Basic Components for Constraint Solver Cooperations

Eric Monfroy
IRIN
Université de Nantes
B.P. 92208
F-44322 Nantes Cedex 3, France
Eric.Monfroy@irin.univ-nantes.fr

Carlos Castro
Departamento de Informática
Universidad Técnica Federico Santa María
Avenida España 1680, Casilla 110-V
Valparaíso, Chile
ccastro@inf.utfsm.cl

ABSTRACT

We propose a predefined set of basic components for designing and implementing constraint solver cooperations and solver cooperation languages. Combining these components into patterns enables one to manage computation, control, and coordination needed for solver cooperations. Our framework has been implemented with the CHR language. We then used it to implement some cooperation primitives, and some constraint propagation with cooperative components.

1. INTRODUCTION

Solver cooperation is a research topic that has been widely investigated during the last years [12, 20, 6, 5]. Nowadays, very efficient constraint solvers are available. The challenge is to make them cooperate in order to 1) solve hybrid problems that cannot be treated by a single solver, 2) improve solving efficiency, and/or 3) reuse (parts of) solvers to reduce implementation costs. Solver cooperation languages (such as [9, 13, 15]) provide one with primitives to manage cooperation. However, implementing such a language is a tedious task since the instructions are complex and do not make a clear separation between computation and control. Indeed, a primitive of such a language not only controls interaction and manages coordination of several functions, but also administers their computations. Moreover, most (if not all) of these languages are tightly related to constraint problem representation: this limits future evolutions and extensions of these languages, and their cooperation features. Finally, studying properties and characteristics of a cooperation designed with such a language is very difficult because of the complexity of the primitives.

The goal of our work is not to propose a new solver cooperation language (such as BALI [5] or the language of [9]), but to propose a framework

- which simplifies the design and implementation of solver cooperations and solver cooperation languages,
- which allows for evolutionary cooperation languages which can be extended, improved, and adapted in order to better fulfil user's needs,
- which enables one to obtain a simple relation between solvers, cooperations, and constraints in order to study and deduce constraint and cooperation properties (such as in [19]) and to automatically generate cooperations.

In this paper, we focus on the first two items. Our framework is based on basic components connected via channels and executing concurrently: they aim at defining computations, behaviors, and interactions occurring in solver cooperations. Connecting components enables one to manage coordination (as defined in [3, 18, 17]) of the cooperation they belong to.

The structure and the behavior of a component are imposed by its type. First, a component type fixes the skeleton of a component: the number of input and output ports connected via channels to other components, and the number of function parameters. These standard functions compose the computational part of a component. The type of a component also determines its behavior and how to react to external events. This event mechanism which is defined by concurrent rules enables one to manage coordination, e.g., a component can synchronize two components by waiting for a message from each of them. Depending on their types, components can thus be used to compute, to control and coordinate other components, or to control the data-flow.

In our framework, the notion of pattern enables one to reuse and to incrementally build some cooperations. In fact, a pattern is a parametrized (by ports, functions, and patterns) combination of components. Thus, a pattern can define some more complex computation, coordination, and behavior than a single component.

The main contribution of this paper is to simplify solver cooperations and cooperation languages design and implementation. In fact, we propose a layer between standard programming languages and complex solver cooperation languages (such as BALI [5] and the language of [9]) and this framework allows one to design concurrent competitive, concurrent cooperative, and sequential cooperations.

Components have clear and simple behaviors that can easily be composed to create some more complex behaviors. We do not propose a new coordination model/language but we use some abstract concepts from the field of coordination (in fact, these concepts happen to be present in the IWIM model [3]) and applies them in the field of cooperative constraint solving.

ACM SAC '03 Melbourne, Florida USA

Our framework does not rely on a specific structure for constraint problems: one can thus freely consider new kinds of cooperations, and easily reuse patterns of components for different cooperations. The main drawback of this freedom is that the topology of components is determined before computation: a component cannot create other components, and cannot react to constraint structure. However, part of this problem can be solved by patterns or by considering more powerful function parameters.

The difference between the predefined set of agents of [21] and our components is that these agents are devoted to a specific distributed algorithm for consistency maintenance, whereas our components are tools to design solver cooperations, and for example, consistency maintenance. DICE [22] is a software framework for constructing distributed constraint solvers. Although symbolic derivations are supported, constraint solving is essentially based on domain reduction and branching. The kernel of our framework is not a given type of solvers, but the concept at the root of our framework is interaction among solvers of any type, and communication between solvers.

Our framework has been implemented with the Constraint Handling Rules (CHR) language [10] to manage concurrent rules implementing components. We then used our framework to implement some common cooperation strategies and the static (i.e., not reacting to constraints) primitives of the cooperation language of [8]. These patterns enabled us to implement some applications such as constraint propagation based on cooperative components.

This paper is organized as follows. In Section 2, we give some basic concepts related to constraints, control, and functions used in solver cooperation. We then define our notion of components and their types (Section 3), and how to combine them to create patterns (Section 4). In Section 5 we briefly describe the implementation of our framework, before presenting in Section 6 some applications. We finally conclude in Section 7.

