

DisChoco: A platform for distributed constraint programming

Redouane Ezzahir,¹ Christian Bessiere,² Mustapha Belaiassaoui,³ and El Houssine Bouyakhf¹

¹ LIMIARF/FSR, U. of Mohammed V Agdal, Maroc
{bouyakhf,ezzahir}@fsr.ac.ma

² LIRMM (CNRS / U. of Montpellier, France
bessiere@lirmm.fr

³ ENCG - U. of Hassan I, Maroc
m.belaiassaoui@encg-settat.ma

Abstract. Open-source platforms are very useful for development and experimentation in constraint programming. However, until recently, no such platform existed for distributed constraint programming. This paper presents DisChoco, a platform for distributed constraint programming. DisChoco is a Java library implemented using the Choco solver and simple agent communication infrastructure (SACI). DisChoco can be used for simulation of a multi-agents environment on a single Java virtual machine, or performed in an environment physically distributed for a realistic use. DisChoco takes into account agent with a complex local problem, message loss, message corruption, and message delay. The implementation of DisChoco was made to offer a modular software architecture which accepts extensions easily. This paper presents the software architecture and illustrates how to implement a specific protocol.

1 Introduction

Constraint programming is a general framework that can formalize various problems in AI. Many theoretical and experimental studies have been performed, and various sophisticated centralized solvers have been developed (Ilog Solver, Chip, Choco, Gecode, etc.). Constraint solvers have the advantage that the user can concentrate her effort on the modelling of the problem, letting the solver run with the 'by default' solving characteristics. But if a researcher wants to implement and test her own techniques, black-box solvers are not the adequate tool. Open-source platforms permit to incorporate and test new ideas in constraint programming without the burden of re-implementing from scratch an ad-hoc solver. Some of the existing constraint solvers are open-source: Choco, Gecode, etc. In [9], the distributed constraint satisfaction problem (DisCSP) is formalized as a constraint satisfaction problem in which variables are distributed among multiple automated agents. The agents solve local constraint satisfaction sub-problems and a communication protocol between agents is performed, in order to allow the distributed system converge to a global solution.

Writing distributed applications is difficult because the programmer has to explicitly juggle with many quite different concerns including centralized programming, asynchronous and concurrent programming, distributed structure, fault tolerance security, open computing and others. The distributed constraint solvers known by the DCR community are MELY [6], DiSolver [4] and Frodo [5]. We propose a new distributed solver namely DisChoco. With DisChoco, our goal is to propose an open-source platform in which it is possible to implement easily and simulate as much as possible the different concerns of distributed constraint programming.

DisChoco is a Java library built on top of the Choco Java open-source solver. Communication is performed via the simple agent communication infrastructure (SACI) if the agents are implemented on distant machines. Otherwise (simulation) the communication is performed via a local communication simulator. The implementation of DisChoco was made to offer a modular software architecture which accepts extensions easily. DisChoco can be used for simulation of a multi-agents environment on a single Java virtual machine, or performed in an environment physically distributed for a realistic use. Each agent in the environment is executed asynchronously in a separate execution thread, and communicates with its peers through message exchange. DisChoco takes into account agent with a complex local problem, message loss, message corruption, and message delay. In this paper, we present the software architecture of DisChoco and the structure of DisChoco agents. We illustrate how to implement a specific protocol. We give experimental results on an ABT implementation.

2 Software Architecture

The DisChoco platform is a Java library implemented using the Choco solver as agent local solver. DisChoco has a communication interface, ChocoCommunication, which makes it possible to implement any mean of communication between local or distant processes. To simulate a multi-agent environment in a single Java virtual machine, each agent must be executed asynchronously in a separate execution thread, and must communicate with its peers through message exchange. For real applications, DisChoco can be performed in an environment physically distributed using the SACI library (Simple Agents Communication Infrastructure [3]). ABT family [1] is the first distributed class of algorithms implemented in this framework.

