for the development 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. In such an application each agent can be an independent CSPCONS process that exchanges messages with other agents in order to achieve a global consistency.

One of the main points that we are going to concentrate on, is the implementation of a more efficient FD solver. Our plans include the implementation of either the indexical approach to constraint solving or the incorporation of new arc consistency algorithms.

We are currently investigating the implementation of other DCSP algorithms as for example those reported in [15],[4] and in [11]. 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 to provide to the programmer the necessary primitives in order to declare which agents share variables under which constraints and propagate messages automatically. 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 areas of application 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. D. Cabeza and 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*, September 1995.
2. Ivan Futo. Prolog with Communicating Processes: From T-Prolog to CSR-Prolog. In D.S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 3–17. The MIT Press, 1993.
3. Ivan Futo. A Distributed Network Prolog System. In *Proceedings of the 20th International Conference on Information Technology Interfaces, ITI 99*, pages 613–618, 1998.
4. Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in Distributed Constraint Networks. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 219–223, Chichester, August 23–28 1998. John Wiley & Sons.
5. M. Hermenegildo, F. Bueno, D. Cabeza, M. Garcia de la Banda, P. Lopez, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. pages 65–85, April 1999.

6. M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 631–646, Cambridge, June 13–18 1995. MIT Press.
7. C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
8. Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
9. T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1–2):227–250, January 1998. Concurrent constraint programming (Venice, 1995).
10. Marius Calin Silaghi, Djamila Sam-Haroud, and Boi 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)*, pages 917–922, Menlo Park, CA, July 30– 3 2000. AAAI Press.
11. M.C. Silaghi, D. Sam-Haroud, and B.V. Faltings. Maintaining hierachical distributed consistency. In EPFL, editor, *Proceedings of the CP2000 Workshop on Distributed Constraint Satisfaction*, Tech. Report # 00/338, 2000.
12. Bo-Ming Tong and Ho-Fung Leung. Data-parallel concurrent constraint programming. *The Journal of Logic Programming*, 35:103–150, 1998.
13. I. Vlahavas, N. Bassiliades, I. Sakellariou, M. Molina, S. Ossowski, I. Futo, Z. Pasztor, J. Szeredi, I. Velbitskiy, S. Yershov, S. Golub, and I. Netesin. System Architecture of a Distributed Expert System for the Management of a National Data Network. In Fausto Giunchiglia, editor, *Proceedings of the 8th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA-98)*, volume 1480 of *LNAI*, pages 438–451, Berlin, September 21–23 1998. Springer.
14. Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 10(5):673–685, 1998.
15. Makoto Yokoo and Katsutoshi Hirayama. Algorithms for Distributed Constraint Satisfaction: A Review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, June 2000.
16. Ying Zhang and Alan K. Mackworth. Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Technical Report TR-92-30, Department of Computer Science, University of British Columbia, November 1992.