2. BASIC NOTIONS

2.1 Constraints

Solver cooperation languages generally impose a constraint representation and map it to an interpretation. For example, a constraint is a disjunction of conjunctions of atomic constraints represented by a list of lists of atomic constraints or by logical terms such as $(c_{1,1} \wedge c_{1,2}) \vee (c_{2,1} \wedge c_{2,2})$. When considering more "computational" information, this representation has to be extended with new structures. In [8], besides the representation of constraints as logical terms, sets and lists of constraints are required to represent different "computational" possibilities, i.e., different candidates on which a solver can be applied.

In order to abstract from standard representations and to allow one to integrate computational information, we consider that constraint representation is a parameter of our framework. We call this pool of information a *constraint problem*, and we denote by $\mathcal{P}$ the set of constraint problems. A problem defines a set of solutions that we want to find in an initial search space. Usual methods amount to reducing the search space until it corresponds to the set of solutions.

We introduce some notions to clarify some of the operations we perform in cooperation. We refer to *choice points* and *branches* to denote a node of the search tree and the choices it leads to. We call *computational choice points* and *candidates* several parts (the candidates) of a problem on which a solver (cooperation) can be effectively applied. For example, consider a solver which extract roots of a polynomial equation. If the problem p contains several polynomial equations, there is a computational choice point, and each of the equations is a candidate. However, if we create p' by adding $x = 1$ to p, and p'' by adding $x \neq 1$ to p, then we have a choice point leading to the branches p' and p'' (this is a common technique to perform enumeration).

2.2 Control

In most of the solver cooperation languages, the control is part of the semantics of the primitives [5]. In our framework, the components we consider are fine grain, less complex, and possess less control capabilities. Thus, we consider some control data to perform complex control among components. We separate constraint problems from controls: problems are data for computation, and controls are data to manage interaction among components and data flow. A control is represented by a Boolean value, and the set of controls is denoted by $\mathcal{B}$. In order to fully separate the two notions, our framework provides channels and ports for data (problems), and channels and ports for controls (Booleans).

2.3 Functions

Functions are parameters of components and are executed by components. We consider different classes of functions with respect to their profiles. Then, we refine these classes with respect to semantics of functions. These sub-classes have no effect upon connections of components but they are standard classes with defined semantics used for solver cooperations.

Transformers The first class of functions we consider are computable functions from constraint problems to constraint problems:

$$f : \mathcal{P} \rightarrow \mathcal{P}$$

Among the functions of this class, the major sub-classes are *solvers*, *filters*, and *selectors*. A *solver* is a function that reduces the search space defined by a constraint problem. Functions of very different nature are considered as solvers, e.g., domain reduction functions [1, 2], symbolic transformations or deductions (such as Gröbner bases computation), or splitting mechanisms (such as enumeration) which create branches in the search space.

A *filter* is a computable function that returns a constraint problem containing one or several computational candidates which fulfil some requirements. A filter is generally used for preparing data for a solver.

A *selector* returns a problem containing a part of the input constraint problem. It can be used to select a branch at a choice point, or a candidate at a computational choice point.

Problem combiners A *constraint combiner* $\sigma_C : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$ combines two constraint problems into a new one. Standard combiners are disjunction, conjunction, or union of constraints.

Control generator A *property* $\pi : \mathcal{P} \rightarrow \mathcal{B}$ is a function which given a problem returns a control data. Usual examples of this type of functions are *isLinear* (to check if a

problem is composed of linear equations only), *isEmpty* (to test whether a problem is empty or not), ...

Comparator A *comparator* $\gamma : \mathcal{P} \times \mathcal{P} \to \mathcal{B}$ is a function that given two problems returns a control. $\sqsubseteq$ and $=$ are standard comparators.

Control combiner A *control combiner* $\sigma_{\mathcal{B}} : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ is a function that combines two controls into one. Standard control combiners are binary logical operators such as *and*, *or*, or *xor*.

Control reverser This class of functions is composed of one function. The *not* function $not : \mathcal{B} \to \mathcal{B}$ reverses a control; this is the standard logical not operator.

3. COMPONENTS

Components are entities that are connected to each other via channels to exchange problems and controls. Components have different tasks: some are devoted to problem transformation, some to control generation, some to control and coordination management, some to data-flow management.

Locally a component possesses some input and output ports, i.e., some openings to the outside world. Ports are connected by one to one, unidirectional channels carrying messages. Ports and channels are dedicated either to problems (computational data) or to controls (control data). A channel is declared by

$$\mathsf{connect}(i : \mathsf{port\_type};\ o : \mathsf{port\_type})$$

which means that messages will pass from port i to port o. i and o are of the same type which also determines the type of the channel. Since ports are used as identifiers to connect components, port names must be unique.

Messages are stored on input ports in a FIFO buffer until they are consumed one by one by the component. Messages pass through channels as soon as they are put on output ports, and thus, channels do not need any buffer. An input port can be connected to several channels whereas an output port is connected to only one channel.

A component may execute some functions (presented in Section 2.3) given as its parameters. Depending on the design of the system, the implementation of a component A can have access to a function f as a library, as a piece of code integrated in the code of the component, or even as an external component connected to A.

