Implementing Global Constraints as Graphs of Elementary Constraints*

Dávid Hanák

Budapest University of Technology and Economics
Dept. of Computer Science and Information Theory
dhanak@cs.bme.hu

Abstract. Global constraints are cardinal concepts of CLP(FD), a constraint programming language. They are means to find a set of integers that satisfy certain relations. The fact that defining global constraints often requires the knowledge of a specification language makes sharing constraints between scientists and programmers difficult. Nicolas Beldiceanu presented a theory that could solve this problem, because it depicts global constraints as graphs: an abstraction that everyone understands.
The abstract description language defined by the theory may also be interpreted by a computer program. This paper deals with the problematic issues of putting the theory into practice by implementing such a program. It introduces a concrete syntax of the language and presents three programs understanding that syntax. These case studies represent two different approaches of propagation. One of these offers exhausting pruning with poor efficiency, the other, yet unfinished attempt provides a better alternative at the cost of being a lot more complicated.

1 Introduction

Constraint Logic Programming (CLP, also referred to as Constraint Programming, CP) [1] is a family of logic programming languages, where a problem is defined in terms of correlations between unknown values, and a solution is a set of values which satisfy the correlations. In other words, the correlations *constrain* the set of acceptable values, hence the name. A member of this family is CLP(FD), a constraint language which operates on variables of integer values. Like CLP solvers in general, CLP(FD) solvers are embedded either into standalone platforms such as the ILOG OPL Studio [2] or host languages, such as C [3], Java [4], Oz [5] or Prolog [6,7].

In CLP(FD), FD stands for *finite domain,* because each variable has a finite set of integer values which it can take. These variables are connected by the constraints, which propagate the change of the domain of one variable to the domains of others. A constraint can be thought of like a "daemon" which wakes up when the domain of one (or more) of its variables has changed, propagates the change and then falls asleep again. This change can be induced either by an other constraint or by the *distribution* or *labeling* process, which enumerates the solutions by successively substituting every possible value into the variables. Constraints can be divided into two groups: simple and global constraints. The former always operate on a fixed number of arguments (like $X = Y$), while the latter are more generic and can work with a variable number of arguments (e.g., "$X_1, X_2, \ldots, X_n$ are all different").

Many solvers allow the users to implement user-defined constraints. However, the specification languages vary. In some cases, a specific syntax is defined for this purpose, in others, the host language is used. There are several problems with this. First, CLP(FD) programmers using different systems could have serious difficulties sharing such constraints because of the lack of a common description language. Second, to define constraints, one usually has to know the solver in greater detail than if merely using the predefined ones. Inspired by these problems, Nicolas Beldiceanu suggested a new method for defining and describing global finite domain constraints [8]. After

* The results reported in this paper were presented at the CS^2 conference held at Szeged, July 1–4. 2002.

studying his theory, I decided to put it into practice by implementing a parser of Beldiceanu's abstract description language (ADL), as an extension to the CLP(FD) library of SICStus Prolog [9, Sect. CLPFD], a full implementation of the CLP(FD) language.

The paper is structured as follows. Section 2 introduces the theory of Beldiceanu, explains how constraints may be represented by graphs and describes the ADL in some detail. Section 3 specifies the concrete syntax of the language used by the implementation, Sect. 4 presents the implemented programs capable of understanding such a description. Section 5 gives some ideas about the possible directions of future research and development, and finally Sect. 6 concludes the paper.

2 The Theory

In [8], Beldiceanu specifies a description language which enables mathematicians, computer scientists and programmers of different CLP systems to share information on global constraints in a way that all of them understand. It also helps to classify global constraints, and as a most important feature, it enables us to write programs which, given only this abstract description, can automatically generate parsers, type checkers and propagators (pruners) for specific global constraints.

Beldiceanu has also defined a large number of constraints in the ADL. Most of them are already known, but the slight modification of existing descriptions has resulted in several new constraints. The potential of these modifications arose only with the use of this schema.

Section 2.1 introduces the essential concepts of Beldiceanu's theory, Sect. 2.2 presents the most important features of the ADL, finally Sect. 2.3 illustrates the usage through the simple example of the widely used `element` constraint.

