Joint NCC & IIS Bull., Comp. Science, 16 (2001), 115–127
© 2001 NCC Publisher

A model of cooperative solvers for computational problems

A. Kleymenov, D. Petunin, A. Semenov, and I. Vazhev

The paper presents a model for building cooperative solvers for computational problems. We suggest an architecture of an environment which allows us to implement the model. It consists of a kernel, a library of methods, a scenario language and a universal internal representation. Methods have a special structure that provides their cooperation. We describe the current implementation of this environment, give examples of several schemes of cooperative solvers and present some computational experiments.

1. Introduction

At present, the solvers based on the interval methods of constraint programming [1, 10] are frequently used in solving the applied computational problems. These methods allow us to simplify the statements of problems, to solve the problems in nonstandard statements or with imprecise values, to combine variables of different types, etc. But these methods are not universal, and there exist classes of problems for which they either are not applicable at all, or have low efficiency (for example, the systems of linear algebraic equations). At the same time, the combination of the constraint programming methods with specialized ones allows us to obtain substantial increase in efficiency and power of each method. So, building cooperative solvers that combine different methods and use current computational techniques (parallelism, distributivity, and so on) is a very promising and important direction in the solution of complex computational problems.

There are many works in the field of building cooperative solvers [2, 3, 4, 5, 6, 7, 8]. Most of them either consider the methods to be used for building cooperative solvers, or describe a concrete cooperative solver. From our viewpoint, many solvers and architectures already built are rather limited in capacity for being extended with other methods or use a very rigid computation scheme. So the main purpose of our work is not to build a cooperative solver, but to develop the environment for designing a family of cooperative solvers and the tools for building them. These tools not only do not contradict the methodological approaches already proposed, but also provide a convenient and universal base for their implementation. Using these tools and the model, we have built the prototypes for cooperative solvers which demonstrate that our approach is very promising. In this

paper, we consider further developments of the works described in [9].

In our approach, the base for building a cooperative architecture is a kernel and a set of methods-components. The kernel is a multicomponent system, and its main function is to perform a computational scenario for solving the source model. In addition, the kernel works as a resource and information flow manager and provides the user interface. The computational scenario can be user-defined or a standard one. It, in fact, defines the computational scheme to be used in the process of solution. In the scenario language (script) we describe the set of methods to be used when solving the problem, interaction between them, and, possibly, the tuning parameters. Pipes are used for connection between methods. We consider this approach to be rather flexible and multipurpose for building any solver.

The structure of the paper is as follows. Section 2 gives the main definitions and terms used below. The third section presents the GMACS architecture and the tools which allow us to achieve cooperativity. Section 4 is devoted to the current implementation of our approach. It also contains examples of cooperative solvers and presents some experimental results. Conclusion proposes the fundamental results and discusses plans of future work.

2. The basic definitions

The fundamental notion of our approach is a model. Formally its definition coincides with the common definition of the constraint satisfaction problem. A **model** M is a triple (X, C, D), where X is a set of variables, C is a set of constraints on them, and D is their domains. But in fact we treat a model as a pair of sets: a set of constraints and a set of variables with their current values. In the general case, any of the two components can be empty and any set can contain the variables absent in the other set. The input and output data for solvers and methods are specified in terms of models.

The task for a solver is to solve the source model. Its **solution** is the result of building, from the source model, one or several new models satisfying certain requirements. The requirements can be, for example, to obtain a set of interval variable values of a given maximal width, or the set of constraints of the obtained model to be linear, etc.

In the paradigm of this definition of a solution, a **solver** can be considered as a transformer with a source model at its input and one or several new models at its output. The idea to combine solvers in order to obtain new ones with the required properties seems to be quite natural.

The solver can use only one method, but it can consist of a combination of several methods, as well. A **method** is the least indivisible entity that

possesses the property of making one or several other models from a given one. A semantic relation between the source and new models is also required from the method, though in some special cases it can be absent, for example, when some mutation generator is implemented for genetic algorithms. The basic distinctions between a method and a solver are the use of control structures and work with variables of the environment in the latter, though in the simplest case they may coincide, i.e., the solver can simply call one method.

The connecting link between methods is a pipe, an abstraction of the communication interface. Pipes are used to transmit data between methods. Availability of such a component makes it possible to transmit data and control commands in a natural way. A **pipe** is an entity that has one or several inputs and outputs and can get data at inputs and transmit them to outputs.

