

FRODO:  
a FFramework for Open/Distributed Optimization  
Version 2.5

User Manual

Thomas Léauté      Brammert Ottens  
Radoslaw Szymanek  
*firstname.lastname@epfl.ch*

EPFL Artificial Intelligence Laboratory (LIA)  
<http://liawww.epfl.ch/>

November 27, 2009

# Contents

<b>1</b>	<b>Legal Notice</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>FRODO Architecture</b>	<b>3</b>
3.1	Communications Layer . . . . .	3
3.2	Solution Spaces Layer . . . . .	4
3.3	Algorithms Layer . . . . .	5
3.3.1	DPOP Implementation . . . . .	6
3.3.2	How to Implement Your Own Algorithm . . . . .	6
<b>4</b>	<b>How to Use FRODO</b>	<b>8</b>
4.1	Installation Procedure and Requirements . . . . .	8
4.2	File Formats . . . . .	8
4.2.1	Problem File Format . . . . .	9
4.2.2	Agent Configuration File Format and Performance Metrics .	11
4.3	Simple Mode . . . . .	12
4.4	Advanced Mode . . . . .	12
4.4.1	Running in Local Submode . . . . .	13
4.4.2	Running in Distributed Submode . . . . .	14
4.5	Troubleshooting . . . . .	15

# 1 Legal Notice

FRODO is free software: you can redistribute it and/or modify it under the terms of the GNU Affero General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

FRODO is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

FRODO includes software developed by the JDOM Project (<http://www.jdom.org/>).

# 2 Introduction

FRODO is a Java open-source framework for distributed combinatorial optimization.

This manual describes FRODO version 2.x, which is a complete re-design and re-implementation of the initial FRODO platform developed by Adrian Petcu. For more details on the initial platform, please refer to [11]. FRODO currently supports SynchBB [5], ADOPT [8], DSA [16], DPOP [12], S-DPOP [13], O-DPOP [14], ASO-DPOP [10],  $\mathbb{E}$ [DPOP] [7] and Param-DPOP.

# 3 FRODO Architecture

This section describes the multi-layer, modular architecture chosen for FRODO. The three layers are illustrated in Figure 1; we describe each layer in some more detail in the following subsections.

## 3.1 Communications Layer

The communications layer is responsible for passing messages between agents. At the core of this layer is the `Queue` class, which is an elaborate implementation of a message queue. Queues can exchange messages with each other (via shared memory if the queues run in the same JVM, or through TCP otherwise), in the form of Java `Message` objects. Classes implementing `IncomingMsgPolicyInterface` can

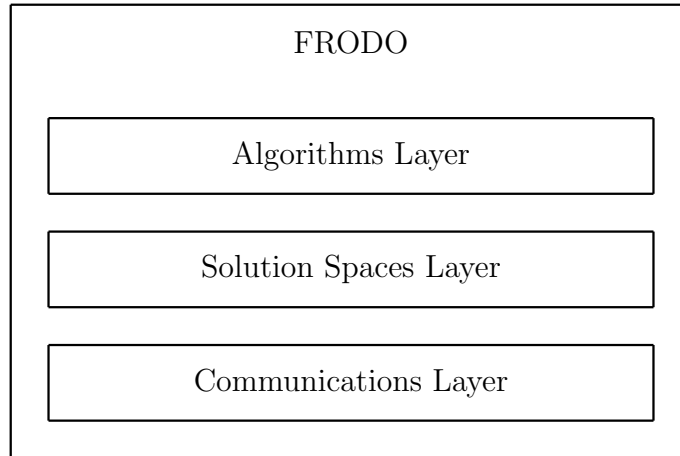


Figure 1: General FRODO software architecture.

register to a queue in order to be notified whenever messages of specific types are received. Such classes can be called *policies* because they decide what to do upon reception of certain types of messages.

Typically, in FRODO each agent owns one queue, which it uses to receive and send messages. Each queue has its own thread, which makes FRODO a multi-threaded framework. Special care has been put into avoiding threads busy waiting for messages, in order to limit the performance implications of having one thread per agent, in experiments where a large number of agents run in the same JVM.

### 3.2 Solution Spaces Layer

FRODO is a platform designed to solve combinatorial optimization problems; the *solution spaces* layer provides classes that can be used to model such problems. Given a space of possible assignments to some variables, a *solution space* is a representation of assignments of special interest, such as assignments that correspond to solutions of a given problem. Intuitively, one can think of a solution space as a constraint or a set of constraints that describes a subspace of solutions to a problem.

