Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Coordination of Heterogeneous Distributed Cooperative Constraint Solving

F. Arbab and E. Monfroy

Information Systems (INS)

SEN-R9828 October 1998

Report SEN-R9828
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Coordination of Heterogeneous Distributed Cooperative Constraint Solving

Farhad Arbab and Eric Monfroy
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
In this paper we argue for an alternative way of designing cooperative constraint solver systems using a control-oriented coordination language. The idea is to take advantage of the coordination features of MANIFOLD for improving the constraint solver collaboration language of **BALI**. We demonstrate the validity of our ideas by presenting the advantages of such a realization and its (practical as well as conceptual) improvements of constraint solving. We are convinced that cooperative constraint solving is intrinsically linked to coordination, and that coordination languages, and MANIFOLD in particular, open new horizons for systems like **BALI**.

1991 Computing Reviews Classification System: D3.3, D.1.3, D.3.2, F.1.2, I.1.3, D.1.m (Cooperative Constraint Solving), D.3.m (Constraint Programming Languages).
Keywords and Phrases: parallel computing, coordination models and languages, dynamic coordination, solver collaboration language, constraint solver cooperation.

1. INTRODUCTION

The need for constraint solver collaboration is widely recognized. The general approach consists of making several solvers cooperate in order to process constraints that could not be solved (at least not efficiently) by a single solver. **BALI** [21, 23, 22] is a realization of such a system, in terms of a language for constraint solver collaboration and a language for constraint programming. Solver collaboration is a glass-box mechanism which enables one to link black-box tools, i.e., the solvers. **BALI** allows one to build solver collaborations (solver cooperation [25] and solver combination [17]) by composing component solvers using collaboration primitives (implementing, e.g., sequential, concurrent, and parallel collaboration schemes) and control primitives (such as iterators, fixed-points, and conditionals).

On the other hand, the concept of coordinating a number of activities, such that they can run concurrently in a parallel and distributed fashion, has recently received wide attention [4, 5]. The IWIM model [1, 2] (Ideal Worker Ideal Manager) is based on a complete symmetry between and decoupling of producers and consumers, as well as a clear distinction between the computational and the coordination/communication work performed by each process. A direct realization of IWIM in terms of a concrete coordination language, namely MANIFOLD [3], already exists.

Due to lack of explicit coordination concepts and constructs, the implementation of **BALI** does not fully realize its formal model: the treatment of disjunctions and the search are jeopardized and this is not completely satisfactory from a constraint solving point of view. This is mainly due to two causes: (1) the dynamic aspect of the formal model of **BALI**, and (2) the use of heterogeneous solvers, i.e. , solvers written in different programming languages, with different data representations. Only a coordination language able to deal with dynamic processes and channels (creation, duplication, dis-/re-/connection), and able to handle external heterogeneous solvers (routines for automatic data conversions) can fullfil the requirements of the formal model of **BALI** and overcome the problem of its current implementation. This guided us through the different coordination models and lead us to the IWIM model, and the MANIFOLD language.

Coordination and cooperative constraint solving are intrinsically linked. This motivated our investigation of a new organizational model for **BALI** based on MANIFOLD. The results show a wider-

than-expected range of implications. Not only the system can be improved in terms of robustness, stability, and required resources, but the constraint solving activity itself is also improved through the resulting clarity of search, efficient handling of the disjunctions, and modularity. The system can be implemented closer to its formal model and can be split up into three parts: (1) a constraint programming activity, (2) a solver collaboration language, and (3) a coordination/communication component. We qualified (and roughly quantified) the improvements coordination languages, and more specifically MANIFOLD, can bring to cooperative constraint solving. The conclusions are promising and we feel confident to undertake a future implementation of **BALI** using MANIFOLD.

The rest of this paper is organized as follows. The next section is a brief overview of **BALI**, its organizational model, and the weaknesses of its implementation. In Section 3, after an overview of MANIFOLD, we describe the coordination/communication of **BALI** using the features of the MANIFOLD system. We then highlight the improvements that we feel are most significant for constraint solving (Section 4). Finally, we conclude in Section 5 and discuss some future work.

2. BALI: AN ENVIRONMENT FOR SOLVER COLLABORATIONS

BALI [21] is an environment for solver collaboration (i.e., solver cooperation [25, 14] and solver combination [26, 29]) that separates constraint programming (the *host language*) from constraint solving (the *solver collaboration language*). The host language is a constraint programming language [34] or possibly a constraint logic programming language [16, 11] which, when necessary, expresses the required solver collaboration through the solver collaboration language. The solver collaboration language supports three strategies called *solving strategies*. The first strategy consists of determining the satisfiability of the constraint store each time a new constraint occurs ("incremental use of a solver"). The second strategy is an alternative to this method that solves the constraint store when a final state is reached (e.g., the end of resolution for logic programming). The last strategy allows the user to trigger the solvers on demand, for example, to test the satisfiability of the store after several constraints have been settled. Furthermore, **BALI** allows several solver collaborations, in conjunction with different solving strategies, to coexist in a single system. For example, solver S_1 can be used incrementally while S_2 only executes at the end, and S_3 and S_4 are always triggered by the user.

Since the constraint programming part of **BALI** is less interesting from the point of view of coordination[1], this paper focuses on its constraint solving techniques, i.e., the constraint solver collaboration language of **BALI**. This domain independent language has been designed for realizing a solving mechanism in terms of solver collaborations following certain solving strategies. The basic objects handled by the language are heterogeneous solvers. They are used inside *collaboration primitives* that integrate several paradigms (such as sequentiality, parallelism, and concurrency) commonly used in solver combination or cooperation. In order to write finer strategies, we have also introduced some *control primitives* (such as iterator, fixed-point, and conditional) in the collaboration language.

At the implementation level, **BALI** is a distributed cooperative constraint programming system, composed of a language for solver collaboration (whose implementation allows one to realize servers to which potential clients can connect) plus a host language (whose implementation is a special client of the server). Solver collaboration is a glass-box mechanism which enables one to link black-box tools, i.e., the solvers.

Some applications have already used **BALI** [23]. For example, a simulation of **CoSAc** [25] has been realized, and some other solver collaborations have been designed for non-linear constraints.

2.1 The Constraint Solver Collaboration Language Of BALI

A detailed description of the solver collaboration language of **BALI** can be found in [23, 21]. In this section, we give a brief overview of some of the collaboration primitives of **BALI**. The complete syntax of the solver collaboration language of **BALI** is given in Figure 1.

[1] The constraint programming part of **BALI** is described in [21] and [22].

Sequentiality (denoted by `seq`) means that the solver E_2 will execute on the constraint store C', which is the result of the application of the solver E_1 on the constraint store C.

When several solvers are working in *parallel* (denoted by `split`), the constraint store C is sent to each and every one of them. Then, the results of all solvers are gathered together in order to constitute a new constraint store analogous to C.

Concurrency (denoted by `dc`) is interesting when several solvers based on different methods can be applied to non-disjoint parts of the constraint store. The result of such a collaboration is the result of a single solver S composed with the constraints that S did not manipulate. The result of S must also satisfy a given property ψ which is a *concurrency function* (the set Ψ in Figure 1). For example, *basic* is a standard function in Ψ that returns the result of the first solver that finishes executing. Some more complex ψ functions can be considered, such as *solved_form* which selects the result of the first solver whose solution is in a specific solved form on the computation domain. The results of the other solvers (which may even be stopped as soon as S is chosen) are not taken into account. The concurrency primitive is similar to a "don't care" commitment but also provides control for choosing the new store (using ψ functions).

$$
\begin{array}{l}
Id \ \in\ \mathcal{I} \text{ (identifiers)}\\
S \ \in\ \mathcal{S} \text{ (solvers)}\\
\psi \ \in\ \Psi \text{ (concurrency functions)}\\
n \ \in\ \mathbb{N} \text{ (positive integers)}\\
OA \ \in\ \mathcal{OA} \text{ (arithmetic observation functions)}\\
OB \ \in\ \mathcal{OB} \text{ (boolean observation functions)}\\
Col \ ::=\ Id = E\\
E \ ::=\ \Diamond \mid Id \mid S \mid seq(SE) \mid\\
\qquad\qquad dc(\psi, SE) \mid split(SE) \mid\\
\qquad\qquad f_p(E) \mid rep(Ar, E) \mid\\
\qquad\qquad if(B, E, E)\\
SE \ ::=\ E \mid E, SE\\
Ar \ ::=\ n \mid Ar + Ar \mid Ar - Ar \mid Ar * Ar \mid OA\\
B \ ::=\ true \mid false \mid\\
\qquad\qquad Ar < Ar \mid Ar \leq Ar \mid\\
\qquad\qquad Ar = Ar \mid B \wedge B \mid\\
\qquad\qquad B \vee B \mid \neg B \mid OB
\end{array}
$$

Figure 1: Syntax of the solver collaboration language of **BALI**

These primitives (which comprise the computation part of the collaboration language) can be connected with combinators (which compose the control part, using primitives such as iterators, conditionals, and fixed-points) in order to design more complex solver collaborations.