2.1 Representing Constraints as Graphs

In order to create an inter-paradigm platform, Beldiceanu reached for a device that is abstract enough and capable of depicting relations between members of a set: the *directed graph*. Before we can show how graphs can represent global constraints, three concepts have to be introduced:

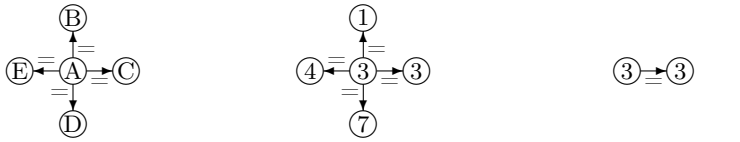
1. The *initial graph* is a regularly structured graph, which is characteristic of the constraint and the number of arguments[1], but is independent from the specific values of the arguments.
2. The *elementary constraint* is a very simple constraint with few arguments, such as $X = Y$.
3. The *graph properties* are restrictions on the number of arcs, sources, connected components, etc.

The description of a constraint specifies how the *initial graph* should be built. Its vertices are assigned one or more variables from the constraint, while the arcs connecting the vertices are generated according to a regular pattern. Finally the chosen elementary constraint is assigned to each arc. The variable belonging to the start point of the arc will become the first argument of the elementary constraint, while the variable assigned to the endpoint will become the second argument. Note that in general, the elementary constraint need not be binary, if it has more arguments, then a hypergraph is built using arcs with the required number of endpoints.

Every distinct instantiation of the constraint arguments results in a separate instance of the constraint. For every such instance, a different *final graph* is derived from the common initial graph by keeping those arcs for which the elementary constraint holds. If a vertex is left without connecting arcs, the vertex itself is also removed. The global constraint succeeds if and only if the specified graph properties hold for this final graph.

The graph of a simplified variant of the `element` constraint can be seen in Fig. 1. This constraint serves as an example throughout this paper, and it is explained in detail in Sect. 2.3. For now, it is enough to know that it succeeds if its first argument, a single variable (denoted by A in the figure), is equal to a member of its second argument, a list of values (denoted by B, C, D and E). The required graph property is that the number of arcs should be exactly one.

[1] As already mentioned, global constraints may (and usually do) have variable number of arguments.



(a) Initial graph **(b)** An instance of (a) **(c)** Final graph

Fig. 1. The graph of the simplified `element` constraint

2.2 The Abstract Description Language (ADL)

The most important feature of the ADL is the ability to describe how the initial graph has to be generated, what is the elementary constraint to be assigned to the arcs, and what graph properties must hold for the final graph.

Beside these, the ADL gives means to limit the set of values to be accepted in the constraint arguments, too. We have to specify the type of each argument, and we may also pose further restrictions on the values. Any concrete application of the constraint that violates these preconditions will be considered as erroneous.

The syntactic order of these language elements in a concrete definition reflects the order in which they are interpreted: the type and value restrictions are followed by the graph generation parameters, finally the required graph properties are specified. The following paragraphs discuss the language features in the very same order.

Argument type restrictions. According to the schema of Beldiceanu, all arguments of the global constraint must be typed. There are three simple data types and a compound type, which are widely used. These are:

`int` is a constant integer;

`atom` is a character sequence (just like a Prolog atom);

`dvar` is a domain variable (which could also be a constant as an special case);

`collection(`$Attr_1$`-`$Type_1$`, `$Attr_2$`-`$Type_2$`, ...)` is an *ordered list of items*, each item being a *set of labeled attributes,* where $Attr_i$ and $Type_i$ are the label and type of the i^{th} attribute, respectively. This type specification does not require the items of such a collection to have all the attributes specified and also allows them to have additional attributes. It only requires the values of the given attributes to have the right type. An example collection and its type specification (taken from [8]) is shown in Fig. 2.

The type `RECTANGLES` corresponds to a collection of rectangles, each rectangle being defined in both dimensions by its origin and either its size or its end. The following is the type definition of `RECTANGLES` and a sample instance of it, that contains two rectangles, one given with its size, the other with its endpoint. (The attributes of each rectangle are separated by spaces, the two rectangles are separated by a comma.)

```
RECTANGLES: collection(ori1-dvar, siz1-dvar, end1-dvar,
                       ori2-dvar, siz2-dvar, end2-dvar)

RECTANGLES = {  ori1-5   siz1-20  ori2-5   siz2-10,
                ori1-25  end1-45  ori2-15  end2-25  }
```

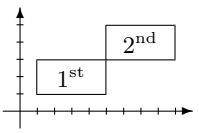


Fig. 2. An example collection

There are other compound data types, too, like `list` or `term`, which are rarely used in the numerous existing constraint descriptions.