In the context of optimization problems, *utility solution spaces* are used in order to describe solutions spaces in which each solution is associated with a *utility*. Alternatively, the utility can be seen as a *cost*, if the objective of the problem is to minimize cost rather than maximize utility.

In order to reason on (utility) solution spaces, FRODO implements operations

on these spaces. Examples of operations are the following:

- *join* merges two or more solutions spaces into one, which contains all the solutions in the input spaces;
- *project* operates on a utility solution space, and removes one or more variables from the space by optimizing over their values in order to maximize utility or minimize cost;
- *slice* reduces a solution space by removing values from one or more variable domains;
- *split* reduces a utility solution space by removing all solutions whose utility is above or below a given threshold.

FRODO will provide two implementations of utility solution spaces: a *hypercube* and a *utility diagram* (only hypercubes are currently implemented). A *hypercube* is an extensional representation of a space in which any combination of assignments to variables is allowed, and is associated with a given utility. Infeasible assignments then have to be represented using a negative, very large – effectively infinite – utility. *Utility diagrams* are more efficient than hypercubes when the space of feasible solutions is sparse, because they only represent the feasible solutions, using data structures similar to decision diagrams.

Solution spaces can be a way for agents to exchange information about their subproblems. For instance, in the *UTIL propagation* phase in *DPOP* [12], agents exchange *UTIL messages* that are hypercubes describing the highest achievable utility for a subtree, depending on the assignments to variables in the subtree’s separator. *H-DPOP* [6] uses some form of utility diagrams instead of hypercubes.

### 3.3 Algorithms Layer

The algorithms layer builds upon the solution spaces layer and the communication layer in order to provide distributed algorithms to solve DCOPs. In FRODO, an algorithm is implemented as one or more *modules*, which are simply policies that describe what should be done upon the reception of such or such message by an agent’s queue, and what messages should be sent to other agents, or to another of the agent’s modules. This modular design makes algorithms highly and easily customizable, and facilitates code reuse, simplicity, and maintenance.

FRODO currently supports SynchBB [5], ADOPT [8], DSA [16], DPOP [12], S-DPOP [13]<sup>1</sup>, O-DPOP [14], ASO-DPOP [10], E[DPOP] [7] and Param-DPOP.

---

<sup>1</sup>The warm restart functionality in S-DPOP is currently only available through the API; see `ch.epfl.lia.frodo.algorithms.dpop.restart.test.TestSDPOP.java` for a sample use.

Param-DPOP is an extension of DPOP that supports special variables called *parameters*. Contrary to traditional *decision variables*, the agents do not choose optimal assignments to the parameters; instead, they choose optimal assignments to their decision variables and output a solution to the parametric DCOP that is a function of these parameters. Section 3.3.1 gives some more details on how DPOP is implemented in the context of FRODO's modular architecture. Section 3.3.2 also explains how other algorithms can be implemented using a similar approach.

### 3.3.1 DPOP Implementation

A DPOP agent uses a single queue, and is based on the generic, algorithm-independent `SingleQueueAgent` class. The behavior of this generic agent is specialized by plugging in four modules, which correspond to DPOP's four phases.

1. The *Variable Election* module describes how the agent should exchange messages with its neighboring agents in order to collectively elect one variable of the overall problem. This is implemented following P-DPOP's naive variable election algorithm [3].
2. The *DFS Generation* module has the agents exchange tokens so as to order all variables in the DCOP along a DFS tree, rooted at the variable elected by the Variable Election module, and following P-DPOP's DFS generation algorithm [3].
3. The *UTIL Propagation* module implements DPOP's traditional *UTIL propagation* phase [12], during which hypercubes describing solutions to increasingly large subproblems are aggregated and propagated along the DFS tree in a bottom-up fashion, starting at the leaves of the tree.
4. The *VALUE Propagation* module corresponds to DPOP's *VALUE propagation* phase [12], which is a top-down propagation of messages containing optimal assignments to variables.

This modular algorithm design makes it easy to implement various versions of DPOP, either by parametrizing one or more modules to make them behave slightly differently, or by completely replacing one or more modules by new modules to implement various behaviors of the agents.

### 3.3.2 How to Implement Your Own Algorithm

This section gives a few tips on how to implement your own algorithm in FRODO.