A component is represented by a term as follows:

$$\begin{array}{ll}\mathsf{component\_type}(& \mathsf{in}\ i_1 : \mathsf{port\_type}, \ldots, i_n : \mathsf{port\_type};\\ & \mathsf{out}\ o_1 : \mathsf{port\_type}, \ldots, o_m : \mathsf{port\_type};\\ & f_1 : \mathsf{function\_class}, \ldots, f_l : \mathsf{function\_class})\end{array}$$

where:

- component_type represents the type of the component (see Section 3.2),
- $i_1, \ldots, i_n$ (resp. $o_1, \ldots, o_m$) are input (resp. output) ports together with their respective types (port for problems or controls),
- $f_1, \ldots, f_l$ are functions as described in Section 2.3 together with their respective classes (e.g., transformer, combiner, control combiner, ...).

3.1 Components as concurrent rules

The way messages are read and sent defines behaviors of components, their interactions, synchronizations, and coordination. A component is described by a set of concurrent rules of the form:

$$\mathsf{Head} \Rightarrow \mathsf{Body}$$

where:

- Head is a sequence of $\mathsf{get}(input\_port, data)$ instructions;
- Body is a sequence of $\mathsf{put}(output\_port, data)$ instructions, and function calls.

To match a rule means that some data are pending on all the ports appearing in the head of the rule (in the get instructions). To fire (or to trigger) a rule signifies that the rule matches and that the actions of the body are achieved in sequence. Data pending on input ports are consumed as soon as a rule that needs them to match is effectively fired. Thus, they cannot be used to match two rules that are fired.

As rules are concurrent, when a component is composed of several rules, the first one that matches is triggered. When a rule is triggered, it cannot be stopped, and it executes all the instructions of the body before finishing; then the component returns to a state in which it tries to match rules.

The $\mathsf{get}(i, D)$ instruction is not blocking: if there is no data on the input port i, then get simply fails and the rule does not match. The instruction $\mathsf{put}(o, D)$ drops D on the output port o. Then, D is delivered in finite time to the other extreme of the channel connected to o.

Let us illustrate this notion of rules by an example. Consider a component managed by the rule

$$\mathsf{get}(i_1, d_1), \mathsf{get}(i_2, d_2) \Rightarrow \mathsf{put}(o, s(d_1, d_2))$$

Assume a message d_1 arrives on i_1, but nothing on i_2. Then the rule does not match and thus it is not triggered. Moreover, d_1 remains on the port i_1. Now assume a message d_2 arrives on i_2. Now the rule matches (there is a message on i_1 and one on i_2), and is triggered: d_1 and d_2 are removed from input ports, the function s is applied and the result $s(d_1, d_2)$ is put on o.

Note that the matching of a rule can be strengthened by requesting some more conditions on messages, e.g., the rule $\mathsf{get}(i, true) \Rightarrow \mathsf{put}(o, false)$ specifies that the component waits for a message on i, and that the message must be the control data *true*.

3.2 Types of components

In the following, we abstract data types to concentrate on component interaction, coordination, and synchronization. This does not create any confusion since the type of data and ports can be easily deduced from the context, and generally by the type of functions given as parameters.

We group components into types that define skeletons of components and their behavior. The skeleton of a component is composed of its input ports, output ports, and the number of functions given as its parameters. The behavior of a component is described by the way the component consumes messages, on which data it applies the functions, and on which ports it puts some data. Although they handle different types of data and different classes of functions,

a component managing a problem combiner, and a component managing a control manager (see function classes in Section 2.3) have the same skeleton, and the same behavior.

We now define the different types of components, their skeletons, their behaviors, and their representations as concurrent rules.

Transformers Components of this type have one input port i, one output port o, and a parameter function f. When they receive a message d on i, they consume it, apply f on d, and send the result on o. Their syntax is the following:

$$\mathsf{transformer}(\mathsf{in}\ i;\ \mathsf{out}\ o;\ f)$$

A transformer component is used with transformer (solvers, filters, ...) control generator (properties), and control reverser (not) functions. A single rule implements and manages a transformer component:

$$\mathsf{get}(i, d) \Rightarrow \mathsf{put}(o, f(d))$$

Synchronizers These components synchronize two components by the mediating of their messages. They have two input ports (i_1 and i_2), one output port (o), and a function parameter f. They act as follows: they wait for two messages (one on i_1 and one on i_2); as soon as they have the two messages, they apply f on these data, and finally put the result on o. A synchronizer component has this form:

$$\mathsf{synchronize}(\mathsf{in}\ i_1, i_2;\ \mathsf{out}\ o;\ f)$$

and is managed by the rule

$$\mathsf{get}(i_1, d_1), \mathsf{get}(i_2, d_2) \Rightarrow \mathsf{put}(o, f(d_1, d_2))$$

A synchronizer component can handle problem combiner functions (i_1, i_2, and o are problem ports), comparator functions (i_1 and i_2, are problem ports, o is a control port), control manager functions (i_1, i_2, and o are control ports).