2.1 User interface

DisChoco is based on Choco, a platform for research in centralized constraint programming and combinatorial optimization. This significant choice of design enabled us to keep the same philosophy of design as Choco and to benefit from the modules already implemented in it. We kept the same instructions notations of programming that were used in Choco. Figure 1 is an example of DisChoco code. This example illustrates the steps of declarations of the objects handled.

```

1. DisProblem DisCSP = new DisProblem ("Example");
2. List infeasPair = {(0,0),(0,3),(1,1)(1,3),(2,0),(2,1)};
3. ChocoAgent ag1 = DisCSP.makeAgent("ABT", "Agent_1");
4. ChocoAgent ag2 = DisCSP.makeAgent("ABT", "Agent_2");

5. Problem p1 = ag1.getLocalProblem();
6. IntVar X0 = p1.makeEnumIntVar("X0", 0, 3);
7. IntVar X1 = p1.makeEnumIntVar("X1", 0, 3);
8. P1.post(p1.neq(X0,X1));
9. P1.post(P1.infeasPairAC( X0, X1, infeasPair ));

10. Problem p2 = ag2.getLocalProblem();
11. DisVar Y0 = p2.makeEnumIntVar("Y0", 0, 3);
12. DisVar Y1 = p2.makeEnumIntVar("Y1", 0, 3);
13. P2.post(p2.neq(Y0,Y1));
14. P2.post(P2.infeasPairAC( Y0, Y1, infeasPair ));

15. DisCSP.post(DisCSP.infeasPairAC(ag1,X0,ag2,Y1,list));
16. DisCSP.post(DisCSP.neq(ag1,X0, ag2, Y1,));

17. DisCSP.solve();

```

Fig. 1. Example of DisChoco code

We start with distributed problem declaration (line 1) followed by the agent declaration which specifies the resolution algorithm to be used (lines 3–4). Next, the declaration of the variables and local constraints of each agent is made the same way as in Choco (lines 5–9 for Agent 1 and lines 10–14 for agent 2). Afterwards, we post the constraints that are external to agents, that is, the constraints that involve several agents (lines 15–16). Finally, we launch the resolution (line 17). We can point out that as in Choco, the constraints declaration and the resolution are clearly separate.

2.2 Object-Oriented model of agent systems

In traditional software development the need for modelling techniques and development methodologies has been recognized for a long time. Several modelling techniques and design methodologies have been developed and used extensively. Among the most successful techniques are the various object-oriented approaches [7]. An object is usually described as a holder of state information together with some behavior using operations upon the state information. Object-oriented methodologies provide support for identifying objects, and allow abstraction via object classes and inheritance via class hierarchies. We have used an object-oriented model for implementing agent systems in the DisChoco

platform. Distributed problems, agents, variables, constraints, and messages, can naturally be represented by objects:

Distributed Problem: All information is encapsulated in a Distributed Problem object (DisProblem) rather than in structures of global data. In the way of an environment physically distributed (several machines) DisProblem objects gather the necessary information for the set of agents performing on other machines.

Agents: Each agent is represented by an Agent object: ChocoAgent. The ChocoAgent object has an identifier that is unique in the system (AgentID), a name (AgentName), a local problem (Choco.Problem), a set of external constraints which connects this agent to the other agents in the system, and a set of properties used in the resolution.

Variables: For each agent, we have two classes of variables: local variables that are already defined in Choco and external variables which we define as a new class, **ExternalVar**, containing external variable knowledge. ExternalVar models a variable belonging to another agent, constrained with the agent. Each variable has a unique identifier VarID that is given by the pair (AgentID, index) where index is the variable index in the local problem and AgentID is the identifier of the agent that owns it. The ExternalVar class implements Event interface to perform propagation events: it propagates instantiations or value removal events when they occur.

Constraints: We have defined a new class of constraints: ExternalConstraint. The ExternalConstraint represents the external links (inter-agent constraints) between local variables and External variables. The ExternalConstraint class implements the Listener interface for receiving ExternalVar modification events and performing constraint propagation. Each ExternalConstraint is recorded on any ExternalVar involving it.

Agent view: The view of an agent is modeled with an object: AgentView. The AgentView class gathers the set of external variables and implements some methods that allow handling the view of the agent.

Communication: To define a communication system for distributed resolution, we have implemented an interface: **ChocoCommunication**. The ChocoCommunication interface defines the necessary methods for managing message communication: sendMessage, receiveMessage and broadcastMessage.

Message: All the types of messages sent in the system may be implemented by the class ChocoMessage and extended from SACI.Message. ChocoMessage defines the necessary methods to handle messages: getSenderID, getReceiverID, getLogicalTimeCounter, getMessageType (Value, Nogood, Heuristic,...).

MailerAMDS : The communication system is represented by MailerAMDS class. The MailerAMDS class implements ChocoCommunication interface, serves as a message relay in the system (see Figure 3) and implements an Asynchronous Message Delay Simulator.