**Choose a modular design.** Modularity is and must remain one of the strong points of FRODO. When considering implementing a new algorithm, first think carefully about possible phases of the algorithm, which should be implemented in separate modules/policies if possible. For instance, the choice of implementing DPOP’s Variable Election and DFS Generation phases in modules that are separate from the two other phases is strategic, because these two modules can be reused without modifications for the ADOPT algorithm [8], which also works on DFS trees.

**Are there known variants?** Before starting the implementation of the algorithm, you should put your chosen modular design to the test, by judging how easy or hard it would be to modify it to implement known variants of the algorithm. If two variants of the algorithm only differ by a small detail, you might want to implement this simply as an input switch parameter to the corresponding module. For instance, in FRODO’s implementation of DPOP, the heuristic to be used when electing root variables and generating the DFS is specified as a parameter (Figure 3). However, if the two versions are more different, but only on one particular routine that could be considered a “phase,” you should try to implement this phase in a separate module, so that the implementations of the two algorithm versions differ only on whether one module is used instead of another. For instance, the implementation of O-DPOP [14] differs from that of DPOP by the use of two different modules for UTIL Propagation and VALUE propagation.

**Implement systematic JUnit tests.** Another important strength of FRODO is that it is systematically, thoroughly tested using JUnit tests. As soon as you have completed a first implementation of a module (or, ideally, even before you start implementing it), write JUnit tests to make sure it behaves as expected on its own. FRODO being an intrinsically multi-threaded framework, you should use repetitive, randomized tests whenever it makes sense to do so. Once all modules are assembled together and the algorithm is completed, write unit tests against other algorithms that have already been implemented, to check that the outputs of the algorithms are consistent.

**A few implementation details.** Here are a few more details that are useful to know when implementing an algorithm.

- `SingleQueueAgent` tells its modules to start the algorithm by sending a message of type `AgentInterface.START_AGENT`; your algorithm’s module(s) should listen for this type of messages to detect when to start the algorithm.

- When the algorithm has finished, one module should send a message of type `AgentInterface.AGENT_FINISHED` to itself. This message will be caught by `SingleQueueAgent`.
- If you want one of the modules to report statistics or other information about its current state to the centralized controller, the module should extend the `StatsReporter` interface instead of `IncomingMsgPolicyInterface`. Refer to the implementation of DPOP's `DFSgeneration` module for an example on how this can be done.

## 4 How to Use FRODO

This section describes how to use FRODO to solve Distributed Constraint Optimization Problems (DCOPs).

### 4.1 Installation Procedure and Requirements

FRODO is distributed in a compressed a ZIP file, which, when expanded, contains the following five elements:

- `frodo2.jar` is a Java 1.5-compliant executable JAR file;
- `LICENSE.txt` contains the FRODO license;
- `RELEASE_NOTES.txt` summarizes changes made from version to version;
- `FRODO User Manual.pdf` is this user manual;
- A `lib` folder. FRODO depends on the JDOM (version 1.1) third-party library that must be downloaded separately from its distributor [2] and put in the `lib` folder.

### 4.2 File Formats

FRODO takes in two types of files: files defining optimization problems to be solved, and configuration files defining the nature and the settings of the agents (i.e. the algorithm) to be used to solve them.

```

<instance>
  <presentation name="sampleProblem" maxConstraintArity="2"
    maximize="false" format="XCSP 2.1_FRODO" />

  <domains nbDomains="1">
    <domain name="three_colors" nbValues="3">1..3</domain>
  </domains>

  <variables nbVariables="3">
    <variable name="X" domain="three_colors" owner="agentX" />
    <variable name="Y" domain="three_colors" owner="agentY" />
    <variable name="Z" domain="three_colors" owner="agentZ" />
  </variables>

  <relations nbRelations="1">
    <relation name="NEQ" arity="2" nbTuples="3" semantics="soft" defaultCost="0">
      1: 1 1|2 2|3 3
    </relation>
  </relations>

  <constraints nbConstraints="3">
    <constraint name="X_and_Y_have_different_colors" arity="2" scope="X Y" reference="NEQ" />
    <constraint name="X_and_Z_have_different_colors" arity="2" scope="X Z" reference="NEQ" />
    <constraint name="Y_and_Z_have_different_colors" arity="2" scope="Y Z" reference="NEQ" />
  </constraints>
</instance>

```

Figure 2: An example FRODO XCSP file.