First A first component has four input ports (i_1, i_2, ii_1, and ii_2) and one output port o. Ports ii_1 and ii_2 are some kind of local memories: a first component can put (store) and get (recall) some messages on them. Ports ii_1 and ii_2 are not accessible from outside of the component, and are not connected to any channel. Thus, they do not appear in the syntax of a first component:

$$\mathsf{first}(\mathsf{in}\ i_1, i_2;\ \mathsf{out}\ o)$$

For a couple of messages (one arriving on i_1, the other one on i_2), only the first one arrived is immediately put on o, the other one will be destroyed when it will arrive. Four rules define the behavior of a first component:

$$\mathsf{get}(i_1, d), \mathsf{get}(ii_2, true), \Rightarrow \mathsf{put}(ii_2, false)$$
$$\mathsf{get}(i_2, d), \mathsf{get}(ii_1, true), \Rightarrow \mathsf{put}(ii_1, false)$$
$$\mathsf{get}(i_1, d), \mathsf{get}(ii_2, false), \Rightarrow \mathsf{put}(ii_1, true), \mathsf{put}(o, d)$$
$$\mathsf{get}(i_2, d), \mathsf{get}(ii_1, false), \Rightarrow \mathsf{put}(ii_2, true), \mathsf{put}(o, d)$$

When the message $true$ is put on ii_1, this indicates that the first component has already read a message on i_1, and forwarded it to o. If the message on ii_1 is $false$, either no message has been read yet on i_1, or the component has been reset. The reset is performed by Rule 1 (respectively Rule 2) which destroys the message on i_1 (respectively i_2), and reinitialize the component by putting $false$ on ii_2 (respectively ii_1). This mechanism "cleans" the input ports of a first component to prepare for its later use. When implementing the component, ii_1 and ii_2 must be initialized by

$$\mathsf{put}(ii_1, false), \mathsf{put}(ii_2, false)$$

This component is interesting when using two similar cooperations in parallel: only the result of the fastest one is kept, and forwarded as soon as it arrives.

Sieve A sieve component has two input ports (i_1 for either problems or controls, and i_2 which is always a control port), and one output port o (of the type of i_1):

$$\mathsf{sieve}(\mathsf{in}\ i_1, i_2;\ \mathsf{out}\ o)$$

A sieve component waits for a message d on i_1, and for a control b on i_2. If b is true, then d is put on o, otherwise d is consumed but no action is performed (nop):

$$\mathsf{get}(i_1, d), \mathsf{get}(i_2, true) \Rightarrow \mathsf{put}(o, d)$$
$$\mathsf{get}(i_1, d), \mathsf{get}(i_2, false) \Rightarrow \mathsf{nop}$$

Such a component blocks a message until a control arrives, and deletes it if the control is $false$. This avoids putting a message d on a component A when we are not sure to consume d in A: A remains "clean" and ready for a next use.

Duplicate A duplicate component gets a message d on its input port i and returns a copy of d on both its output ports o_1 and o_2. d can be a constraint problem, or a control; i, o_1, and o_2 must be of the same type:

$$\mathsf{duplicate}(\mathsf{in}\ i;\ \mathsf{out}\ o_1, o_2)$$

and its behavior is described by the rule

$$\mathsf{get}(i, d) \Rightarrow \mathsf{put}(o_1, d), \mathsf{put}(o_2, d)$$

4. SOLVER COOPERATIONS

In this section, we use our restricted set of components to design some usual forms of solver cooperations. A solver cooperation is a set of connected interacting components that exchange information. We can distinguish two types of cooperations: compositions and patterns.

By composition, we mean a set of components that are totally fixed. The coordination, the behavior, the topology, and function parameters are fixed. A simple example consists of applying two solvers $s1$ and $s2$ in sequence (this is similar to the sequence primitive of BALI). This cooperation is defined by the set of components

$$\{\ \mathsf{transformer}(\mathsf{in}\colon i_{s1};\ \mathsf{out}\colon o_{s1};\ s1),\ \mathsf{connect}(o_{s1}, i_{s2}),\ \mathsf{transformer}(\mathsf{in}\colon i_{s2};\ \mathsf{out}\colon o_{s2};\ s2)\ \}$$

As soon as a constraint problem p is put on i_{s1}, it is solved by $s1$, and put on o_{s1}. When $q = s1(p)$ has reached i_{s2}, it is treated by $s2$, and the result is put on o_{s2}.

¿From a coordination point of view, patterns are more interesting than composition. They define some more complex and parametrized components that are built from basic components (see Section 3) and patterns.

Unlike composition, patterns hide their internal structure: this structure is visible in the definition of the patterns (i.e., similar to the body of a function), but not in its use (i.e.,

similar to the declaration of a function). The advantage is to reuse some cooperations without any knowledge about their internal structures and the components they involve while having the possibility of having some ports, functions, and patterns as parameters. The declaration of a pattern pat has the form:

$$\mathsf{pat}(\mathsf{in}:\ i_1,\ldots,i_n;\ \mathsf{out}:\ o_1,\ldots,o_m;\ f_1,\ldots,f_l;\ p_1,\ldots,p_k)$$

where the parameters are: $i_1,\ldots,i_n$ are input ports; $o_1,\ldots,o_m$ are output ports; $f_1,\ldots,f_l$ are functions; and, $p_1,\ldots,p_k$ are patterns.

We say that $\mathsf{in}:\ i_1,\ldots,i_n;\ \mathsf{out}:\ o_1,\ldots,o_m$ is the *profile* of the pattern. Note that to shorten notation, we have omitted the types of ports, classes of functions, and *profiles* of patterns.

Ports of the profile of a pattern are the *external ports* of the pattern, whereas ports that are used in the definition of a pattern are said *internal*: external ports are visible from the outside of the pattern and appear in its declaration, whereas internal ports are only visible in its definition. Note that function and pattern parameters of the p_i's do not appear since they are internal to the pattern pat.