DSolver : The resolution search is accomplish and controlled by DSolver object. This entity serves to solve distributed problem. DSolver is inherited from MailerAMDS.

2.3 Model of distributed solver

DisChoco provides two cases of usage: simulation and realistic use. Both use a communication system for managing message communication between agents. For the simulation use, the multi-agent system is implemented in a single Java virtual machine environment. Thus, each agent is a simple thread and the communication system is locally implemented. For solution detection of an asynchronous search, DisChoco uses Silaghi et al.'s solution detection that allows to detect solution before quiescence [8].

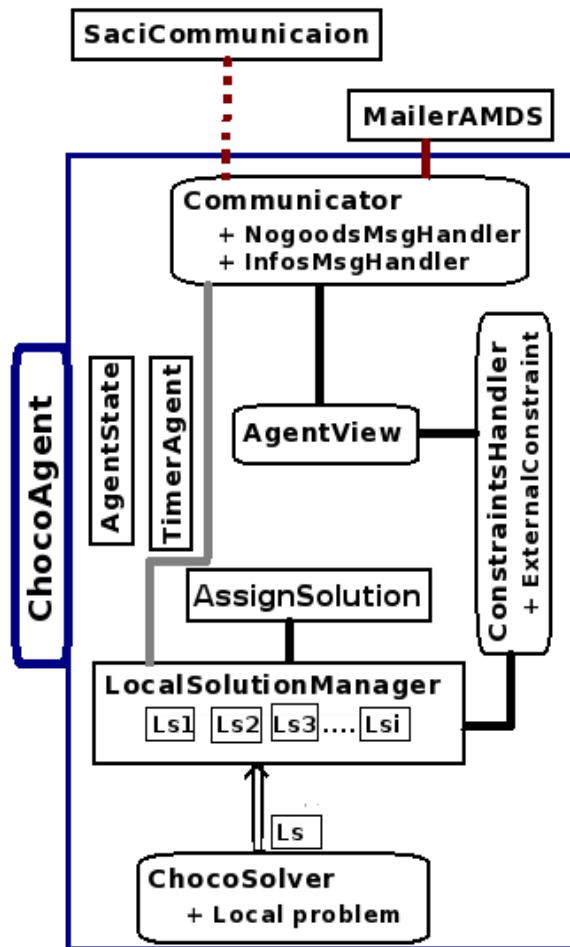


Fig. 2. The overall structure of a DisChoco agent.

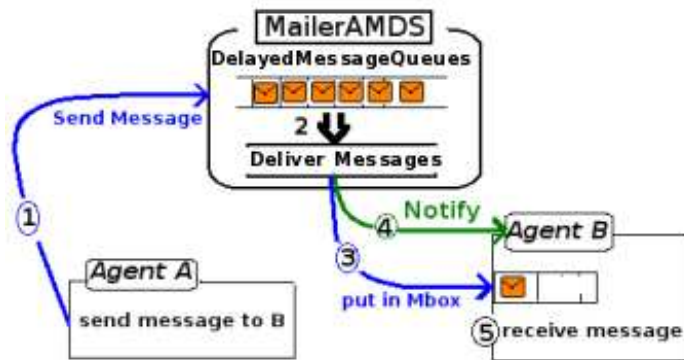


Fig. 3. The messages communication mechanism for simulation use of DisChoco.

Simulation use The user can simulate a distributed protocol within DisChoco. For this case, a communication system is implemented to simulate realistic internet network models. The communication system is represented by MailerAMDS class. The MailerAMDS class implements ChocoCommunication interface, serves as a message relay in the system (see Figure 3) and implements an Asynchronous Message Delay Simulator [11]. For each agent, the MailerAMDS maintains a queue of delayed messages (DelayedMessagesQueues). The delayed messages queue is a priority queue in which the messages are sorted by delivery time (in step). Each agent has a mailbox where messages for that agent are inserted. When a message is put in the mailbox the agent is notified. When an agent (A) wants to send a message to an agent (B), the message exchange between agents is performed as follows:

- Agent (A) is responsible for computing the content of the message. (A) builds a ChocoMessage object that contains a destination ID.
- Agent (A) sends this message over to the MailerAMDS.
- The MailerAMDS reads the message, finds the ID of the intended recipient, assigns to the message a delivery time which is the sum of the current value of the Mailers logical time counter (LTC) and the selected delay (in steps), and places it into the recipient's delayed messages queue that corresponds to intended recipient ID.
- At each step, the MailerAMDS delivers delayed messages with LTC less than or equal to its LTC and notifies concerned agents.