The *fixed-point* combinator (denoted by `f_p`) repeatedly applies a solver collaboration until no more information can be extracted from the constraint store. This combinator allows one to create an idempotent solver/collaboration from a non-idempotent solver/collaboration.

The above primitives and combinators are completely statically defined. We now introduce *observation functions* of the constraint store which allow one to get more dynamic primitives. These functions are evaluated at run-time (when entering a primitive) using the current constraint store. These functions may be either arithmetic (the set $\mathcal{OA}$ in Figure 1) or Boolean (the set $\mathcal{OB}$ in Figure 1). Arithmetic observation functions have the profile: $\mathcal{S}tores \rightarrow \mathbb{N}$. Three such functions are:

(1) *card_var* computes the number of distinct variables in the constraint store. This is interesting for solvers that are sensitive to the number of variables. (2) *card_c* returns the number of atomic constraints that comprise the store. This is important for solvers whose complexity is a function of the number of constraints (such as solvers based on propagation). (3) *card_uni_var* returns the number of univariate atomic constraints. This is essential for solvers whose efficiency is improved with univariate constraints (such as interval propagation solvers).

Boolean observation functions have the profile: $\mathcal{S}tores \rightarrow Boolean$. Three such functions are: (1) *linear* tests whether there exists any variable that occurs more than once in an atomic constraint. This is of interest in deciding the applicability of a linear solver. (2) *uni_var* tests whether there is at least one univariate equality in the store. This information is important since, for example, univariate constraints are generally the starting point of interval propagation. (3) *tri* tests whether the store is in triangular form (i.e., there are some equality constraints over a variable X, some over variables X and Y, some over X, Y and Z, ...). This is interesting for eliminating variables, or determining an ordering for the Gröbner bases computation.

The *repeat* combinator (denoted by `rep`) is similar to the *fixed-point* combinator, but allows applying a solver n times: n is the result of the application of an *observation function* (or a composition of observation functions) to the constraint store. Since this primitive takes into account the constraint and its form at run-time, it improves the dynamic aspect of the collaboration language.

Finally, the *conditional* combinator (denoted by `if`) applies one solver/collaboration or another, depending on the evaluation of a condition (which can also depend on observation functions of the constraint store).

The following example illustrates the solver collaboration language:

```
seq(A,dc(basic,B,C,D),split(E,F),f_p(G))
```

Consider applying this collaboration scheme to the constraint store c [2]. First `A` is applied to c and returns c_1. Then, `B`, `C`, and `D` are applied to c_1. The first one that finishes gives the new constraint store c_2. Then `E`, and `F` execute on c_2. The solution c_3 is a composition of c_3' (the solution of `E`) and c_3'' (the solution of `F`). Finally, `G` is repeatedly applied to c_3 until a fix-point, c_4, is reached, which is the final solution of the collaboration.

2.2 Organizational Model And Implementation

The role of the organizational model we have implemented is: 1) to create a distributed environment for integrating *heterogeneous solvers* [3], 2) to establish communication between solvers in spite of their differences, 3) to coordinate their executions. Such an organizational model turns solver collaborations into *servers* to which clients (such as the implementation of the host language or all kinds of processes requiring a solver) can *connect*. This model enabled us to implement **BALI** and create/execute solver collaborations [21].

2.2.1 Agent The realizations of solvers and solver collaborations are heterogeneous. However, by an encapsulation mechanism we homogenize the system, and obtain what we call *agents*. Each agent is autonomous and is created, works, and terminates independently from the others. Hence, agents can execute in parallel or concurrently in a distributed architecture.

Solvers are encapsulated to create *simple agents*. As shown in [21], a solver collaboration is a solver. Applying this concept to the architecture, encapsulation becomes a hierarchical operation. Hence, several simple agents can be encapsulated in order to build a *complex agent*. However, viewed from the outside of a capsule, simple and complex agents are identical.

[2] In order to simplify the explanation, we consider here solvers that return only one solution (one disjunct). We detail the treatment of disjunctions in the next sections.

[3] Each solver (software, library of tools, client/server architecture) has its own data representation, is written in a different programming language, and executes on a different architecture and operating system.

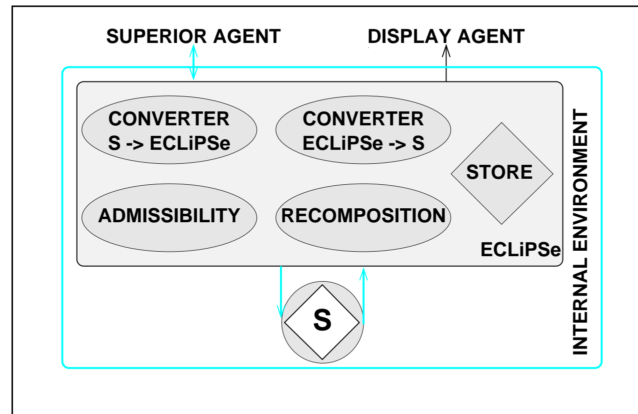

Figure 2: Simple agent

In the current implementation of BALI, solvers are encapsulated into ECLiPSe[4] processes (see Figure 2). Hence, ECLiPSe launches the solvers and re-connects their input and output through pipes. The data structure converters are written in Prolog and the data exchanges between capsules and solvers are performed via strings. The encapsulation also provides a constraint store for the solver it represents (a local database for storing the information), an admissibility function (which is able to recognize which constraints of the store can be handled by the solver), and a re-composition function (which recreates an equivalent store using the constraints treated by the solvers, and the constraints not admissible by the solver). The interface of an agent is an ECLiPSe process. Moreover, Prolog terms can be transmitted between two ECLiPSe processes. Inter-agent communication is thus realized with high level terms, and not strings or bits. Furthermore, there is no need for syntactic analyzers between pairs of agents.

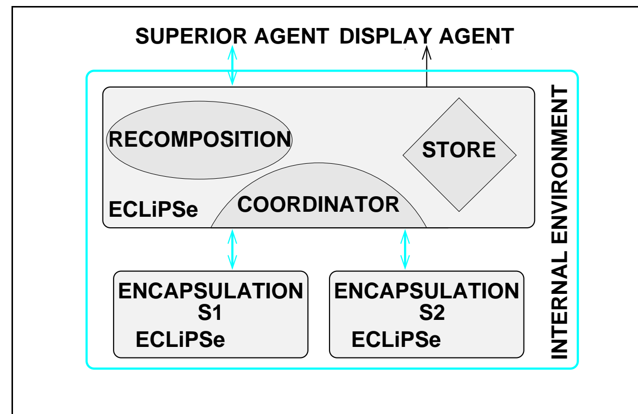

Figure 3: Complex agent

A complex agent (encapsulation of a solver collaboration) behaves like a simple agent, though its internal environment is a bit different (see Figure 3). It has a constraint store for keeping the information it receives: this is its knowledge base. For managing this base, it has a re-composition function which re-builds the constraint store when some agents send some of their solutions. The major work of a complex agent is the coordination (as determined by the collaboration primitive it represents) of the agents it encapsulates.

[4] ECLiPSe [20] is the "Common Logic Programming System" developed at ECRC.

2.2.2 Coordination We now describe the coordination of the implementation of BALI (see [21] for more details), but not the coordination of its formal model. An agent can be in one of three different *states*: running (R), sleeping (S), or waiting (W). When an agent receive a constraint c, it becomes running to solve c. An agent is in the W state when it is waiting for the answer from one or more agents. An agent is in the S state when it is neither running nor waiting. These states together with the communication among agents, enable us to describe the coordination of the constraint solvers.