Hence, a pattern q can be used as a parameter of a pattern $p(\ldots p' \ldots)$ if function parameters of q are instantiated (or they are also parameters of p; in this case, they must have the same formal name) and the profile of q is identical to the one of p'. For execution, ports of the profile of q are connected (with channels) to ports of p.

Thus, except the notion of pattern parameters, the concept of pattern is very similar to the one of component: a pattern has a fixed skeleton and a fixed behavior defined by the components and patterns it is built upon (called internal), their composition, and their connections.

Let us illustrate the notion of pattern on simple examples. On figures, normal lines represent problem channels, and other lines are control channels.

MultiDuplicate pattern A data must often be duplicated several times. For composing several duplicate components, we propose a multiDuplicate pattern:

$$\mathsf{multiDuplicate}(\mathsf{in}\ i;\ \mathsf{out}\ o_1,\ldots o_n)$$

This pattern (Figure 1, definition (left) and declaration

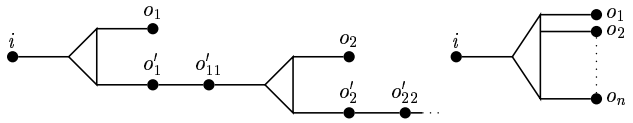

Figure 1: MultiDuplicate pattern

(right)) duplicates a message (arriving on i) on several output ports o_1, ..., o_n. This pattern is implemented by the following definition:

$$\begin{array}{l}\mathsf{duplicate}(i,o_1,o_{1'}),\ \ \mathsf{connect}(o_{1'},o_{11'}),\\ \mathsf{duplicate}(o_{11'},o_2,o_{2'}),\ \ \mathsf{connect}(o_{2'},o_{22'}),\\ \quad\vdots\\ \mathsf{duplicate}(o_{n-3'},o_{n-2},o_{n-2'}),\ \ \mathsf{connect}(o_{n-2'},o_{n-22'}),\\ \mathsf{duplicate}(o_{n-22'},o_{n-1},o_n)\end{array}$$

Although this pattern is rather simple (no function or pattern parameter), its writing is complex. The number of channels grows in direct ratio with the number of duplicate components. We consider *implicit channels* for simplification.

Implicit channels Consider a component A and a component B, such that the output port a of A is connected to the input port b of B by a channel. When this does not cause any confusion, instead of writing $\mathsf{connect}(a,b)$ we can rename b to a. Implicitly, there exists a channel between these two ports, but explicitly, it looks like the two components are sharing a port. Using this notion, the multiDuplicate pattern simplifies to

$$\mathsf{duplicate}(i,o_1,o_{1'}),\ \mathsf{duplicate}(o_{1'},o_2,o_{2'}),\ \ldots,$$

Switch pattern A *switch* pattern receives a constraint problem, and depending on the value of the application of a property π on this problem, puts the result on one of the two output ports. Its declaration is:

$$\mathsf{switch}(\mathsf{in}\ i;\ \mathsf{out}\ o_1,o_2\ ;\pi)$$

This switch pattern can be defined (and thus implemented)

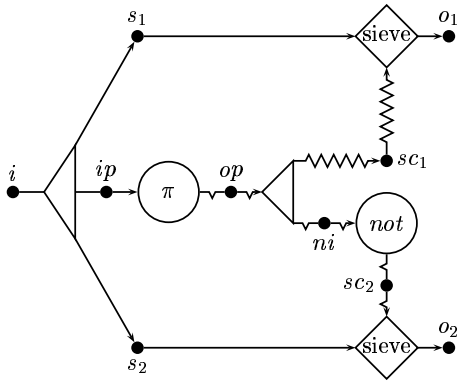

Figure 2: Switch pattern

by (see Figure 2):

$$\begin{array}{l}\mathsf{multiDuplicate}(i,s_1,ip,s_2),\\ \mathsf{transformer}(ip,op,\pi),\\ \mathsf{duplicate}(op,sc_1,ni),\\ \mathsf{transformer}(ni,sc_2,not),\\ \mathsf{sieve}(s_2,sc_2,o_2),\\ \mathsf{sieve}(s_1,sc_1,o_1)\end{array}$$

Fixed point A fixed point is a common primitive of solver cooperations: it applies iteratively a cooperation (in our case, a pattern p) until the result does not change any more, i.e., the input and the output of the cooperation are the same. The declaration of a fixed-point pattern is

$$\mathsf{fp}(\mathsf{in}\ i;\ \mathsf{out}\ o\ ;\mathit{Equal}\ ;(\mathsf{p}:\ \mathit{coop};\ \mathsf{in}:\ \mathit{ic};\ \mathsf{out}:\ \mathit{oc}))$$

where *Equal* is a function parameter [1] to test equality of two constraint problems, and *coop* is a pattern having one input and one output port, and possibly some function and pattern parameters (see next section for an example of instantiation of fixed-point). A possible definition of a fixed-point

[1]The *Equal* function is a parameter since depending on the structure of constraint problems, testing equality can be different.