In addition, MailerAMDS is used to check termination condition and to simulate message loss and message corruption. The termination condition occurs when there are no incoming messages, all message queues are empty, and all agents are idle. The MailerAMDS checks if all messages queues are empty. This is a necessary condition for termination. If so, this information is used by the distributed solver (DSolver) to terminate execution of the resolution. To achieve

this goal, each agent has a Boolean variable (`idle_state`), indicating if the agent is in idle state. When the necessary termination condition is detected by MailerAMDS, the DSolver checks if all agents are idle (i.e., `Idle_state=true`). If so, DSolver stops all running threads (Agents) and prints the result. The simulation of messages loss is obtained by calling `MailerAMDS.withMessageLoss()` method, by which we mean that at any point it is possible that a message is not delivered and the recipient agent does not know if this message was sent. The MailerAMDS holds a variable (`MsgLossProbabililty`) to define message loss probability. When it is fixed to a specific value, the MailerAMDS generates a random value and checks if this value is smaller than message loss probability. If so, it deletes the message without delivering it. To generate corrupted messages we have defined for AllMessage object a function `corrupt(): T→T` for each message type T. It is used to modify the content of the message. When the mailer receives a message and its `withMessageCorruption` variable is true, it generates a random value and checks if this value is smaller than the value of message corruption probability (`corruptprob`). If so, the MailerAMDS converts the message to corrupted message by calling the `message.corrupt()` function. The impact of the messages corruption on resolution depends on the algorithm used. The resolution search is accomplish and controlled by DSolver object. This entity serves to solve distributed problem. DSolver is inherited from MailerAMDS. It:

- starts all agents of the system,
- finishes once all agents finished,
- announces solution, statistics.

```

AgentName = Square_5
Algorithm = ABT
DisProblemFactory = dischoco.samples.sudoku
#ProblemFileName = sudoku_5.txt
Comm=saci
#Comm=local
MasterAgent = false
SimpleAgent = true
MasterHostName = 211.44.34.11
NbAgents = 9
ConnectToMasterTimeOut = 300
ResolutionTimeOut = 7200

```

Fig. 4. Example of problem property file

3 Structure of DisChoco Agent

Realistic use DisChoco can be performed in an environment physically distributed using SACI library. The system is controlled by an agent Master that extends saci agent. Each agent has a **SaciCommunication** entity that allows him to communicate with other agents. A Main class has been implemented in DisChoco for loading problem properties file, instantiate agent and start resolution. Figure 4 shows an example of Sudoku problem property file. When Main class is started, it creates a new instance of SaciCommunication using file property. SaciCommunication tries to connect at master host. If it cannot connect in the duration specified by the ConnectToMasterTimeOut argument, a message error is displayed, otherwise the resolution starts. Distributed constraint programming and distributed combinatorial optimization are problems where each agent owns a local problem (variables and constraints) and where some constraints involve variables from several agents. Each agent executes an algorithm to solve the problem. Any algorithm run by an agent handles an AgentView and uses procedures with specific role: constraint handling, message handling, nogood handling, costs handling... These procedures differ from an algorithm to another. The use of virtual methods and dynamic selection of the right function implementing the polymorphic method has many impacts on the architecture of DisChoco. The overall structure of a DisChoco agent is shown in Figure 2. The agents are defined by ChocoAgent objects that gather and define all categories of component handlers:

AgentState: This entity records agent state (running, idle, solution, ...) and some statistic performance parameter (number of concurrent constraint checks).

TimerAgent records start time and time limit of resolution.

ConstraintsHandler: This entity handles constraints. The ConstraintsHandler interface defines a method for initializing the constraints to evaluate and it defines procedures for processing these constraints. (For example Distributed breakout algorithm uses all external constraints to solve the problem; ABT uses only the constraints with higher agents. In addition, constraint processing in DBA is different from that in ABT). Thus, both ABTConstraintsHandler and DBAConstraintsHandler must implement the ConstraintsHandler interface.

MessagesHandler: Modeled by MessagesHandler interface. This interface defines ProcessMessage method to treat communicating message. The user can define the core of this method according to the used resolution protocol. For example in ABT implementation we have implemented this interface with two components: InfosMsgHandler and NogoodsMsgHandler. They respectively process info messages and nogood messages.

Communicator: Defines communication procedure (send, receive and broadcast message), records agents neighbors, and dispatches arrived messages to its corresponding handler and to the TerminationDetector.