Sequential primitive: seq(S_1,S_2, ... ,S_n) tries to solve a constraint by sequentially applying several solvers. It first sends a constraint to S_1 and waits for a solution c_1 for it. When it receives a solution from S_1, it sends it to S_2, waits for a solution c_2, sends it to S_3, and so on, until it reaches S_n. Finally, the solution c_n from S_n is forwarded to the superior agent as one of the solutions of the sequential primitive. Since we consider solvers that enumerate their solutions (i.e., each solution represents a disjunct of the complete solution), the sequential agent must wait for the other disjuncts of S_n which will be treated the same way as c_n. Backtracking is then performed on S_{n-1}, S_{n-2} and back to S_1. In a sequential collaboration, several agents are "pipelined" and work in "parallel", but the solutions are passed "sequentially" from one agent to the next.

Split primitive: split(S_1,S_2, ... ,S_n) applies several solvers in parallel on the same constraints. The solution of split is a Cartesian-product-like re-composition of all the solutions of S_1,S_2, ... ,S_n. When a split agent receives a solve request from its superior, it forwards it to all its S_i's. Then, it waits and stores all the solutions of each S_i. Finally, the split agent creates all the elements of the Cartesian-product of the solutions, and sends them one by one to its superior agent.

ψ_don't care primitive: dc(ψ_1,S_1,S_2, ... ,S_n) introduces concurrency among solvers. Upon receiving a constraint c from its superior, the don't care agent forwards c to all its sub-agents, S_i's. Then it waits for a solution c' from any of its sub-agents. If c' does not satisfy ψ_1 [5] then c' is forgotten and the don't care agent waits for a solution from another sub-agent (other than the one that produced c'). As soon as the don't care agent receives a solution c' from some S_i that satisfies ψ_1, all other sub-agents are stopped and c', as well as all other solutions produced by S_i, are forwarded to the superior agent.

fix-point primitive: f_p(S) repeatedly applies S on a constraint, until no more information can be extracted from the constraint. The solving process starts when the fix-point agent receives a constraint c from its superior. It is an iterative process and in each iteration k, we consider a set C_k of disjuncts to be treated by S (e.g., in iteration 1, C_1 consists of a single element, c). In iteration k, the m_k disjuncts of C_k must be treated by S: the fix-point agent chooses one element of C_k, $c_{k,i}$, removes it from C_k, sends it to S and collects all the solutions from S. If the [6] solution from S is equal to $c_{k,i}$ (a fix-point has been reached for this disjunct), the fix-point agent forwards it to its superior agent. Otherwise, the solutions produced by S are added to C_{k+1}. The same treatment is applied to all the elements of C_k to complete the set C_{k+1} and the solving process enters iteration $k+1$. The process terminates when at the end of iteration k, the set C_{k+1} is empty.

repeat primitive: The coordination for the *repeat* primitive (rep(δ,S)) is identical to the fix-point collaboration, except that it stops after a given number n of iterations. The number n is computed at run-time: it is the result of the application of the arithmetic function δ to the current constraint store. The arithmetic function δ is composed of arithmetic observation functions of the constraint store (elements in $\mathcal{OA}$, see Figure 1). The solving process starts when the repeat agent receives a constraint c from its superior. First, n is computed: $n = \delta(c)$. Then, the coordination is analogous to the one of the fix-point primitive. The process terminates at the end of iteration n, when every solution returned by S for every disjunct in C_n is sent to the superior agent.

[5] ψ_1 is an element of the set ψ of boolean functions. They test whether or not a constraint satisfies some properties.
[6] When reaching a fix-point, a solver can return only one solution.

conditional primitive: `if(γ,S1,S2)` is reather simple. When it receives a constraint c from its superior agent, this primitive applies the function γ to c. The Boolean function γ is composed of both arithmetic and Boolean observation functions of the constraint store. If $\gamma(c)$ is true, then c is forwarded to the sub-collaboration `S1`, otherwise to the sub-collaboration `S2`. Then, this primtive becomes an intermediary between one of the sub-agents and its superior agent, i.e., as soon as the selected sub-agent sends a solution, it is forwarded immediately to the superior agent. In fact, after evaluation of $\gamma(c)$ the conditional primitive acts similarly to a sequential primitive having a single sub-agent.

2.3 Weaknesses Of The Implementation

Although ECLiPSe provides some functionality for managing processes and communication, it is not a coordination language. Thus, our implementation does not exactly realize the formal model of **BALI**: some features are jeopardized, or even missing, as described below.

Disjunctions of constraints The disjunctions of constraints returned by a solver are treated one after the other, and for some primitives, they are even stored and their treatment is delayed. For the sequential primitive, this does not drastically jeopardize the solving process. But for the fix-point primitive, this really endangers the resolution. We must wait for all the disjuncts of a given iteration before entering the next one. A solution would be to duplicate the solver; but due to the encapsulation mechanism, this is not reasonable. This treatment of disjunction leads to a loss of efficiency, and to a mixed search[7] during solving (which is not completely convenient from the constraint programming point of view).

Static architecture Another limitation of **BALI** is due to the fact that architectures representing collaborations are fixed. Due to some implementation constraints and the limitations of coordination features of ECLiPSe, the collaborations are first completely launched before being used to solve constraints. Thus, we have a loss of dynamics: 1) parts of the architecture are created even when they are not required, 2) agents cannot be duplicated (although this would be interesting for some primitives such as fix-point), and 3) as stated before, the disjunctions are not always handled efficiently.

Other compromised features Although the formal model of **BALI** allows the use of "light" solvers, the implementation is not well suited to support such agents: their coarse grain encapsulation uses more memory and CPU than the solver. Thus, mixing heavy solvers (such as GB [10], Maple [12]) and light solvers (such as rewrite rules or transformation rules) is not recommended.

No checks are made to ensure that an architecture and its communication channels have been created properly. Management of resources and load balancing are static: before launching a collaboration, the user must decide on which machine the solver will run.

3. MANIFOLD: A NEW COORDINATION FOR BALI

We now explain how we can use the coordination language MANIFOLD [3] to significantly improve the implementation of **BALI**, and remain closer to its formal model.

3.1 The Coordination Language MANIFOLD

MANIFOLD is a language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperative processes [1]. MANIFOLD is based on the IWIM model of communication [2]. The basic concepts in the IWIM model (thus also in MANIFOLD) are *processes*, *events*, *ports*, and *channels*. Its advantages over the *Targeted-Send/Receive* model (on which object-oriented programming models and tools such as PVM [13], PARMACS [15], and MPI [7] are based) are discussed in [1, 27].

[7] The search strategy is breadth-first for the fix-point and repeat primitives, but depth-first for the sequential and don't care primitives.

A MANIFOLD application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages.

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, libraries of built-in and pre-defined processes, a link file generator called MLINK and a run-time configurator called CONFIG. The system has been ported to several different platforms (e.g., SGI Irix 6.3, SUN 4, Solaris 5.2, IBM SP/1, SP/2, and Linux). MLINK uses the object files produced by the (MANIFOLD and other language) compilers to produce link files and the makefiles needed to compose the executables files for each required platform. At the run time of an application, CONFIG determines the actual host(s), where the processes (created in the MANIFOLD application) will run.

The library routines that comprise the interface between MANIFOLD and processes written in the other languages (e.g., C), automatically perform the necessary data format conversions when data are routed between various different machines.

MANIFOLD has been successfully used in a number of applications, including in parallelization of a real-life, heavy duty Computational Fluid Dynamics algorithm originally written in Fortran77 [8, 9, 18], and implementation of Loosely-Coupled Genetic Algorithms on parallel and distributed platforms [31, 33, 32].