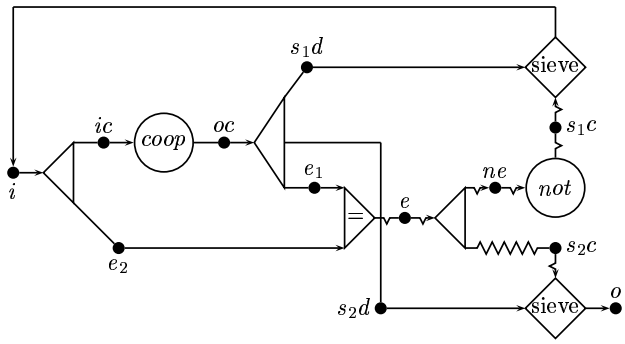

Figure 3: Fixed point pattern

(Figure 3) is:

$$\{\ \mathsf{duplicate}(i, ic, e_2),\ \mathsf{multiDuplicate}(oc, s_1d, s_2d, e_1),\ \mathsf{synchronizer}(e_1, e_2, e, Equal),\ \mathsf{duplicate}(e, ne, s_2c),\ \mathsf{sieve}(s_1d, s_1c, i),\ \mathsf{sieve}(s_2d, s_2c, o),\ \mathsf{transformer}(ne, s_1c, not),\ \mathsf{coop}(ic, oc)\}$$

The *Equal* parameter of the fixed-point pattern is the parameter of an internal component (a synchronizer). When instantiated, the pattern *coop* is replaced by its full declaration (function and pattern parameters are added).

5. IMPLEMENTATION

To implement such a framework, several languages can be very good candidates: Manifold [4] for its coordination facilities [3, 18] and its appropriateness to our coordination needs, ELAN [7, 14] for rule-based programming, or the implementation of a tuple-space based model (such as Linda [11]) for pattern matching features.

We choose the CHR [10] language since with its rule and concurrency capacities, it offered the opportunity to directly implement the rules as presented before.

The instructions of our language are some Prolog predicates that generate a set of CHR rules. Compiling these CHR rules generates the topology of components, their communication, and their interactions. The definition of components as rules is used nearly straightforward in our implementation. In CHR, there is a shared memory which is the constraint store. By shared, we mean that each rule uses concurrently constraints of this memory to match, and each generated constraint is stored in this memory.

By tagging constraints with port names (that are unique), a component can easily find in the store which constraint it can use. Thus, we consider CHR constraints of the form get(port,D). Depending on the type of component, D is a constraint problem or a control; port is a unique identifier which corresponds to the name of the port. For a component, getting data on an input port corresponds to getting a get constraint in the shared memory. The instruction put($port, D$) corresponds to adding the constraint get(port,D) in the store.

Firing a rule is the usual CHR mechanism: if the rule matches, then the body of the rule is executed and the constraints of the head are removed from the store. This behavior is the same as the one of our components for which messages are removed from ports when they are used. Different functions (such as solvers, filters, ...) are implemented as Prolog predicates. Note that these predicates can call some external solvers.

Creating a channel between two ports is performed by the connect(a, b) component which is implemented by the rule:

```
get(a,C) <=> put(b,C).
```

The shortcut that consists in removing channels between an input and an output port can be kept with our implementation. This is done by sharing the name of the port. One component will be allowed to put problems on this port, the other one to get problems.

We now illustrate our implementation with a simple example. A transformer is stated as transformer(In, Out, S). The call transformer(a, b, s) creates the rule

```
get(a,C) <=> s(C,CC), put(b,CC).
```

A cooperation is built as a list of components. A predicate enables one to automatically create the corresponding components as a set of rules. To use these rules, one has to just put a message on the input port (`put(input_port, problem)`). The cooperation executes, and the result is stored on the output port that can be read or connected to a print component.

6. APPLICATIONS

In this section, we present the implementation of some of the primitives of the cooperation language of [9, 8]. We also present some executions of our components for constraint propagation, a standard method for constraint solving [1].

We now present a simple structure for problems that we use in the following applications. We consider atomic constraints; a conjunction is a list (denoted as [...]) of atomic constraints; a disjunction is a list of conjunctions. Computational choice points are also represented by lists of candidates. We consider finite domain variables and two types of atomic constraints: $in(a, [la, ua])$ determining the domain of a (the values v that a can take are $la \leq v \leq ua$), and $leq(a, b)$ meaning $a \leq b$.

We consider a solver *narrow_leq* which given a problem $leq(a, b)$, $in(a, [la, ua])$, $in(b, [lb, ub])$ reduces the domain of a and b in order to make the problem locally consistent [16] (i.e., using a single constraint $leq(a, b)$, no more value can be removed from the domain of a and b).

Basic primitives The implementation of the basic primitives of [8] is straightforward with our components. The sequence consists of connecting the output of a cooperation to the input of the next one. The don't care of [8] is a simple pattern where the dc component is a synchronizer component with a don't care function to randomly choose one of the inputs ($dc(d_1, d_2) = d_1$ *or* $dc(d_1, d_2) = d_2$). The fixed-point primitive corresponds to our fixed-point pattern (see Section 4).

Best_apply In [8], the semantics of this primitive is: given a problem p, make the best (w.r.t. some criteria given as parameters) possible application of a cooperation *coop* on

a part (defined by some criteria given as parameters) of p such that p is effectively modified. In other words, w.r.t. our framework: there is a computational choice point defining several candidates on which *coop* can be applied; *coop* is applied on one of them only; and this candidate is the best one (w.r.t. some criteria) that *coop* will really modify.