TerminationDetector: This class implements MessagesHandler, it processes message if it contains a termination detecting information. For solution detection of an asynchronous search, DisChoco uses Silaghi et al.'s solution detection (see above).

ChocoSolver: This entity defines the local solver. The ChocoSolver class extends thread objects. It controls local solver. The local solver is a Choco solver that is responsible of local problem resolution and it reports local solutions to the agent. The ChocoSolver can be started, suspended, resumed, and stopped by its owner agent.

LocalSolutionManager: The LocalSolutionManager class extends thread objects and exchanges solution with local solver. The LocalSolutionManager stores reported local solutions and uses interchangeability to avoid redundant work [2].

Main class of program: The `dischoco.Main` class task is to load the environment class, instruct it to load the problem from a file, create a distributed representation of the problem (i.e., create a new instance of `DisProblem` class using a specific Factory class), create the agents for simulation use or a single agent in realistic use, and instruct the environment to start the agents. For example `dischoco.samples.urbdcsp` is a uniform random binary `DisCSP` Factory class.

During asynchronous search, ChocoAgent runs a generic procedure presented in Figure 5. The agent starts its local solver (ChocoSolver) and waits for state of the local problem (Fig. 5, line 2). If not feasible it sends a no solution message and terminates. Otherwise (i.e., one solution is found), the agent starts its LocalSolutionManager that manages reported local solutions from ChocoSolver. Next, the agent starts the distributed search (Fig. 5, line 4). It chooses an instantiation (a local solution), sends it through its outgoing links. After that, the agent waits for arrival messages. In parallel, the local solver and the local solution manager continue to exchange discovered local solutions, so that other solutions are immediately available if needed. The function `waitForArrivalMessages()` (Fig. 5, line 7) is used to set agent in a wait state. When the agent is notified that a message has arrived, the agent suspends its local solver if needed, and calls `DispatchReceivedMessages()` defined in its communicator. This procedure dispatches received messages to all messages handler components.

DisChoco is extensible and powerful enough to allow users to extend it and implement any distributed constraint/optimization algorithm. In this section we present a sample implementation of ABT, the most well-known solving protocol for DisCSPs [10, 1]. The first step for implementing this protocol is to create a new ABT agent class. Our ABT agent must extend `dischoco.search.ChocoAgent` and define the `CheckAgentView()`, `Backtrack()` and `checkAddLink()` procedures. The ChocoAgent class creates a Communicator, an AgentState, a TimerAgent and a local Choco problem, and initiates all implemented components handlers. The Choco problem allows the user to define local variables, local constraints and their propagation procedures. The next step focuses on the components handler implementation. The requested components handler are:

ABTConstraintsHandler: it must extend `AbstractConstraintsHandler` class and define methods for handling constraints. The `ABTConstraintsHandler` processes constraints that are activated by an `AgentView` modification event.

ABTInfosMessagesHandler and **ABTNogoodsHandler** that both must extend `AbstractMessageHandler` class. The `ABTInfosMessagesHandler` processes Infos messages and updates the `AgentView`. The `ABTNogoodsHandler` defines

methods for processing the Nogood messages and some procedures for storing an resolving nogoods.

The final step is to record the `ABTInfosMessagesHandler` and the `ABTNogoodsHandler` in the communicator of the agent.

```

0. start ChocoSolver;
1. WaitForLocalProblemState();
2. if( local problem is not feasible)
   send no solution and terminate;
3. start LocalSolutionManager ;
4. start distributed search;
5. End= false;
6. While (not End)
7.     waitForArrivalMessages();
8.     If(ChocoSolver has not finished)
9.         suspend ChocoSolver;
10.    communicator.DispatchReceivedMessages();
11.    If(ChocoSolver has not finished)
        resume ChocoSolver;

```

Fig. 5. The main procedure running by a `ChocoAgent` in `DisChoco`.

4 Example of Implementation

`DisChoco` is extensible and powerful enough to allow users to extend it and implement any distributed constraint/optimization algorithm. In this section we present a sample implementation of ABT, the most well-known solving protocol for `DisCSPs` [10, 1]. The first step for implementing this protocol is to create a new ABT agent class. Our ABT agent must extend `dischoco.search.ChocoAgent` and define the `CheckAgentView()`, `Backtrack()` and `checkAddLink()` procedures. The `ChocoAgent` class creates a `Communicator`, an `AgentState`, a `TimerAgent` and a local `Choco` problem, and initiates all implemented components handlers. The `Choco` problem allows the user to define local variables, local constraints and their propagation procedures. The next step focuses on the components handler implementation. The requested components handler are:

ABTConstraintsHandler: it must extend `AbstractConstraintsHandler` class and define methods for handling constraints. The `ABTConstraintsHandler` processes constraints that are activated by an `AgentView` modification event.