#### 4.2.1 Problem File Format

The file format used to describe DCOPs is a subset of the XCSP 2.1 format [9], with two small extensions necessary to describe which agent owns which variable, and whether the problem is a maximization or a minimization problem, as the XCSP format was designed for centralized CSPs and WCSPs, not distributed optimization problems. Figure 2 shows an example FRODO XCSP file.

A FRODO XCSP file consists of four main sections:

1. The `<domains>` section defines domains of values for the variables in the DCOP. There need not be one domain per variable; several variable definitions can refer to the same domain.
2. The `<variables>` section lists the variables in the DCOP, with their corresponding domains of allowed values, and the names of the agents that own them. One agent may own more than one variable. This `owner` field is an extension made to the XCSP 2.1 format [9] in order to adapt it to distributed problems.
3. The `<relations>` section defines generic *relations* over variables. A relation is to a *constraint* what a domain is to a variable: it describes a generic

```

<agentDescription className = "ch.epfl.lia.frodo.algorithms.SingleQueueAgent"
  measureTime = "false" measureMsgs = "false" >

  <modules>
    <module className = "ch.epfl.lia.frodo.algorithms.dpop.VariableElection"
      nbrSteps = "150" >
      <idFactory
        className = "ch.epfl.lia.frodo.algorithms.dpop.VariableElection$MostConnectedHeuristic" />
      </module>

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.DFSgeneration"
      reportStats = "true" >
      <heuristic
        className = "ch.epfl.lia.frodo.algorithms.dpop.DFSgeneration$MostConnectedHeuristic" />
      </module>

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.UTILpropagation"
      reportStats = "true"
      utilClass = "ch.epfl.lia.frodo.solutionSpaces.AddableInteger"
      countNCCCs = "false" />

    <module className = "ch.epfl.lia.frodo.algorithms.dpop.VALUEpropagation"
      reportStats = "true" />
  </modules>
</agentDescription>

```

Figure 3: Example of a FRODO agent configuration file, corresponding to the classical version of DPOP.

notion over a certain number of variables, without specifying the names of the variables. This notion can then be implemented as constraints on specific variables.

Among all possible types of relations that are defined in the XCSP format, FRODO currently only supports the *soft* relations (`semantics = "soft"`), which list possible utility values, and for each utility, the assignments to the variables that are associated with this utility. In the example in Figure 2, the binary relation assigns the utility value 1 to all assignments in which the two variables are equal, and a utility value of 0 to all other assignments (as specified by the `defaultCost` field). This example relation is essentially a soft relation representation of the hard inequality relation. Support for other types of relations will be provided in later releases of FRODO.

4. The `<constraints>` section lists the constraints in the DCOP, by referring to previously defined relations, and applying them to specific variable tuples.

### 4.2.2 Agent Configuration File Format and Performance Metrics

FRODO takes in an agent configuration file that defines the algorithm to be used, and the various settings of the algorithm's parameters when applicable. Figure 3 presents a sample agent configuration file.

**Performance Metrics** FRODO supports the following performance metrics:

- **Number of messages and amount of information sent:** To activate this metric, set the attribute `measureMsgs` to `"true"` in the agent configuration file. FRODO then reports the total number of messages sent, sorted by message type, as well as the total sum of the sizes of all messages sent, also sorted by message type. Note that this can be computationally expensive, as measuring message sizes involves serialization.

**NOTE:** The Variable Election module exchanges a number of messages that is linear in the parameter `nbrSteps`. For optimal results, this parameter should be set to a value just above the diameter of the largest connected component in the constraint graph. A good rule of thumb is to set it to a value just above the total number of variables in the DCOP.

- **Simulated time** [15]: When the attribute `measureTime` is set to `"true"`, FRODO uses a centralized mailman to ensure that messages are delivered sequentially one at a time, in increasing order of their time stamps.
- **Non-Concurrent Constraint Checks (NCCCs)** [4]: To activate the counting of NCCCs, set the attributes `countNCCCs` to `"true"` in the relevant module definitions.

**WARNING:** In several algorithms, the computational bottleneck does not lie in checking constraints, in which case *the NCCC metric does not properly reflect the algorithm's distributed runtime*. In particular, this is the case of ADOPT, whose computational bottleneck is in processing contexts, and of DPOP, whose bottleneck is in adding utilities. **Use NCCCs carefully**, or do not use them at all, and favor the (implementation-specific) metric based on *simulated time*.