It is easy to note that this approach is substantially oriented to modular construction of solvers in the context of agent and distributed architectures. Pipes can incapsulate a transport network protocol, which automatically allows us to port the solver to the distributed architecture. No adaptation of methods and algorithms is needed in this case.

A **computation** scheme is a set of methods, the sequence of their running, means for input-output information exchange between methods, and the pipe structure connecting them.

3. Architecture of the environment for building cooperative solvers

General Module Architecture for building Cooperative Solvers (GMACS) is the environment for building cooperative solvers from separate components, such as methods, pipes, and solvers earlier constructed. To build a solver, we choose the necessary methods, specify their properties and combine them into an integrated computational network, possibly, via pipes. GMACS allows us to implement any model of cooperativity: synchronous, asynchronous, sequential, or distributed.

By cooperativity we mean joint solution of parts (intersecting in the general case) of the source problem by different methods, where each of the methods presents its results to other methods. We do not restrict ourselves to the case, when the methods only exchange the results of solution of their subproblems (final or intermediate). We suppose that the methods can also pass the results of analysis of the source models and some heuristics that may be useful in the process of solution. For example, the module extracting the linear part from the source model can analyze it (it is whether subdefi-

nite, degenerate, or something else) and pass the results of this analysis for choosing the appropriate method to solve the system.

To implement different models of cooperativity, we propose the following structure of a method. A method is the entity with at least one input and one output called principal ones. A model is received through the principal entry of the method, and one or several resulting models are returned through the principal exit. A method can have several additional inputs and outputs. Each input can receive some information, so each output can return some information. Each input and each output can be connected to a pipe that can receive and transmit information.

The basic working cycle of a method consists in receiving a model through the principal input, processing the model, and passing the results to the principal output. When processing the model, the method can read information from additional inputs and pass some intermediate results to the outputs. If no real pipe is connected to the input (output), we consider this input (output) to be connected to the empty pipe which makes no information exchange. When working with additional pipes, the method in the general case absolutely does not take care of the source of this or that information or which method will use its intermediate results. This approach substantially simplifies the development and usage of the methods and combining them into various solvers. The functions of the method, including the set of its additional inputs and outputs, are defined by the method developer at the design stage.

Along with the computational methods which, from the source model, find the values of all or some of its variables, there also exist **pseudomethods** which only perform transformations of the source models. Pseudomethods completely satisfy our definition of the method, but are not methods in the mathematical sense. As an example, we can consider a method that builds the Grbner bases for the polynomial source model or a method that separates the linear and nonlinear parts of a model. As a result, this method returns two models that are solutions of the source model and each of them may be empty. One more example is a unitor performing the reverse operation: it unites several models into one.

The architecture of the proposed system for building cooperative solvers is shown in the picture.

The source model is written in a language of model description. It can be one of well-known and generally used languages (for example, OpenMath or a language of some available system), as well as our own internal language (in particular, our input language). After that, the compiler translates the model from this language into the universal internal representation (IR) which is used below by almost all methods and is a communication means

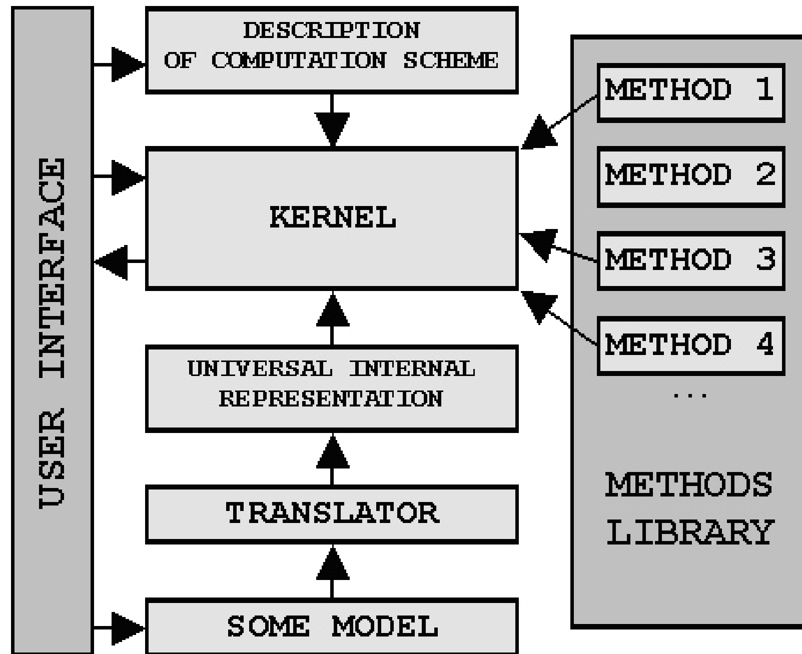


Figure 1. Architecture of the environment for building cooperative solvers

between methods.