3.2 BALI *In* MANIFOLD

Although BALI solvers are black-boxes and are heterogeneous, this does not cause any problems for MANIFOLD, because it integrates the solvers as external workers. Thus, communication and coordination can be defined among them in the same way as with normal MANIFOLDagents. MANIFOLD can bring many improvements to BALI such as:

- robustness: managing the faults in the system is not an easy task with ECLiPSe.
- portability: MANIFOLD runs on several architectures, and requires only a thread facility and a subset of PVM [13].
- modularity: in the current implementation, constraint solving is separated from constraint programming. Using MANIFOLD, we can also split up the coordination part from the solving part.
- extension of the collaboration language: each primitive will be an independent coordinator. Thus, adding a new primitive will be simplified.
- additional new features: MANIFOLD provides tools to implement certain functionalities that are not available in the current version of BALI (e.g., choice of the machines, light weight processes, architectures, load balancing, etc.).

In the following, we elaborate only on the most significant of the above points, i.e., the ones that make an intensive use of the MANIFOLD features or the ones that are the most significant for constraint solving.

3.2.1 Lighter Agents The current encapsulation (one ECLiPSe process for each solver/collaboration) is really heavy. MANIFOLD can produce lighter capsules using threads to realize filters and workers. They will replace the computation modules of ECLiPSe. Thus, a simple agent (see Figure 4) can consist of:

- a coordinator for managing the messages and agents inside the encapsulation. This coordinator is also the in/out gate of the capsule (when communicating with superior agents).
- a solver, which is the same as in the previous implementation.
- four *filters* (MANIFOLD workers): the first filters the constraints the solver can handle, the second converts the data into the syntax of the solver, the third converts the solutions of the solver into the global syntax [8] and the last re-composes equivalent solutions based on the solutions of the solver and the constraints it cannot handle.

[8] Global syntax is the syntax used in the filters and between agents.

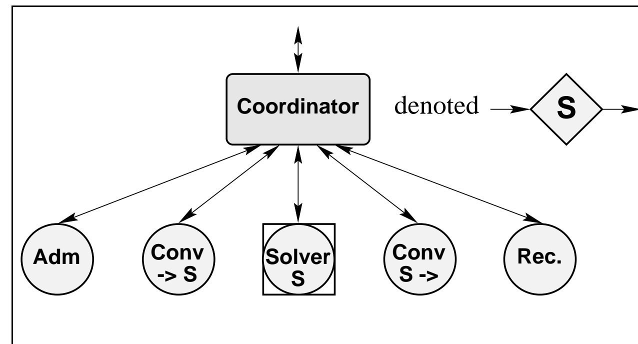

Figure 4: Lighter simple agent

A complex agent (see Figure 5) is now the encapsulation of several simple/complex agents together with some filters. The filters and the coordinator (coordinators are described in Section 3.2.2) are specialized for the collaboration primitive the agent represents. For a split collaboration, only one filter is required: a store manager which collects all the solutions from the sub-agent and incrementally builds the elements of their Cartesian-product (as soon as one element is completed, it is sent to the coordinator). In a ψ_don't care primitive, one filter is required for applying the ψ function to the constraints. For the sequential primitive, as well as the fix-point, no filters are required.

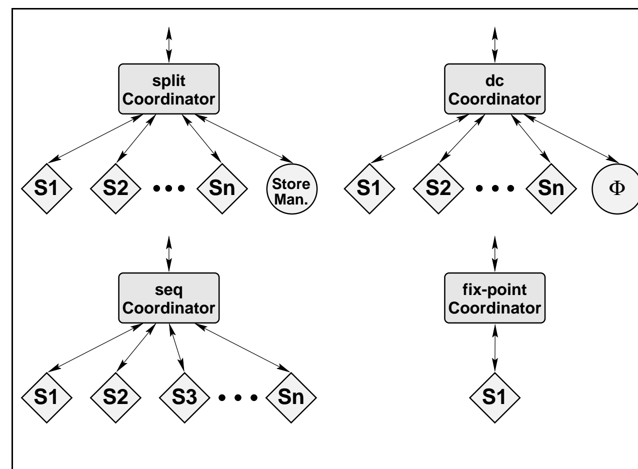

Figure 5: Lighter complex agent

This new kind of encapsulation has several advantages. The global architecture representing a solver collaboration will require less processes than before, and also less memory. This is due to several facts: the use of threads instead of heavy processes, the notion of filters, and the sharing of workers, filters, and solvers between several agents (see Figure 6). The creation of another instance of a solver will depend on the activity of the already running instances. Agents are not black-boxes anymore: they become glass-boxes sharing solvers and filters with other agents. But the main advantage is certainly the following: the coordination is now separated from the filters, encapsulated into individual modules, each of which depends on the specific type of collaboration it implements, and can use all the features of MANIFOLD. Thus, it is possible to arrive at a coordination scheme that respects the formal model of BALI.

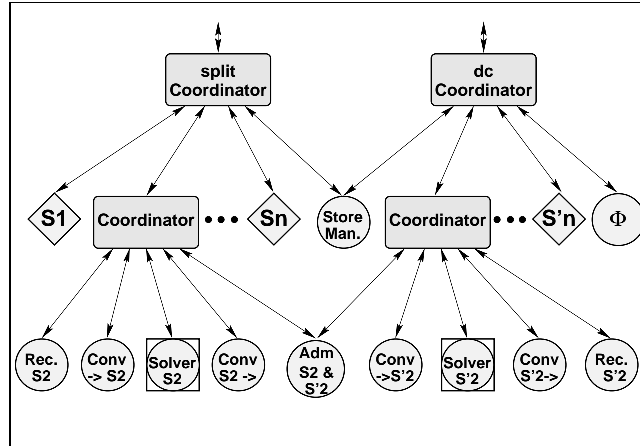

Figure 6: Shared solvers and filters

3.2.2 Coordinators Using MANIFOLD and the new encapsulation process, it is now possible to overcome the problems inherent in the previous implementation of BALI.

Dynamic handling of the solvers Since the coordination features are now separated from the filters and workers, the set up of the distributed architecture and its use are no longer disjoint phases. This means that when a solving request is sent, the collaboration will be built incrementally (agent after agent) and only the necessary components will be created. For example, in a conditional or guarded collaboration, only the "then" or the "else" sub-collaboration will be launched. If another request is sent to the same collaboration, the launched components will be re-used, possibly augmented by some newly created components.

When a solver/collaboration is requested to solve a constraint, several cases can arise. If the solver/collaboration S has not already been launched, then an instance of S will be created. If it is already launched but all of its instances are busy (i.e., all instances of S are currently working on constraints) another instance will be created. Otherwise, one of the instances will be re-used for the new computation. The function `find_instance` manages this functionality (see Appendix A.1).

Dynamic handling of the disjunctions Contrary to the current implementation [9], disjunctions are treated dynamically. We demonstrate this for the sequential collaboration seq(S_1,S_2, ... ,S_n). All the disjuncts produced by S_1 must be sent to S_2. With ECLiPSe, a disjunct c_1 of S_1 is completely solved by S_2, ... , S_n (meaning all possible disjuncts created by S_2, ... , S_n are produced), before treating the next disjunct c_2 of S_1. MANIFOLD allows us to use pipelines to solve c_2 as soon as it is produced by S_1. If S_2 is still working on c_1, and all the other instances of S_2 are busy, then a new instance of S_2 is created for solving c_2. The treatment of c_2 is no longer postponed. This mechanism applies to all sub-agents of the sequential agent.

This introduces a new problem: there may be a combinatorial explosion of the number of instances of S_2, ... , S_n. However, this can rarely happen: while an agent S_i is producing solutions, the agent S_{i+1} is already solving (and has already solved) some of the previous constraints. Thus, some instances have already returned to a sleeping state and can be re-used. Nevertheless, the following case may arise. Suppose the solvers S_i are arranged such that as the index i grows, the designated solvers, S_i, become slower, and suppose every S_i creates disjuncts. The number of instances will become exponential in this case, and the system will therefore run out of resources. In order to overcome this problem, the number of instances can be limited (see Appendix A.1). Thus, when a solving request is

[9] Currently, the fix-point coordinator waits for all the solutions of the sub-collaboration before entering the next iteration.

to be sent to the agent S, and the maximal number of (its) instances is reached, and all its instances are busy, the superior agent will wait for the first instance to return to the sleeping state. This mechanism does not imply a completely dynamic treatment of the disjunctions. However, it gives a good compromise between the delay for solving a disjunct and the physical limitation of the resources.