**Other Statistics** Several algorithmic modules can also report other statistical information about the problem. Whenever applicable, you can set the attribute `reportStats` to `"true"` to get access to these statistics. For instance, in the case of DPOP (Figure 3), the DFS Generation module can report the DFS tree that is computed and used by DPOP, using the DOT format [1], while the VALUE Propagation module can report the optimal assignments to the DCOP variables and the corresponding total utility.

### 4.3 Simple Mode

FRODO can be run in two modes: in simple mode, and in advanced mode (Section 4.4). In simple mode, all agents run in the same Java Virtual Machine. The distributed nature of the algorithm is simulated by assigning (at least) one thread per agent. Agents exchange messages by simply sharing pointers to objects in memory.

This mode is launched using the `main` method of the class `AgentFactory` in the package `ch.epfl.lia.frodo.algorithms`. This is defined as the default entry point of `frodo2.jar`, therefore the following command should be used from within the directory containing the FRODO JAR file:

```
java -jar frodo2.jar
```

The method takes in two arguments:

1. the path to the problem file;
2. the path to the agent file.

If the path to the agent file is omitted, FRODO uses the DPOP agent file `DPOPagent.xml` by default. The simple mode supports two options:

- `-license`: FRODO prints out the license and quits.
- `-timeout msec`: sets a timeout, where *msec* is a number of milliseconds. The default timeout is 10 minutes. If set to 0, the timeout is disabled.

### 4.4 Advanced Mode

FRODO's advanced mode can be used to run algorithms in truly distributed settings, with agents running on separate computers and communicating through TCP. In this mode, each computer runs a *daemon*, which initially waits for a centralized *controller* to send the descriptions of the algorithm to use and the problem(s) to solve. The controller is only used during the initial setup phase; once the algorithm is started, the agents communicate with each other directly, and the controller could even be taken offline. In the context of experiments, for the purpose of monitoring the solution process on a single computer, agents can also be set up to report statistics and the solution to the problem(s) to the controller.

Another advantage of using the advanced mode is that it is very easy to set up batch experiments. The configuration file (see Figure 4) can contain a list of problems that will be solved sequentially by the controller. The agent to be

```

<experiment>

  <configuration>
    <resultFile fileName = "resultFile.log"/>
    <agentDescription agentName = "algorithms/dpop/DPOAgent.xml"/>
  </configuration>

  <problemList nbProblem = "2">
    <file fileName = "problem1.xml"/>
    <file fileName = "problem2.xml"/>
  </problemList>

</experiment>

```

Figure 4: Example of a configuration file for FRODO’s advanced mode.

used is defined by the field `agentName` in the `agentDescription` element. It is also possible to replace the `agentName` field with `fileName = "agent.xml"`, where `agent.xml` is the name of a file describing the agent to be used.

FRODO’s advanced mode has two submodes:

- The *local* submode uses only one JVM and a single computer; there is only one daemon, spawned by the controller itself, and all agents run in the controller’s JVM.
- In *distributed* submode, daemons are started by the user in separate JVMs (possibly on separate computers), and wait for the problem and algorithm descriptions from the centralized controller.

**IMPORTANT NOTE:** The advanced mode does not support the simulated time metric (Section 4.2.2).

#### 4.4.1 Running in Local Submode

To run the controller in local submode, the `Controller` class in the package `ch.epfl.lia.frodo.controller` must be launched with the argument `-local`, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.controller.Controller -local
```

As an optional argument, one can set the work directory by giving the argument `-workdir path`. The default work directory is the one from where the controller is launched.

When the controller is launched, a simple console-based UI is started. To load a particular configuration file, one uses the `open` command:

```
>open configuration_file
```

This command tells the controller to load the configuration file that contains all the information necessary to run the experiments. A sample configuration file can be found in Figure 4. To run the experiments, simply give the command **start**. When all the experiments are finished, the controller can be exited by giving the **exit** command.

#### 4.4.2 Running in Distributed Submode

To run the controller in distributed submode, the `Controller` class in the package `ch.epfl.lia.frodo.controller` must be launched, without the `-local` option, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.controller.Controller
```