The library of methods is a dynamic library of independent components (methods and pseudomethods) used for building the computation scheme. Each method consists of the following components:

- interface is a set of functions to control the method, in particular, a set of available pipes;
- computational algorithm is the computational part of the method;
- debugging is means intended to analyze the work of the method.

Script is a language for specification of the computational scheme. It is used to write down the sequence of calls of methods and connections of inputs and outputs of methods with the pipes, to specify the conditions for termination of computations, to setup global settings of the computational process, etc. The computational scheme so constructed can be considered as a **meta-method**, since it possesses all the features of the method but is not indivisible and uses other methods. In addition to the methods, Script can call meta-methods earlier constructed and include them into the current scheme of computations. The computational scheme so constructed can be

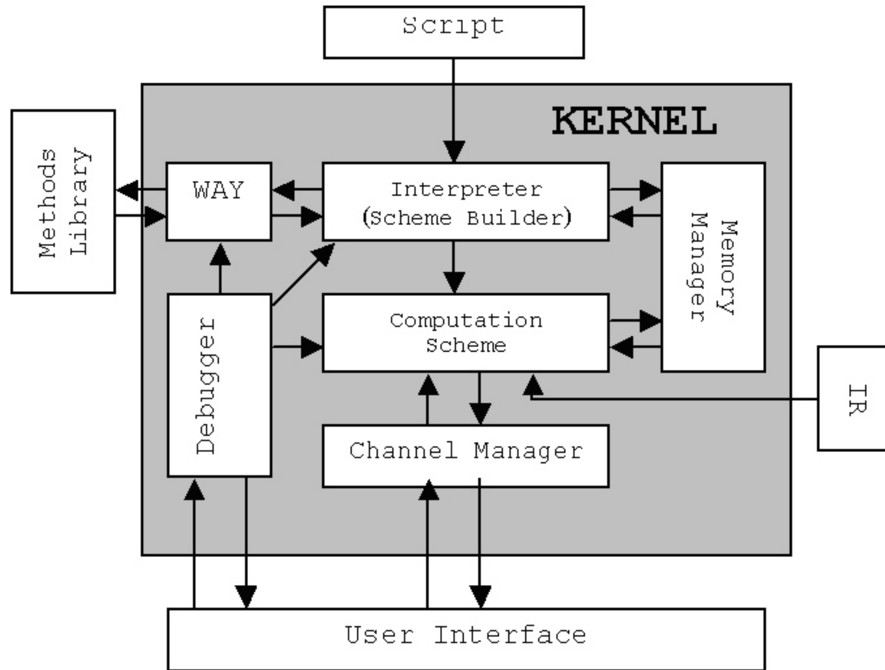


Figure 2. Interaction between the kernel components

written down into the library of meta-methods and used either as one of the default built-in schemes or as a component for building new schemes.

The kernel of the system consists of several components:

- script interpreter is an engine that performs actions prescribed by the script and initiates the computational scheme;
- pipe manager is a process that controls the pipe functioning;
- Memory manager is a system that efficiently control the use of memory;
- who are you? This is a mechanism of inquiry of the components about their properties and capabilities;
- user interface;
- debugging mechanism is a system that controls the computations.