Coordinators for the primitives We now describe the coordinators for the sequential primitive. Some other primitives are detailed in Appendix A. The algorithms are presented here in a Pascal-like language extended with an `event` functionality. We consider a queue of messages `m from p` meaning that the message `m` was received on the port `p`. `task m from p alg` means that we remove the message `m` from the port `p` and execute the algorithm `alg` (the message `m from p` is the condition for executing the task `alg`). The latter cannot be interrupted. `end` is a message that is sent by an agent when it has enumerated all its disjuncts. The agents have a number of flags representing the states described in Section 2.2.2.

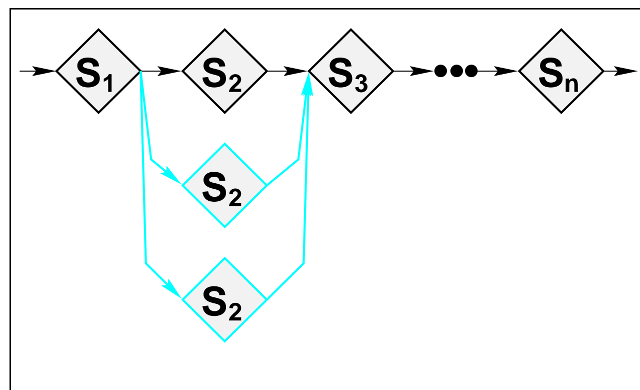

Figure 7: Duplication of a sequential primitive

```
coordinator for seq(S1,...,Sn)

S1...Sn: sub-agents; S0: sup-agent

ports: p.0.in ... p.n.in
       % for 0=<i<n+1 p.i.in is linked to p.Si.out of Si

       p.0.1.out ... p.0.n+1.out
       % for 0<i<n+1 p.0.i.out is linked to p.Si.in of Si
       % p.0.n+1.out is linked to p.S0.in of S0

task c from p.i.in:
       j=find_instance(i+1)
       if j<>NULL
            then send c to p.j.i+1.out; Mi=Mi+1
            else send c to p.i.in
            % c is sent again and again to p.i.in
            % till an instance of Si+1 becomes free

task end from p.i.in:
       end.i=end.i + 1
       if end.i = M.i-1 and Si-1 is Sleeping
          then Si is Sleeping
```

```
task end from p.n.in:
       end.n=end.n + 1
       if end.n = M.n-1 and Sn-1 is Sleeping
          then Sn is Sleeping;  send end to p.0.n+1.out
```

`task c from p.i.in` is used to forward a solution from S_i to S_{i+1}. If no instance of S_{i+1} is free, and it is not possible to create a new instance, then the same message is sent again, and will be treated later.

To detect the end of a sequential primitive, we count the solutions and `end` messages of each of the sub-agent. An agent S_i becomes sleeping when S_{i-1} is already sleeping, and when S_i produces as many `end` messages as the number $M.i-1$ of solutions of S_{i-1}.

The superior agent S_0 is never duplicated inside a collaboration, since the coordinator can create only a sub-architecture; the collaboration does not duplicate itself. That is the job of the superior agent: it either finds a free instance of the collaboration, or creates one if the maximal number of instances is not yet reached (see Figures 7 and 8 for an example of duplication).

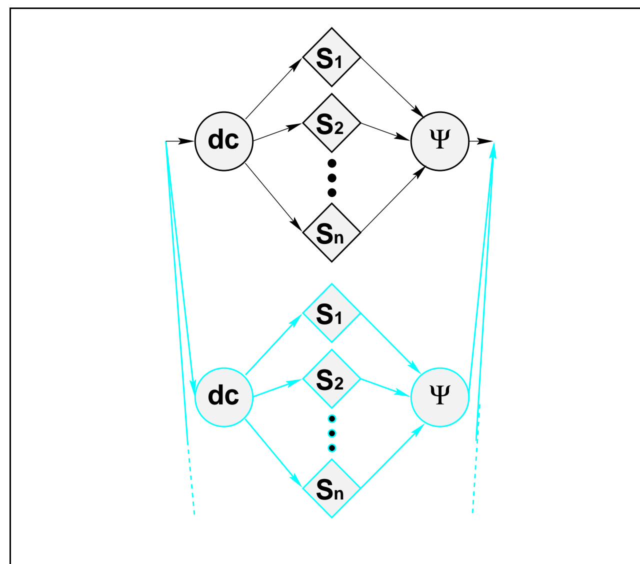

Figure 8: Duplication of a ψ-don't care primitive

4. EXPECTED IMPROVEMENTS

We have seen that coordination languages, and **MANIFOLD** in particular, are helpful for implementing cooperative constraint solving. However, the advantages are not only at the implementation level. **MANIFOLD** allows an implementation closer to the formal model of **BALI**, and this implies some significant benefits for constraint solving: faster execution time, better debugging, and clarity of the search [28] during constraint solving (see Table 9). The architecture also gains through some improvements: robustness, reliability, quality, and a better management of the resources (see Table 10). This last point also has consequences for the end user: as the architectures representing a solver collaboration become lighter, the end user can build more and more complex collaborations, and thus, solve problems that could not be tackled before.

Constraint solving Treatment of disjunctions is a key point in constraint solving. The most commonly required search is depth-first: each time several candidates appear, take one, and continue with it until reaching a solution, then backtrack to try the other candidates. One of the reasons for this choice is that, generally, only one solution is required. Contrary to the first implementation of

BALI, the coordination we described with **MANIFOLD** leads to what we call a "parallel depth-first and quick-first" search. The parallel depth-first search is obvious. The quick-first search arises from the fact that each constraint flows through the agents independently from the others. Hence (ignoring the boundary condition of reaching the instance limits of solvers, mentioned above), it is never delayed by another constraint, nor stops at the input of a solver or in a queue. The result is that the solution which is the fastest to compute (even if it is not originated from the first disjunct of a solver) has a better chance to become the first solution given by the solver collaboration [10].

Debugging, collaboration improvement, and graphical interface to present output will be eased. The coordinators can duplicate the messages and send them to a special worker. This latter can then be linked to a display window (text or graphic) or a profiler. It will enable users observe the flow of data in a collaboration. Thus, users can extract statistics on the utilization of the solvers and draw conclusions on the efficiency of a newly designed collaboration. All this process can lead to a methodology for designing solver collaborations.

Due to its encapsulation techniques, the current implementation jeopardizes the use of "fine grain" solvers (solvers that require little memory and CPU). Although we can envisage encapsulating a single function with an ECLiPSe process, this is not reasonable. Though not really designed for fine grain agents, **MANIFOLD** still gives more freedom to use single functions (such as rewrite rules or constraints transformations) as solvers. With **MANIFOLD**, single functions for simplifying the constraints can easily be inserted in a collaboration as threads without compromising the efficiency of the whole architecture; this significantly enlarges the set of solvers that can be integrated in **BALI**.

BALI in:	ECLiPSe	**MANIFOLD**
search during solving	mixed	depth-first
first solution	−	++++
execution time	+	++
treatment of the disjunctions	−	+++
use of "fine grain" solvers	−	++
add of solvers (encapsulation)	+	+++
extension of the collaboration language	−	+++
"debugging tools"	−	+++
improvement of solver collab.	−	+++
input graphical interface	−	++
output graphical interface	+	++

Figure 9: Improvements for constraint solving

Coordination is now separated from collaboration: a collaboration primitive implies a coordinator separated from the converters, recomposition functions, and admissibility functions. Thus, with the same filters we can easily implement new primitives: only the coordinator has to be modified, and in some cases a filter must be added.

Architecture The major limitation of **BALI** is the large amount of resources it requires. Of course, this is an intrinsic problem with cooperative solvers: they are generally costly in memory, CPU, etc. But another limiting factor is the overhead of the current encapsulation mechanism. With the new encapsulation technique, **MANIFOLD** will decrease the required resources. Furthermore, with dynamic handling of disjunctions, we expect the new architecture to be less voracious.

[10] When a branch leads faster to a solution, we find it quickly, because we do not have to explore all branches before this one.

BALI in:	ECLiPSe	MANIFOLD
construction of the architecture	static	dynamic
robustness	-	+++
extension of BALI	+	+++
stability	-	++
graphical interface (in/output)	-	++
quality of communication	-	++++
coordination functionalities	+	+++++
number of processes but solvers	-	+
number of communications	++	-

Figure 10: Improvements for architectures

The system will gain in robustness, since currently no failure detection of the architecture is possible. The collaborations will be more stable and less susceptible to broken communication and memory allocation problems. The dynamic building of the architecture will decrease the number of unnecessary processes: only the agents required in a computation are launched.