To set the work directory one can again use the `-workdir` argument. When running in distributed mode, the controller assumes that the agents must be run on a set of daemons. These daemons can run on the same machine or on different machines. To start a daemon, open a new console, and launch the `Daemon` class in the package `ch.epfl.lia.frodo.daemon`, using the following command from within the directory containing `frodo2.jar`:

```
java -cp frodo2.jar ch.epfl.lia.frodo.daemon.Daemon
```

The IP address of the controller can either be given with the command-line argument `-controller ip_address`, or by issuing the command in the daemon console:

```
>controller ip_address
```

The port number used for the controller is 3000. When all the daemons are running, one can check whether they are correctly registered to the controller by using the following command in the controller console:

```
>get daemons
```

The configuration file can be loaded by the `open` command and the experiments started by using the `start` command. When the experiments are started, the agents are assigned to the different daemons in a round robin fashion. In the future we intend to allow for more flexibility in assigning agents to particular daemons.

## 4.5 Troubleshooting

If you encounter unexpected errors or exceptions when using FRODO, you might want to pass the option `-ea` to the JVM in order to enable `asserts`. With this option on, FRODO will perform some optional (potentially expensive) tests on its inputs, which can sometimes help resolve problems. The FRODO development team is also open to requests, and can be contacted by email (refer to the contact details on the title page of this document).

## References

- [1] Graphviz – Graph Visualization Software. <http://www.graphviz.org/>.
- [2] The JDOM XML toolbox for Java. <http://www.jdom.org/>.
- [3] Boi Faltings, Thomas Léauté, and Adrian Petcu. Privacy guarantees through distributed constraint satisfaction. In *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'08)*, pages 350–358, Sydney, Australia, December 9–12 2008.
- [4] Amir Gershman, Roie Zivan, Tal Grinshpoun, Alon Grubshtein, and Amnon Meisels. Measuring distributed constraint optimization algorithms. In *Proceedings of the AAMAS'08 Distributed Constraint Reasoning Workshop (DCR'08)*, pages 17–24, Estoril, Portugal, May 13 2008.
- [5] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume 1330, pages 222–236, Linz, Austria, October 29–November 1 1997. Springer.
- [6] Akshat Kumar, Adrian Petcu, and Boi Faltings. H-DPOP: Using hard constraints for search space pruning in DCOP. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI'08)*, pages 325–330, Chicago, Illinois, USA, July 13–17 2008. AAAI Press.
- [7] Thomas Léauté and Boi Faltings. E[DPOP]: Distributed constraint optimization under stochastic uncertainty using collaborative sampling. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*, pages 87–101, Pasadena, California, USA, July 13 2009.
- [8] Pragnesh J. Modi, W Shen, Milind Tambe, and Makoto Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
- [9] Organising Committee of the Third International Competition of CSP Solvers. *XML Representation of Constraint Networks – Format XCSP 2.1*, January 15 2008.
- [10] Brammert Ottens and Boi Faltings. Coordinating agent plans through distributed constraint optimization. In *Proceedings of the ICAPS'08 Multiagent Planning Workshop (MASPLAN'08)*, Sydney, Australia, September 14 2008.

- [11] Adrian Petcu. FRODO: A Framework for Open/Distributed constraint Optimization. Technical Report 2006/001, Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland), 2006.
- [12] Adrian Petcu and Boi Faltings. DPOP: A Scalable Method for Multiagent Constraint Optimization. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 266–271, Edinburgh, Scotland, July 31 – August 5 2005. Professional Book Center, Denver, USA.
- [13] Adrian Petcu and Boi Faltings. S-DPOP: Superstabilizing, fault-containing multiagent combinatorial optimization. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 449–454, Pittsburgh, Pennsylvania, U.S.A, July 9–13 2005. AAAI Press / The MIT Press.
- [14] Adrian Petcu and Boi Faltings. O-DPOP: An algorithm for open/distributed constraint optimization. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI'06)*, pages 703–708, Boston, Massachusetts, U.S.A., July 16–20 2006. AAAI Press.
- [15] Evan A. Sultanik, Robert N. Lass, and William C. Regli. DCOPolis: A framework for simulating and deploying distributed constraint optimization algorithms. In Jonathan P. Pearce, editor, *Proceedings of the Ninth International Workshop on Distributed Constraint Reasoning (CP-DCR'07)*, Providence, RI, USA, September 23 2007.
- [16] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Journal of Artificial Intelligence Research (JAIR)*, 161(1–2):55–87, January 2005.