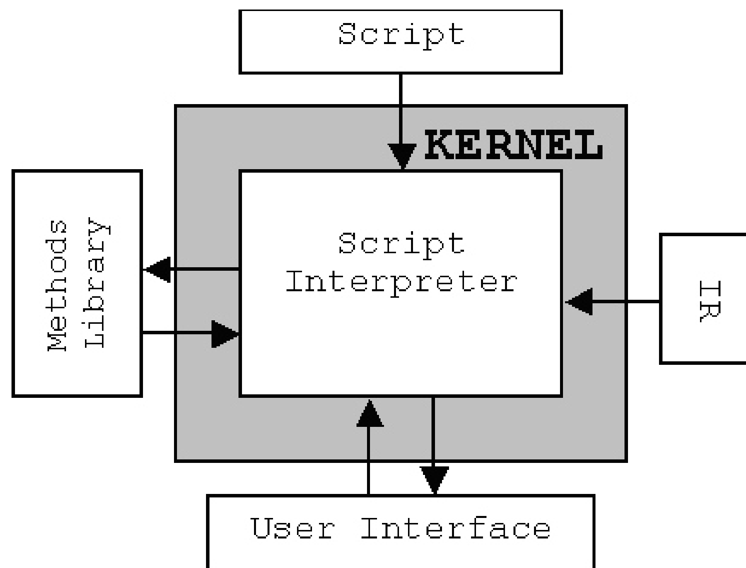
The main function of the kernel is to initiate and run the given computational scheme. Interaction between the kernel components is shown in the picture.

This architecture is easy to extend and upgrade, in particular, it allows us to model most of the existing cooperative systems. To do this it is sufficient to write a compiler from the language of the problem description

and a transformer of the control constructions of the system to be modelled into the script. In some cases it may be necessary to write wrappers for the methods, for example, for the case of using the methods of already available systems.

4. Current implementation

The concept of building cooperative solvers described in the above chapter has been partially implemented within an experimental system based on the constraint programming methods. The current implementation includes a universal internal representation, a library of methods, a script and a kernel. At present, not all components of the kernel are implemented completely. As a result, the kernel of the system looks as shown in the picture:



The internal representation of the model has been specified and implemented. We have a clearly determined interface for building a model in the internal representation and the interface for receiving and changing its data. This interface is used by the methods and the script to call the internal representation.

A large **library of methods** has been implemented. All the methods have common interface, which essentially simplifies the development and introduction of new methods. In particular, the interface supports setting of the initial data, receiving the results, running of the method, etc. Each method has a set of its specific attributes to identify it in the library of methods. There are also varying attributes whose values have effect on the

method functioning. Using them, we can control the process. At present, the library of methods includes the algorithms of constraint programming on finite domains (AC-4 and AC-5) and on continuous domains (interval AC-3), the Newton interval method, the methods for solution of interval systems of linear equations, linear programming methods (an interval extension of the "interior point" method), a wide set of methods for search over continuous and finite domains, automatic differentiation and a number of other methods.

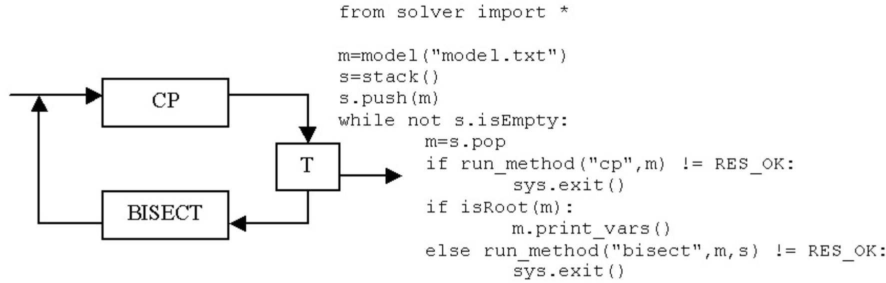
The Python language is used as a **script** in the current implementation. Script performance is supported by the **script interpreter** which, in addition to creation of a computational scheme, provides and controls its functioning. Script explicitly describes which methods and meta-methods are used in it and how they interact. Script can contain all the control structures of Python; it has access to global variables of the environment and can tune computations in the optimal way. The user interface allows the interpreter to be controlled and the computational scheme and the data to be corrected in the interactive mode. Thus, the **debugging** mechanism in this implementation is inseparably linked with the user interface and does not contain any special means.

At this stage of the system development, the notion of the pipe is abstract: data transmission between the methods is the function of the interpreter. The methods can receive at their inputs the output models of other methods, the current values of all or selected variables and other information according to the description of interaction between methods in the script.