Argument value restrictions. In addition to type restrictions it is also possible to specify preconditions on the *values* of the arguments. These conditions can be expressed with the following formulæ:

Name Relop Expression means *Name* must be in relation *Relop* with *Expression*. Here *Name* is the name of either an argument or an attribute of a collection, in the form *Coll*.*Attr*, *Coll* being the collection. *Relop* is a relational operator, like $\neq$ or $\geq$. *Expression* is an arbitrary expression consisting of constants, other names and mathematical operators.

Name `in {`*List*`}` means *Name* (same as before) must appear in *List*, a list of comma separated constants.

`distinct(`*Coll*`/`*Attr*`)` means that all appearances of attribute *Attr* in the collection *Coll* must be distinct, i.e. no two attributes of this name can share the same value.

`required(`*Coll*`.`*Attr*`)` means that all items in collection *Coll* must have attribute *Attr* specified.

There are several other value restricting statements, but those are seldom used.

Graph generation parameters. The initial graph generation consists of three phases. In the first phase the vertices are created, in the second phase they are connected by arcs, and in the third phase, the specified elementary constraint is assigned to each arc.

In the most common case, one has to specify an *input collection* to create the vertices: each element of this collection is assigned to a vertex. The collection may either be a constraint argument or it can be built for this purpose. The created vertices provide the input of the *arc generator,* which manages the second phase. Each generator incorporates a regular pattern, which is reflected in the created set of arcs. The arity of the arcs is characteristic of the generator, and it must also match the arity of the specified elementary constraint.

In general, the arc generator may require the vertices to be divided into disjoint subsets. In that case, not one but several input collections must be specified, each of these is mapped to a separate subset of vertices. Currently, most existing arc generators need a single set of vertices (i.e., one collection) as an input, and there are two of them expecting two.

Figure 3 shows four example arc generators. All of these generators create binary arcs, which means that they can be used with binary elementary constraints only. (This is the most common case.) The `loop` generator connects each vertex to itself. The `path` generator exploits that the collections are ordered lists, and connects the first vertex to the second, the second to the third, and so on. The `clique` generator connects all vertices to all others by default, but it can have a relational operator as an argument, in which case it only connects vertices with indices which sustain the relation. Such a case is shown in the figure. The `product` generator gets two sets as an input, and connects all the vertices in the first set to all the vertices in the second set. This generator can also have a relational operator as an argument.

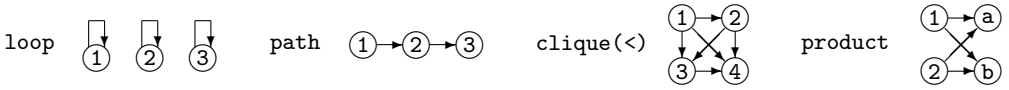


Fig. 3. Arc generators

The elementary constraint, the third ingredient of the graph generation, is basically a mathematical relation containing symbolic references to values assigned to the vertices (i.e., the endpoints of the arcs).

A constraint definition contains three terms to specify this information. We have to determine the input collection(s), select the arc generator by its name, and define the elementary constraint assigned to the arcs.

Graph property requirements. These statements also have the form of an equation, with a graph property name on the left hand side, constants and arguments of the global constraint on the right. Let us see several graph properties:

`nvertex`	is the number of vertices;[2]
`narc`	is the number of arcs;
`ncc`	is the number of connected components;
`nscc`	is the number of *strongly* connected components;
`nsource`	is the number of sources (those vertices which do not have arcs leading into them);
`nsink`	is the number of sinks.

2.3 An Example – The `element` Constraint

The `element` constraint is one of the most common global constraints. It receives a single item and a set of items as arguments and it succeeds iff the item is a member of the set. In some implementations, both the item and the elements of the set may be domain variables, but in the following interpretation the elements of the set must be constants. The formal definition of `element` according to [8] is shown in Fig. 4, an instance of its graph with specific arguments was presented in Fig. 1. It can be explained as follows.