ABTInfosMessagesHandler and **ABTNogoodsHandler** that both must extend `AbstractMessageHandler` class. The `ABTInfosMessagesHandler` processes `Infos` messages and updates the `AgentView`. The `ABTNogoodsHandler` defines methods for processing the `Nogood` messages and some procedures for storing an resolving nogoods.

The final step is to record the ABTInfosMessagesHandler and the ABTNogoodsHandler in the communicator of the agent.

5 Experiments

We tested DisChoco on uniform random binary DisCSPs. A uniform random binary DisCSP class is characterized by $(\#A, n, d, iC, iT, Cx, C, T)$ where $\#A$ is the number of agents, n is the number of variables, d the number of values per variable, iC the number of intra-agent constraints on each agent, iT the constraint tightness defined as the number of forbidden value pairs on intra-agent constraints, Cx the number of edges in the agents connexion graph, C the number of interagent constraints on each edge, and T the interagent constraint tightness. We have tested DisChoco with 100 problem instances of five different classes of uniform random binary DisCSPs. These classes are:

C1: (4, 16, 8, 6, 28, 6, 5, 28)
 C2: (6, 30, 8, 8, 22, 8, 5, 22)
 C3: (6, 30, 8, 10, 22, 15, 6, 22)
 C4 (6, 48, 4, 18, 6, 15, 7, 6)
 C5 (9, 54, 4, 15, 6, 10, 5, 5)

Figure 6 shows the results without message delay and with message delay simulator. $\#ccks$ represents number of concurrent constraint checks and $\#Tmsgs$ the total number of messages sent in the system. We compare Dischoco both without message delay and with message delay simulator. This simple experiment shows that it is easy to evaluate any implemented protocol with performance measures implemented in Dischoco.

Class	Without Message Delay		With Message Delay	
	$\#ccks$	$\#Tmsgs$	$\#ccks$	$\#Tmsgs$
C1	1522	20	1622	21
C2	3819	36	4006	39
C3	24163	2546	25767	2679
C4	37366	9934	39453	10359
C5	14408	1266	15345	1312

Fig. 6. Simulation results averaged over 100 problems per class.

6 Conclusion

We presented DisChoco, a platform for distributed constraint programming. DisChoco is used for simulation of a multi-agent system in a single Java virtual

machine and can be performed in an environment physically distributed using SACI infrastructure. The platform allows all the agents in the system to execute concurrently, and can be extended to implement any distributed constraint programming method. For simulation use, DisChoco models the realistic communication network by the MailAMDS simulator. The MailAMDS simulator takes into account message loss, message corruption, and message delay. Future work will address efficiency issues, parallel and distributed constraint propagation.

References

1. C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the ABT family. *Artificial Intelligence*, 161:7–24, 2005.
2. D.A. Burke and K.N. Brown.. Applying interchangeability to complex local problems in distributed constraint reasoning. In *Proceedings AAMAS'06 workshop on Distributed Constraint Reasoning*, Hakodate, Japan, 2006.
3. J.F. Hübner and J.S. Sichman. SACI: A simple agent communication infrastructure. <http://www.lti.pcs.usp.br/saci/>, 2005.
4. Y. Hamadi and Y. Chong. Distributed Log-based Reconciliation. In *European Conference on Artificial Intelligence Riva del Garda, Italy*, 2006.
5. A. Petcu. Frodo: a FRamework for Open/Distributed Optimization. Technical Report EPFL:2006/001, LIA, EPFL, CH-1015 Lausanne, <http://liawww.epfl.ch/frodo/>, 2006.
6. Michel Galley. Distributed constraint programming platform using sJavap. <http://cs.fit.edu/Projects/asl/#MELY>.
7. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
8. M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings AAAI'00*, pages 917–922, Austin TX, 2000.
9. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 614–621, 1992.
10. M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.
11. R. Zivan and A. Meisels. Message delay and discsp search algorithms. In *Annals of Mathematics and Artificial Intelligence*, Springer Netherlands, vol. 46, no. 4, pp. 415–439, 2006.