The only negative point is the increased communication. With the current implementation, the encapsulation is composed of two communicating processes: ECLiPSe and a solver. All the filters are modules of the ECLiPSe process. With MANIFOLD, the filters are independent agents that also exchange information. However, this should not create a bottle-neck since messages are generally short, communicating agents are usually threads in the same process that end up using shared memory to communicate, and no single agent conducts nor monitors all communication.

5. CONCLUSION

We have introduced an alternative approach for designing cooperative constraint solving systems. Coordination languages, and MANIFOLD in particular, exhibit properties that are appropriate for implementing BALI. However, implementation improvement is not the only advantage. Using MANIFOLD we can produce a system closer to the formal model of BALI, and some significant benefits are also obvious for constraint solving. The major improvements are the treatment of the disjunctions, the homogenization of search, and the reduction of required resources. A fare management of the disjuncts returned by a solver often leads to quicker solutions. Moreover, due to replication, the complete set of solutions is always computed more efficiently. Although the mixed search used in the current implementation of BALI does not really influence resolution when looking for all the solutions of a problem, it becomes a real nuisance when looking for only one solution. Furthermore, observing the resolution and following the flow of constraints is not conceivable. MANIFOLD overcomes this problem by providing a “parallel depth-first and quick-first” search: each disjunct is handled independently, and thus no constraint resolution is delayed or queued.

Comparing BALI to other systems (such as cc [30] and Oz [19]) is not easy since they do not have the same objectives [21]. cc is a formal framework for concurrent constraint programming, and Oz is a concurrent constraint programming system. However, one of the major distinctions is that BALI, contrary to Oz and cc, enables the collaboration of *heterogeneous* solvers. Another essential difference concerns the separation of tasks. With Oz and cc, constraint programming, constraint solving, and coordination of agents are mixed. With BALI, constraint programming is separated from cooperative constraint solving, and using MANIFOLD, cooperative constraint solving is split up into coordination of agents and constraint solving: each aspect of cooperative constraint programming is an independent task.

Since the implementation model of BALI with MANIFOLD is clearly defined, we can surely start with

the implementation phase. Moreover, we know the feasibility of the task, and have already qualified (as well as roughly quantified) the improvements. Hence, we know that it is a worthwhile work.

In the future, we plan to integrate a visual interface to assist programmers in writing more complex solver collaborations. This can be achieved using Visifold [6] and some predefined "graphical" coordinators.

In order to perform optimization, we are thinking of adding another search technique to **BALI**: a best solution search (branch and bound). This kind of search is generally managed by the constraint language. However, MANIFOLD coordinators that represent collaboration primitives can perform the following tasks: they can eliminate the disjuncts that are above the current "best" solution, and also manage the updating of the current best solution. Branching can, thus, be improved and performed sooner.

The constraint solver extension mechanism of **SoleX** [24] consists of rule-based transformations seen as elementary solvers. Until now, the implementation of **SoleX** with **BALI** was not really conceivable: rule-based transformations are too fine grain solvers to be encapsulated. With the new model, the implementation of **SoleX** becomes reasonable.

Finally, we are convinced that cooperative constraint solving is intrinsically linked to coordination, and that coordination languages open new horizons for systems like **BALI**.

References

1. Arbab, F. Coordination of massively concurrent activities. Report CS–R9565, CWI, Amsterdam, The Netherlands, Nov. 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.
2. Arbab, F. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models* (Apr. 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 34–56.
3. Arbab, F. *Manifold2.0 reference manual.* CWI, Amsterdam, The Netherlands, May 1997.
4. Arbab, F. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI* (1998), 11–22. Available on-line http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html.
5. Arbab, F., Ciancarini, P., and Hankin, C. Coordination languages for parallel programming. *Parallel Computing 24*, 7 (July 1998), 989–1004. special issue on Coordination languages for parallel programming.
6. Bouvry, P., and Arbab, F. Visifold: A visual environment for a coordination language. In *Coordination Languages and Models* (Apr. 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 403–406.
7. Dongarra, J., Otto, S., Snir, M., and Walker, D. An introduction to the MPI standard. Report CS-95-274, University of Tennessee, 1995.
8. Everaars, C. T. H., Arbab, F., and Burger, F. J. Restructuring sequential Fortran code into a parallel/distributed application. In *Proc. of the International Conference on Software Maintenance '96* (Nov. 1996), IEEE, pp. 13–22.
9. Everaars, C. T. H., and Koren, B. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Computing 24*, 7 (July 1998), 1081–1106. special issue on Coordination languages for parallel programming.
10. Faugere, J.-C. *Résolution des systèmes d'équations algébriques.* PhD thesis, Université Paris 6, 1994.
11. Frühwirth, T., Herold, A., Kuechenhoff, V., Le Provost, T., Lim, P., Monfroy, E., and Wallace, M. Constraint Logic Programming - An Informal Introduction. In *Proc. of Logic*

Programming in Action (LPSS'92), Zurich, Switzerland (Sep. 1992), G. Comyn, M. Ratcliffe, and N. Fuchs, Eds., vol. 636 of *Lecture Notes in Computer Science*, Springer Verlag.

12. Geddes, K., Gonnet, G., and Leong, B. *Maple V : Language reference manual*. Springer Verlag, New York, Berlin, Paris, 1991.
13. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. PVM 3 user's guide and reference manual. Report ORNL/TM-12187, Oak Ridge Nat. Laboratory, 1994.
14. Granvilliers, L. A symbolic-numerical branch and prune algorithm for solving non-linear polynomial systems. *Journal of Universal Computer Science, Springer 4* (1998), 125–146.
15. Hempel, R., Hoppe, H., Keller, U., and Krotz, W. PARMACS v6.1 specification. Report, PALLAS GmbH, Hermulheimer Strasse 10, D-50321, 1995.
16. Jaffar, J., and Maher, M. Constraint Logic Programming: a Survey. *Journal of Logic Programming 19,20* (1994), 503–581.
17. Kirchner, H., and Ringeissen, C. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation 18*, 2 (1994), 113–155.
18. Koren, B., Hemker, P., and Everaars, C. Multiple semi-coarsened multigrid for 3d cfd. In *Proc. of the 13th AIAA Computational Fluid Dynamics Conference, Snowmass Village, CO, 1997* (Reston, VA, 1997), American Institute of Aeronautics and Astronautics, pp. 892–902. AIAA-paper 97-2029.
19. Mehl, M., Müller, T., Popov, K., and Scheidhauer, R. *DFKI Oz User's manual*. DFKI, Sarrebrucken (Germany), May 1995.
20. Meier, M., and Schimpf, J. ECLiPSe User Manual. Report ECRC-93-6, ECRC, Munich, Germany, 1993.
21. Monfroy, E. *Collaboration de solveurs pour la programmation logique à contraintes*. PhD Thesis, Université Henri Poincaré-Nancy I, Nov. 1996. Also available in english. Available on-line http://www.cwi.nl/~eric/Private/Publications/index. html.
22. Monfroy, E. An environment for designing/executing constraint solver collaborations. In *Proceedings of COTIC'98, Nice, France* (1998), ENTCS, Elsevier Science Publishers. To Appear.
23. Monfroy, E. The Constraint Solver Collaboration Language of BALI. In *Proceedings of the International Workshop Frontiers of Combining Systems, FroCoS'98* (Amsterdam, The Netherlands, 1998). To Appear.
24. Monfroy, E., and Ringeissen, C. SOLEX: a Domain-Independent Scheme for Constraint Solver Extension. In *Proc. of the Fourth International Conference on Artificial Intelligence and Symbolic Computation (AISC'98)* (Sep. 1998), no. 1476 in Lecture Notes in Artificial Intelligence, Springer Verlag.
25. Monfroy, E., Rusinowitch, M., and Schott, R. Implementing Non-Linear Constraints with Cooperative Solvers. In *Proc. of ACM Symposium on Applied Computing (SAC'96)* (Feb. 1996), K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, Eds., pp. 63–72.
26. Nelson, C. G., and Oppen, D. C. Simplifications by cooperating decision procedures. *ACM Trans. Program. Lang. Syst. 1*, 2 (1979).
27. Papadopoulos, G. A., and Arbab, F. Control-Driven Coordination Programming in Shared Dataspace. In *Proc. of Fourth International Conference on Parallel Computing Technologies (PaCT-97)* (Yaroslavl, Russia, Sep. 1997), Lecture Notes in Computer Science, Springer Verlag. (to appear).
28. Pearl, J. *Heuristics : intelligent search strategies for computer problem solving*, 2nd print ed. Addison-Wesley, 1984.