1. The `element` constraint has two arguments (line 1).
2. The first, called `ITEM` is a collection with two attributes, `index` and `value`, both are domain variables (line 2). The second argument, called `TABLE` is also a collection with two attributes, also called `index` and `value`, but these are constants (line 3).
3. The following restrictions must hold:
   - both attributes of both collections must be specified in all items (lines 4–5);
   - there must be exactly one item in `ITEM` (line 6);
   - the indices in both collections must be between 1 and the size of `TABLE` (lines 7–8);
   - all indices in `TABLE` must be distinct (line 9).
4. The arc generator is `product` (line 11), which requires two collections as its input, namely `ITEM` and `TABLE` (line 10).
5. The elementary constraint assigned to the arcs appears in lines 12–13. It is to be read like this: the value assigned to the first endpoint of the arc (`[1]`) is a member of the `ITEM` collection, and its attributes labeled as `index` and `value` must both be equal to the equivalent attributes of the value assigned to the second endpoint (`[2]`), which is a member of the `TABLE` collection. The syntax looks a bit weird and perhaps even confusing, this question is addressed in Sect. 3.1.
6. The number of arcs must be exactly 1 in the final graph (line 14).

Note. It might seem strange to define `ITEM` as a collection when it must have exactly one element (line 6). However, passing the index and value as two separate arguments of the constraint would be less symmetric with respect to `TABLE`. Another advantage is that `ITEM`, being a collection, can serve directly as an input for the `product` arc generator.

[2] This property is sensible to examine, because unconnected vertices are removed from the graph, therefore it is not necessarily equal to the size of the input collection.

```
1  Constraint:      element(ITEM,TABLE)

2  Arguments:       ITEM:  collection(index-dvar, value-dvar)
3                   TABLE: collection(index-int, value-int)

4  Restrictions:    required([ITEM.index,ITEM.value]),
5                   required([TABLE.index,TABLE.value]),
6                   |ITEM| = 1,
7                   ITEM.index ≥ 1, ITEM.index ≤ |TABLE|,
8                   TABLE.index ≥ 1, TABLE.index ≤ |TABLE|,
9                   distinct(TABLE/index)

10 Arc input:       ITEM, TABLE
11 Arc generator:   product
12 Arc constraint:  ITEM.index[1] = TABLE.index[2] ∧
13                  ITEM.value[1] = TABLE.value[2]

14 Graph property:  narc = 1
```

Fig. 4. The `element` constraint in abstract syntax

3 The Concrete Syntax

In order to be able to put the theory into practice, we had to define a concrete syntax of the language. The chosen representation closely resembles the abstract syntax, but follows the syntax of Prolog, too. This has the advantage that it can be effortlessly parsed by a Prolog program.

This work has helped to discover weaknesses of the ADL. First, it turned out that the semantics of the `distinct` operator is unclear in certain contexts, because it is under-specified. Second, as it was already noted at the end of the previous section, the syntax of the elementary constraint specification can be confusing.

Section 3.1 covers the two problematic issues and suggests a solution to both. Section 3.2 illustrates the concrete syntax itself, supported by the updated version of the already familiar `element` example.

3.1 Clarifying the Language Specification

The problem of the `distinct` operator. Let us consider the following type declaration of a collection of collections:

```
COLL: collection(c-collection(val-int))
```

Such a data structure can be used to model different data in different constraints. Two of these are the following:

1. A list of sets, where each set must have distinct elements in itself, but the same element can appear in more than one set.
2. A partitioning of a single set, in this case all elements must be distinct, no element can appear twice in the whole construct.

Since their type signature is the same, they must be distinguished with value restrictions. Unfortunately, we find that we cannot express both with `distinct` statements. Moreover, it is unclear which of these `distinct(COLL/c/val)` really means. The inability to exactly specify what values are to be selected leads us to the definition of two concepts in the following paragraph.

Selectors and designators. When we refer to attributes of items of collections, sometimes we want to reach single values, in other cases we need the list of values of all items within the collection. The `required` and `distinct` operators are good examples of the two possibilities, respectively.