The operational semantics of this primitive is rather complex. Thus, we divide the problems into sub-problems, each of them corresponding to a pattern. At the end, these patterns are grouped in a pattern whose operational semantics is equivalent to the one of the best_apply primitive.

We do not detail all the patterns, but only some of them. The csolve pattern takes as input a problem p, applies a pattern *coop* (corresponding to the solver cooperation of best_apply) on p and has three output ports: if *coop* really modifies p, then *true* is put on the success port, p is put on the old port, and the result of applying *coop* is put on the new port. Otherwise, $false$ is put on success, and nothing on old and new.

The cselect pattern creates a computational choice point (using a filter function). As long as it reads *true* on a more input port, it sends candidates on the outC output port. When $false$ is read on more, or when there is no more candidate (i.e., p is empty), cselect stops.

The solba pattern is responsible for replacing transformed constraints in the initial problem.

The declaration of the fully generic bestApply pattern is the following:

bestApply(In, Out,
$Filter, Selector, IsEmpty, SubList, SubC, AddC, Equal$,
$(Coop : in_c, out_c)$)

where In and Out are the input and output ports of the pattern. $IsEmpty$, $SubList$, $SubC$, $AddC$, and $Equal$ are functions related to constraint problem representation [2], $Coop$ is the cooperation to apply, and in_c, out_c are its input and output ports. $Filter$ and $Selector$ are parameters to create the candidates, and then to determine which one is best (they represent the criteria given as the parameters of the best_apply primitive). Here is an example of use of our pattern:

bestApply(i, o,
$filter\_ldd, firstList, isEmpty, subList, subC, addC, equal$,
$transformer(ii, oo, narrow\_leq)$)

where $filter\_ldd$ filters a *leq* constraint and the related domain constraints (i.e., a suitable input for the $narrow\_leq$ solver); $firstList$ is a strategy that performs depth first search (first in the list with our constraint problem representation); the other functions are related to problem structure: *isEmpty* is true when there is no more computational choice point, *subList* removes a computational choice point, *subC* and *addC* removes and adds constraints in a problem, and *equal* checks whether two problems are equal.

When sending the problem [in(a,[2,8]), leq(a,b), in(b,[3,6])] on port i, $filter\_ldd$ finds one candidate [leq(a,b), in(a,[2,8]), in(b,[3,6])] which is obviously chosen by the $firstList$ selector, and then solved by $narrow\_leq$. The final result of the bestApply pattern, can be found on o and is [leq(a,b), in(a,[2,6]), in(b,[3,6])].

On the problem [leq(c,d), in(c,[1,1]), in(d,[3,6]), leq(a,b), in(a,[1,10]), in(b,[3,6])], $filter\_ldd$ finds 2 candidates. The selector sends the candidate [leq(c,d), in(c,[1,1]), in(d,[3,6])] to the solver. This candidate cannot be modified by the solver. Thus, the selector sends the next candidate [leq(a,b), in(a,[1,10]), in(b,[3,6])] which is effectively reduced by the $narrow\_leq$ solver. The answer on o is finally [leq(a,b), in(a,[1,6]), in(b,[3,6], leq(c,d), in(c,[1,1]), in(d,[3,6])].

As in the best_apply primitive, we can change the strategy for applying cooperation by changing the Filter and Selector given as parameters. Note that one of the main difficulties in defining the bestApply pattern is that there should not remain any message in any component after applying it. Otherwise, we could not use it in a fixed-point as shown later.

Other primitives Implementing the dc_apply primitive of [8] is rather simple: we can just reuse the bestApply pattern with a selector that makes a random choice. The parallel_best_apply primitive consists in adding a multiDuplicate pattern in front of several bestApply patterns, and then to connect these bestApply patterns with some synchronizers components with combiner functions. With our framework, the parallel_concurrent_apply primitive of [8] is implemented as a parallelBestApply pattern in which combiners of the synchronizers (after the bestApply's) are replaced by the don't care function (see its definition in the don't care pattern).

We can easily imagine new patterns to complete the language of [8]: for example, an allApply pattern which would apply a solver on every candidate. To this end, we just have to change the csolve pattern (used in a bestApply pattern) such that 1) it always sends *true* on the more port of the cselect pattern; and 2) it always sends the old problem and new problem on old and new output ports.

Fixed-point of a best apply We now use the bestApply example we had before (with the $narrow\_leq$ solver, the $filter\_ldd$ filter, and the $firstList$ selector) as the pattern parameter of a fixed-point pattern. In terms of cooperation, we compute the full reduction of a problem (the problem is locally consistent w.r.t. each constraint), and this requires a fixed-point.

When flattened, this pattern is composed of 45 components. When we send on in_fp the problem [leq(a,b), leq(b,c), in(a,[7,8]), in(b,[1,9]), in(c,[2,7])] the system reduces domains of a, b, and c. We obtain on out_fp the answer [leq(a,b), in(a,[7,7]), in(b,[7,7]), leq(b,c), in(c,[7,7])]. The trace of execution shows that the bestApply pattern was called 4 times: 3 times successfully (i.e., the solver really modified the problem), and once unsuccessfully (no modification of the problem); this determines that the fixed point is reached. The $narrow\_leq$ solver was called 7 times, and among them, 3 times successfully (this corresponds to the 3 successes of the bestApply). The other 4 times are tries that do not reduce the domains, i.e., either a candidate that fails in the bestApply, or the reach of the fixed-point.