29. Ringeissen, C. Cooperation of decision procedures for the satisfiability problem. In *Frontiers of Combining Systems* (1996), Applied Logic, Kluwer Academic Publishers.
30. Saraswat, V. *Concurrent Constraint Programming*. MIT Press, Cambridge, London, 1993.
31. Seredynski, F., Bouvry, P., and Arbab, F. Distributed evolutionary optimization in manifold: the rosenbrock's function case study. In *FEA'97 - First International Workshop on Frontiers in Evolutionary Algorithms (part of the third Joint Conference on Information Sciences)* (Mar. 1997). Duke University (USA).
32. Seredynski, F., Bouvry, P., and Arbab, F. Parallel and distributed evolutionary computation with Manifold. In *Proc. of Fourth International Conference on Parallel Computing Technologies (PaCT-97)* (Yaroslavl, Russia, Sep. 1997), Lecture Notes in Computer Science, Springer Verlag. (to appear).
33. Seredynski, F., Bouvry, P., and Arbab, F. Parallel evolutionary computation: Multi agents genetic algorithm. In *International Conference on Parallel and Distributed Systems (Euro-PDS '97)* (Jun. 1997), E. Luque, A. R. Hurson, and H. El-Rewini, Eds., pp. 293–298.
34. Van Hentenryck, P., and Saraswat, V. Strategic Directions in Constraint Programming. *ACM Computing Surveys 28*, 4 (Dec. 1996), 701–726.

A. COORDINATORS FOR THE COLLABORATION PRIMITIVES

A.1 The `find_instance` Function

First, we give the algorithm for the function `find_instance`. This function tries to find a free instance of the i-th sub-agent. If no instance is free and the maximum number of instances (`max_inst`) is not reached, then a new instance is created (see Figures 7 and 8 for example). It is then linked to certain ports of the agent that called `find_instance`.

```
function find_instance(i)
   j=is_free_instance(i)
   if j=NULL and nb_inst(i) < max_inst
         then nb_inst(i) = nb_inst(i) + 1
              create new instance of Si
              s.t. Si.out is linked to p.i.in
                   Si.in is linked to p.nb_inst(i).i.out
              return(nb_inst(i))
         else return(j)
```

A.2 Coordinator For The Split Primitive

```
coordinator for split(S1,...,Sn)

S1...Sn: sub-agents
St_M: store manager worker
S0: sup-agent

ports: p.0.in ... p.n.in
       % for 0<i<n+1 p.i.in is linked to p.Si.out of Si
       % p.0.in is linked to p.S0.out of S0

       p.0.1.out ... p.0.n.out
       % for 0<i<n+1 p.0.i.out is linked to p.Si.in of Si
       % p.0.out is linked to p.S0.in of S0

       p.stm.in, p.stm.out
       % p.stm.in is connected to the output port of St_M
       % p.stm.out is connected to the input port of St_M

       p.queing.in
       % port used for queuing messages that cannot be
       % sent to an agent
       % (when all instances of the agent are busy)

task c from p.0.in:      % solving request from the
                         % superior agent
      for i from 1 to n
           j=find_instance(i)
           if j<>NULL
                 then send c to p.i.j.out
                 else send (c,i) to p.queing.in

task (c,i) from p.queing.in % the request is sent again
                            % and again till an instance
                            % of Si is free
      j=find_instance(i)
      if j<>NULL
            then send c to p.i.j.out
```

```
            else send (c,i) to p.queing.in

task c from p.i.in:
      send (c,i) to p.stm.out

task end from p.i.in:
      send (end,i) to p.stm.out

task c from p.stm.in:
      send c to p.0.out

task end from p.stm.in:
      send end to p.0.out
```

The store manager *St_M* receives all disjuncts returned by the sub-agents S_1, ... ,S_n, one at a time. As soon as it has enough information (i.e., at least one disjunct from each S_i), a new element of the Cartesian product is built and sent to the split coordinator. *St_M* knows that it has received all solutions when it has collected all of the (end, i) messages. Thus, after solution re-composition is complete, it sends an *end* message to the coordinator. The solutions sent by *St_M* are forwarded by the coordinator to the superior agent. We point out that as soon as *St_M* produces a solution, it is sent to the fix-point coordinator, and then forwarded to the superior agent. In the current implementation, all disjuncts are collected before recreating solutions. This is due to the fact that the coordination and the store management are performed by only one ECLiPSe process. Thus both tasks cannot be performed simultaneously.

A split coordinator never receives several disjuncts from its superior agent. This latter duplicates the split collaboration if it needs several resolutions.

p.queuing.in is a special port not linked to a communication channel. It is used for queuing requests that cannot be sent to a sub-agent, when all instances of this latter are busy. Thus, the split coordinator will try later to send the request to a sub-agent. This method avoids delaying the sending of a request to the other sub-agents.

A.3 Coordinator For The ψ_don't-care Primitive

```
coordinator for dc(S1,...,Sn)

S1...Sn: sub-agents
psi: psi function worker
S0: sup-agent

ports: p.0.in ... p.n.in
       % for 0<i<n+1 p.i.in is linked to p.Si.out of Si
       % p.0.in is linked to p.S0.out of S0

       p.0.1.out ... p.0.n.out
       % for 0<i<n+1 p.0.i.out is linked to p.Si.in of Si
       % p.0.out is linked to p.S0.in of S0

       p.psi.in, p.psi.out
       % p.psi.in is connected to the output port of Psi
       % p.psi.out is connected to the input port of Psi

       p.queuing.in
       % port used for queuing messages that cannot be
       % sent to an agent
       % (when all instances of the agent are busy)
```

```
task c from p.0.in:      % solving request from the
                         % superior agent
      for i from 1 to n
           j=find_instance(i)
           if j<>NULL
                 then send c to p.i.j.out
                 else send (c,i) to p.queuing.in

task (c,i) from p.queuing.in  % the request is sent again
                              % and again till an
                              % instance of i is free
      j=find_instance(i)
      if j<>NULL
            then send c to p.i.j.out
            else send (c,i) to p.queuing.in

task c from p.i.in:
     send c to p.psi.out
     wait for message on.psi.in
     if message=true  % Si sent a solution verifying Psi
        then send c to p.0.in
             stop all Sj, except Si
             enter state.i
        else stop Si
             remove messages of Si from the queue
             if S1...Sn are stopped % no sub-agent is
                                    % satisfactory
                then send fail to p.0.out
                     % the collaboration fails


States % n states state.i for 0<i<n+1

state.i
      task c from p.i.in
            send c to p.0.out

      task end from p.i.in
            send end to p.0.out
            empty message queue
            enter normal state
```

In a given state, tasks related to that state are treated, but unrelated tasks are not consumed. They remain in the task queue, and will be treated later, when entering a state that they are related to (except if the queue is emptied).

The ψ_don't care coordinator does not duplicate the `psi` worker [11] and waits for its solution. This is intentional: the formal model specifies that the chosen agent is the first one returning a solution that satisfies ψ. Thus, duplicating `psi` could lead to the following undesirable configuration: an agent sends a constraint to the coordinator; its validation by an instance of `psi` is currently under way; another agent sends a constraint to the coordinator; its validation by another instance of `psi` is quicker; thus, the coordinator receives its validation before that of the first agent; finally, the second agent is chosen,

[11] The `psi` worker is the worker that applies the ψ function to constraints. It determines whether or not a given constraint satisfies the properties of ψ.

although it should have been the first one.