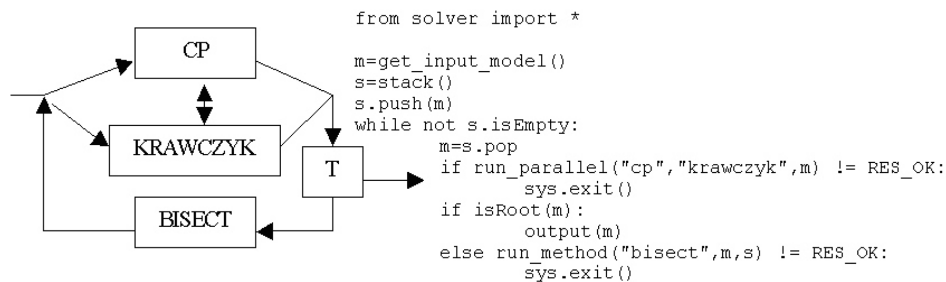
Let us consider some examples of building cooperative solvers with the help of the GMACS architecture.

Example 1. A simple solver with sequential cooperativity. In this solver the source model is received through the input of the method of interval constraint propagation (**ICP**) and its result is passed to the bisection method (**BISECT**). This method divides the obtained model into two submodels with different domains of one or several variables. The two resulting models are again received by **ICP** and the process goes on until all solutions are found. **T** is the method for checking whether the model is a root and, depending on this, passes it to one of the two outputs. Some works on cooperative solvers call this scheme Sequential Evaluation. Below we present the computational scheme and its description in Script.

Example 2. A cooperative solver of nonlinear problems. In principle, it differs from the first one only in that there is an additional **Krawczyk** method for solving the problem asynchronously with the **ICP** method. The **Krawczyk** method implements the Newton interval method and, similar to **ICP**, is a narrowing method. In this scheme the methods use an additional

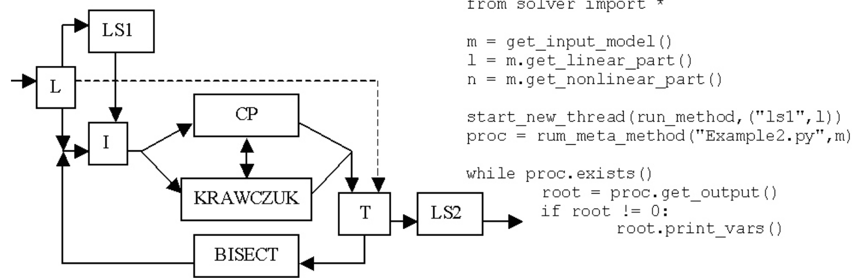


pipe for exchange the intermediate results with each other. If no method can refine the obtained solution, the result is passed to the bisection method which divides the domain in two parts. The corresponding computational scheme and its Script representation are:



Example 3. A cooperative solver for nonlinear problems with a linear part. This example uses a more complex scheme containing the previous solver as its part. The source model is received through the entry of the pseudomethod (**L**) which separates the linear and nonlinear parts of the model. The linear part is sent to the solver of linear equation systems (**LS1**). The nonlinear part of the model is received by the solver from the previous example. In this case, before sending the source model to the nonlinear solver, the domain intersection method (**I**) intersects current values of the variables with the values obtained in the **LS1** method. Note that in this example the bisection method is controlled by the **L** method, namely, the **L** method transmits the set of variables of the nonlinear part via a special pipe. Only these variables will be used in bisection. This pipe also determines the set of variables used by the **T** criterion for finding out if this model is a solution of the nonlinear part of the source model. It can be described as follows.

Two examples of applying our approach to solving the computational problems are given below. We present scenarios in Script describing appropriate computational schemes and give the run time for them.



Problem 1. We consider a model that consists of linear and nonlinear equations. The model has 21 variables and 21 equations. It is possible either to solve the system only by the method of interval constraint propagation or to use a combination of **ICP** and the interval Gauss method. This combination is described by the following scenario.

```
from solver import *
m = model("test1.txt")
while 1:
    tmp_m = m.clone()
    if run_method("gauss",m) != Methres_OK
            or run_method("cp",m) != Methres_OK:
        sys.exit(1)
    if m.compare(tmp_m,eps):
        break
print_vars(m)
```

Problem 2. A global optimization problem (a nonlinear function in 5 variables). We look for the global minimum of the function by splitting the domain of its values from the left to the right and applying **ICP** on subdomains. We need to find the result with a given accuracy (10^{-10}). The first approach is to apply the algorithm with this accuracy. The second approach is to start with a low accuracy in order to narrow the initial domain and then to refine domains iteratively by increasing the accuracy. A scenario for this approach is as follows.

```
from solver import *
m = model("test2.txt")
acc=0.1
while acc > 1e-9:
    if run_method("bisect",m,LEFT,"objf",acc) != Methres_OK:
        print "no solution"
        sys.exit(1)
```

```
    acc *= 0.1
print_vars(m);
```

Here we provide the computational results for these problems. We have performed the experiments on the Athlon-600 processor (time is in seconds).