7. CONCLUSION

[2]Patterns can be simplified by fixing the problem structure. On the best_apply pattern, this would fix all functions related to problem structure; thus, this would significantly reduce the number of parameters. However, the definition would remain the same, calling some fixed functions instead of function parameters.

We have presented a set of basic components to design and implement solver cooperations. The advantage is to provide a clear separation between computation and control, and thus, to ease design and implementation of cooperations and cooperation languages as well. The notion of pattern allows for incremental realization of cooperations by considering some kinds of complex components: patterns have more complex skeletons and behaviors than components. Moreover, constraint problem structure is free, and thus cooperation and coordination can be reused for other domains, or other constraint representations. Languages implemented with our framework can thus evolve and progress.

The goal of our first implementation was to test the capacity of our framework to design cooperations and cooperation languages. To go further and produce some more results, we now need an implementation that allows distributed computation, and that can easily call bigger and complex external solvers. With all its coordination features, a language such as Manifold [4] will be a good candidate for the next implementation.

We also plan to extend the work of [19] using our framework in order to be able to deduce more properties about constraints and cooperations, and finally, to automatically create cooperations w.r.t. to the problems to be solved.

Acknowledgement

We would like to thank all the referees for useful suggestions and comments. The authors have been partially supported by a grant from the Chilean National Science Fund through the project FONDECYT 1010121, and by the IST project COCONUT from the European Community.

8. REFERENCES

[1] K. R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999. Available via `http://arXiv.org/archive/cs/`.

[2] K. R. Apt. The Role of Commutativity in Constraint Propagation Algorithms. *ACM Trans. on Programming Languages and Systems*, 22(6):1002–1036, 2000.

[3] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Proc. of the International Conference Coordination Languages and Models*, volume 1061 of *LNCS*, pages 34–56. Springer-Verlag, 1996.

[4] F. Arbab. *Manifold2.0 reference manual*. CWI, Amsterdam, The Netherlands, May 1997.

[5] F. Arbab and E. Monfroy. Coordination of Heterogeneous Distributed Cooperative Constraint Solving. *ACM SIGAPP Applied Computing Review*, 6(2):4–17, 1998.

[6] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In *Logic programming: formal methods and practical applications*, Studies in Computer Science and A.I. Elsevier, 1995.

[7] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of elan. *ENTCS*, 15, 1998.

[8] C. Castro and E. Monfroy. A Control Language for Designing Constraint Solvers. In *Proc. of the 3rd Int. Conf. Perspective of System Informatics, PSI'99*, volume 1755 of *LNCS*, pages 402–415, Novosibirsk, Akademgorodok, Russia, 2000.

[9] C. Castro and E. Monfroy. Basic Operators for Solving Constraints via Collaboration of Solvers. In J. A. Campbell and E. Roanes-Lozano, editors, *Proc. of the 5th Int. Conf. on Artificial Intelligence and Symbolic Computation (AISC'2000)*, volume 1930 of *LNCS*, pages 142–146, Madrid, Spain, 2001. Springer.

[10] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[11] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] L. Granvilliers, E. Monfroy, and F. Benhamou. Symbolic-interval cooperation in constraint programming. In *Proc. of the 26th Int. Symp. on Symbolic and Algebraic Computation (ISSAC'2001)*, pages 150–166, London, Ontario, Canada, 2001. ACM Press.

[13] P. Hofstedt. Better communication for tighter cooperation. In *Proceedings of Computational Logic, London, UK*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 342–357. Springer-Verlag, 2000.

[14] C. Kirchner and C. Ringeissen. Rule-based constraint programming. *Fundamenta informaticae*, 33(3):225–262, 1998.

[15] A. Kleymenov, D. Petunin, A. Semenov, and I. Vazhev. A model of cooperative solvers for computational problems. *Joint Bulletin of NCC and IIS. Series: Computer Science*, 16:115–128, 2001.

[16] A. K. Mackworth. Constraint Satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of A.I.*, volume 1. Addison-Wesley, 1992. 2nd Edition.

[17] G. A. Papadopoulos and F. Arbab. Control-Driven Coordination Programming in Shared Dataspace. In *Proc. of the 4th Int. Conf. on Parallel Computing Technologies (PaCT-97)*, volume 1277 of *LNCS*, pages 247–261, Yaroslavl, Russia, 1997. Springer-Verlag.

[18] G.A. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers – The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.

[19] E. Petrov and E. Monfroy. Automatic analysis of composite solvers. In *Proceedings of the 14th International Conference on Tools with Artificial Intelligence*, 2002. To appear.

[20] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Frontiers of Combining Systems (FroCoS'96)*, Applied Logic, pages 121–140. Kluwer Academic Publishers, 1996.

[21] M.C. Silaghi, D. Sam-Haroud, and F. B.Faltings. Maintaining hierarchical distributed consistencies. In *CP2000 DCS Workshop*, Singapore, 2000.

[22] P. Zoeteweij. Coordination-based solver cooperation in DICE. In *Proceedings of the COSOLV'2002 Workshop on Cooperative Solvers in Constraint Programming (held in conjunction with CP 2002)*, Ithaca, NY, USA, 2002.