A.4 Coordinator For The Fix-point Primitive

```
coordinator for f_p(S)

S: sub-agent
S0: sup-agent

ports: p.in       % p.in is linked to p.S.out of S
       p.sup.in   % p.S0.in is linked to p.S0.out of S0

       p.j.out    % for 0<j<max_inst is linked to p.S.in
       p.sup.out  % p.S0.out is linked to p.S0.in of S0

       p'.j.out   % for 0<j<max_inst ports for storing
                  % the constraints sent to p.j.out

       p.queuing.in
       % port used for queuing messages that cannot be
       % sent to an agent
       % (when all instances of the agent are busy)

task c from p.sup.in:      % solving request
                           % from the superior agent
      j=find_instance(S)
      if j<>NULL
            then instance j of S is Running
                 send c to p.j.out
                 send c to p'.j.out
            else send c to p.queuing.in

task c from p.queuing.in  % the request is sent again
                          % and again till
                          % an instance of i is free
      j=find_instance(S)
      if j<>NULL
            then instance j of S is Running
                 send c to p.j.out
                 send c to p'.j.out
            else send c to p.queuing.in

task c from p.j.in & c' from p'.j.in:
      if c=c'
            then send c to p.sup.out
            else k=find_instance(S)
                 if k<>NULL
                       then instance k of S is Running
                            send c to p.k.out
                            send c to p'.k.out
                       else send c to p.queuing.in

task c from p.j.in:
      k=find_instance(i)
      if k<>NULL
            then instance k of S is Running
```

```
                 send c to p.i.k.out
                 send c to p'.i.k.out
            else send (c,i) to p.queuing.in

task end from p.j.in:
      instance j of S is Sleeping
      if all instances of S are Sleeping
            then send end to p.sup.out
```

In order to test whether an agent reaches a fix-point, all requests for solving a constraint `c` are also sent to special ports. When a solution is received from an agent, we test whether it is equal to the sent constraints. For this purpose, we use a task attached to two messages: we read two messages from two ports (one from an agent, and the other one from the "memory" of the input to this agent).

`task c from p.j.in`: this task results from a disjunct coming from the instance `j` of S that has already been treated by the task `task c from p.j.in & c' from p'.j.in`. This means that the constraint `c'` from `p'.j.in` has already been consumed. Thus, we cannot verify whether we have reached a fix-point. However, this does not cause any problem: if an agent returns several disjuncts, this means that its input is different from its output, and a fortiori the fix-point has not been reached.

The end of the collaboration is reached when all instances of S are sleeping. An *end* message is then sent to the superior agent.

A.5 Coordinator For The Repeat Primitive

```
coordinator for rep(delta,S)

S: sub-agent
S0: sup-agent

ports: p.in       % p.in is linked to p.S.out of S
       p.sup.in   % p.S0.in is linked to p.S0.out of S0

       p.j.out    % for 0<j<max_inst is linked to p.S.in
       p.sup.out  % p.S0.out is linked to p.S0.in of S0

       p'.j.out   % for 0<j<max_inst ports for storing
                  % the counter (current number of
                  % iterations) associated to the
                  % constraint sent to p.j.out

       p.queuing.in
       % port used for queuing messages that cannot be
       % sent to an agent
       % (when all instances of the agent are busy)

task c from p.sup.in:      % solving request
                           % from the superior agent
      n=delta(c)
      if n >= 0
           then j=find_instance(S)
                if j<>NULL
                      then instance j of S is Running
                           send c to p.j.out
                           send 1 to p'.j.out
                           % first resolution of
                           % c with S
                      else send (c,1) to p.queuing.in
```

```
           else send c to p.sup.out
                send end to p.sup.out

task (c,n') from p.queuing.in  % the request is sent
                               % again and again till
                               % an instance of i
                               % is free
      j=find_instance(S)
      if j<>NULL
            then instance j of S is Running
                 send c to p.j.out
                 send n' to p'.j.out
            else send (c,n') to p.queuing.in

task c from p.j.in & n' from p'.j.in:
      send n' to p'.j.in:
      if n=n'
            then send c to p.sup.out
            else n'=n'+1
                 k=find_instance(S)
                 if k<>NULL
                       then instance k of S is Running
                            send c to p.k.out
                            send n' to p'.k.out
                       else send (c,n') to p.queuing.in

task end from p.j.in:
      instance j of S is Sleeping
      if all instances of S are Sleeping
            then send end to p.sup.out
```

As soon as a constraints `c` is received from the superior agent, the number of iterations is computed (`n=delta(c)`). Then, `S` must be applied `n` times on `c`, i.e., the depth of the solving tree will be `n`. Each time a constraint is sent to a sub-agent, the number of times `S` has been applied to it is also sent to a special port.

In order to test whether we reached the required depth for a given constraint, we use a task attached to two messages: one from a sub-agent, and the other one from the "memory" port associated to this agent. This task, `task c from p.j.in & n' from p'.j.in`, results from a disjunct `c` (coming from the instance `j` of `S`) that have already been treated `n'` times. If `n'=n`, the disjunct `c` is forwarded to the superior agent, otherwise `c` is sent again to an instance `k` of `S`, and `n'+1` is sent to the "memory" port associated with the instance `k`.

Notice that in all cases, `n'` is re-sent to the "memory" port associated with the instance `j` of `S`: indeed, `S` can produce several disjuncts, and each of them is at the same depth.

The end of the collaboration is reached when all instances of S are sleeping, i.e., none of them is working and the depth of each branch of the solving tree is `n`. An *end* message is then sent to the superior agent.

The queuing mechanism stores a constraint together with the number of times it has already been solved with `S`.

We can consider this coordinator as a transformation of coordinators: as soon as `n` is computed, the coordinator `repeat(delta,S)` can be replaced by the coordinator `seq`$(\underbrace{S,\ldots,S}_{n \text{ times}})$. However, this can lead to a system that may be less easy to trace.

A.6 Coordinator For The Conditional Primitive

```
coordinator for if(gamma,S1,S2)

S1: sub-agent     % ``then part'' of the collaboration
S2: sub-agent     % ``else'' part of the collaboration
S0: sup-agent

ports: p.1.in     % p.1.in is linked to p.S1.out of S1
       p.2.in     % p.2.in is linked to p.S2.out of S2
       p.sup.in   % p.S0.in is linked to p.S0.out of S0

       p.0.1.out  % p.0.1.out is linked to p.S1.in of S1
       p.0.2.out  % p.0.2.out is linked to p.S2.in of S2
       p.sup.out  % p.S0.out is linked to p.S0.in of S0

       p.queuing.in
       % port used for queuing messages that cannot be
       % sent to an agent
       % (when all instances of the agent are busy)

task c from p.sup.in:      % solving request
                           % from the superior agent
      cond=gamma(c)
      if cond
           then j=find_instance(S1)
                if j<>NULL
                      then send c to p.j.1.out
                      else send (c,1) to p.queuing.in
           else j=find_instance(S2)
                if j<>NULL
                      then send c to p.j.2.out
                      else send (c,2) to p.queuing.in

task (c,1) from p.queuing.in  % the request is sent
                              % again and again till
                              % an instance of S1
                              % is free
      j=find_instance(S1)
      if j<>NULL
            then send c to p.j.out
            else send (c,1) to p.queuing.in

task (c,2) from p.queuing.in  % the request is sent
                              % again and again till
                              % an instance of S2
                              % is free
      j=find_instance(S2)
      if j<>NULL
            then send c to p.j.out
            else send (c,2) to p.queuing.in

task c from p.i.in            % i is either 1 ou 2
                              % depending on the
                              % evaluation of gamma
      send c to p.sup.out
```

```
task end from p.i.in:
      send end to p.sup.out
```

This coordinator is the simplest one. When the superior agent sends a constraints `c` to the `if` coordinator, it evaluates γ on `c` (`cond=gamma(c)`). If `cond` is true, then `c` must be forwarded to `S1`, and otherwise to `S2`. Thus, as soon as a solution is received (`c from p.i.in`, either from `S1` if `cond` is true, or from `S2`), it is forwarded to the superior agent.

Since `cond=gamma(c)` is evaluated once, the queuing mechanism must store whether `S1` or `S2` is to be used.

We can consider another version of the conditional coordinator without queuing ports: waiting messages can be re-sent to `p.sup.in`. But, in order to get homogeneous messages on `p.sup.in`, only the constraint can be re-sent, which leads to the disadvantage that `cond=gamma(c)` will have to be executed several times.