Problem	Only **ICP**	By scenario
Problem 1	36.0	26
Problem 2	6.7	0.3

5. Conclusion and future work

This paper proposes a model for building cooperative solvers. The emphasis is made on the combination of methods intended to solve the computational problems, but we believe this approach to be applicable to a wider class of problems. The architecture that we have proposed differs from already known ones in some essential points. In particular, we do not restrict the range of methods to be used for building solvers and allow the developer to implement any computational schemes. The availability of pipes for synchronization and transmission of control signals makes it possible to implement mixed asynchronous-synchronous schemes. The possibilities of building the distributed solvers on the basis of this architecture open boundless prospects in using computational capabilities of the cluster systems and local and global networks, too.

Our approach is very convenient for constructing means of solver visual design and can be used in cooperative solver designing and prototyping. Besides, the pipe mechanism allows us to integrate easily the schemes built on the basis of this architecture with GUI, which let us consider this product as a RAD tool for creating computational applications.

We have already implemented a considerable part of this architecture and on this basis a number of cooperative solvers have been built which show the possibility of a substantial increase in computational capabilities of the applied algorithms. The results of experiments also show great potentialities of GMACS as a means for building cooperative solvers.

It is clear that further investigations, experiments and new projects are needed to increase efficiency and create practical applications. In particular, we plan the following work on the development of our approach.

1. Separation between stages of work of the interpreter and computational scheme. In this case there appears an opportunity for more advanced communication system between methods independent of the script describing the corresponding computational scheme.

2. Implementation of pipes and control means for them. In particular, this gives us opportunity to perform the distributed computations.
3. Creation of debugging means for a computational scheme.
4. Creation of a visual design environment.
5. Creation of intellectual constructors for computational schemes using the mechanism "Who are you?"
6. Creation of the mechanism for generation of specialized solvers according to the input script. Such a solver will be completely optimized to the given computational scheme, which allows us to increase the computation efficiency at the cost of decrease in overhead expenses on building, communicating and debugging the scheme.
7. Replenishment of the library of methods and search for the most effective their combinations.

References

[1] Babichev A. B., Kadyrova O. B., Kashevarova T. P., Leshchenko A. S., Semenov A. L. UniCalc, A Novel Approach to Solving Systems of Algebraic Equations. Interval Computations; 5(3):29-47, 1992.

[2] Hofstedt P. An Architecture for the Combination of Constraint Solvers. In Proceedings of ERCIM/COMPULOG workshop on constraints. CWI, Amsterdam, 1998.

[3] Marti P. and Rueher M. A Distributed Cooperating Constraint Solving System. International Journal on Artificial Intelligence Tools, 4(1&2):93-113, 1995

[4] Monfroy E. From Solver Collaboration Expressions to Communicating and Coordinated Agents. In Proceedings of the International Workshop for Object-oriented and Constraint Programming for Time Critical Applications, COTIC'99, Lisbon, Portugal, 1999.

[5] Monfroy E. A Solver Collaboration in BALI. In Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98), Manchester, England, The MIT Press, pages 349-350, 1998.

[6] Monfroy E. An Environment for Designing/Executing Constraint Solver Collaborations. In Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS'96), Kyoto, Japan, The MIT Press, 1996.

[7] Rueher M. An Architecture for Cooperating Constraint Solvers on Reals. In: Podelski, A., editor, Constraint Programming: Basics and Trends, vol. 910 of LNCS, pages 231-250, Springer, 1995.

[8] Semenov A., Kashevarova T., Leshchenko A., Petunin D. Combining Various Techniques with the Algorithm of Subdefinite Calculations. In Proc. of the 3rd International Conference on the Practical Application of Constraint Technology PACT'97, London, England, pages 287-306, 1997.

[9] Semenov A., Petunin D., Kleymenov A. GMACS — the general-purpose module architecture for building cooperative solvers. In Proceedings of the 2000 ERCIM/Compulog Net Workshop on Constraints. Padova, Italy, 2000.

[10] Van Hentenryck P., Michel L., and Deville Y. Numerica: A Modelling Language for Global Optimization. MIT Press, 1997.