Keeping the notation of [8], which uses the name designator to refer to a sequence of names selected by slashes, let us introduce two new concepts, defined with BNF notation as follows:

selector ::= *Coll* | selector . *Attr*
designator ::= selector | designator / *Attr*

Selectors can be used to point out single values. They state something about values of items of a collection *separately*. Designators, on the other hand, point to a list of values. They state something about the values in all the items of a collection *together*.

By starting a designator with a selector, we express that we want to divide the list of all the values into sublists and state something about these sublists separately. The division points are determined by the selector part of the designator. To clarify this, Fig. 5 shows a somewhat degenerated collection as a tree, along with two designators and the corresponding sublists marked with ovals.

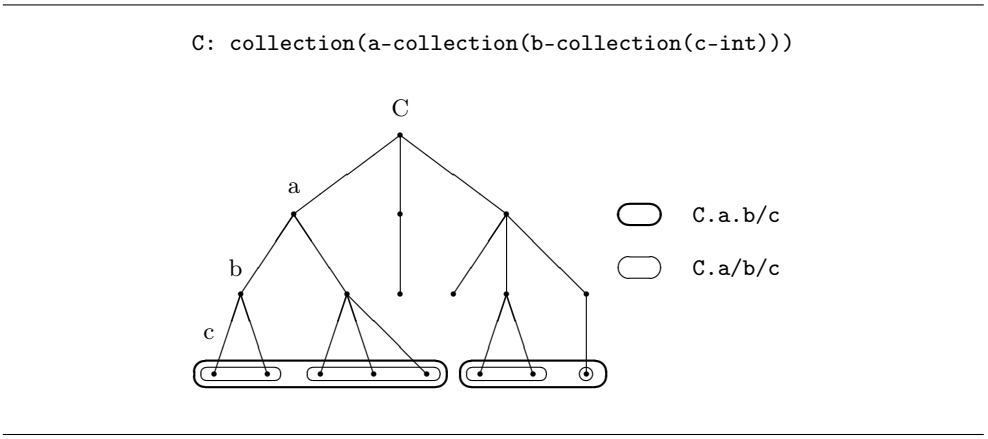


Fig. 5. The meaning of designators

In certain contexts only selectors are accepted. Among others, such places are where the dot notation was already used, such as the argument of `required`, or `TABLE.index` $\geq$ `1` in Fig. 4. In the latter we want to express that all values labeled as `index` must be greater than or equal to 1, separately.

Elsewhere designators are required. The argument of `distinct` is such a place. One would also use a designator to count those items of a collection which possess a certain attribute. Then one needs to write `|COLL/attr|`, because `COLL/attr` brings all the items with `attr` attribute together into a list, and `|...|` returns the length of this.

Now let us return to the problem of `distinct`. In the example presented there, the statement `distinct(COLL.c/val)` means that for all `c` collections separately, the `val` values must be distinct, but the same value can appear in more than one collection (case 1). `distinct(COLL/c/val)`, on the other hand, means that the *list of all values* in all collections must be distinct (case 2).

The elementary constraint notation. As we have seen in Fig. 4, `ITEM.value[1]` means the value of the `value` attribute of the first argument which is the `ITEM` collection. Thus we can say that the general form is something like *Coll*`.`*Attr*`[`*ArgIndex*`]`. This is rather confusing and does not resemble any of the notations we are used to:

– the specification of `ITEM` is redundant, because it is well known from the graph structure that the first argument of the elementary constraint *must be* an item of that collection;
– the position of the index `1` between the brackets is misleading because this notation suggests some kind of array indexing, which is not the case.

We would be better off with a notation like `Args[1].value` (where `Args` would be an array of all arguments of the elementary constraint) or `Arg1.value`. As we will see, the concrete syntax uses a very similar notation.

3.2 The Prolog-like Concrete Syntax

As stated in the introduction of Sect. 3, the chosen representation, while closely resembling the abstract syntax, follows the syntax of Prolog. Hence, each global constraint is described by a Prolog clause with seven arguments, as shown in Fig. 6 for the `element` constraint. These arguments are the following:

1. the name and arguments of the global constraint (as a Prolog term);
2. the list of type restrictions of the form *Arg*`-`*Type* where *Arg* is the name of the argument and *Type* is the type specification;
3. the list of value restrictions in a form very similar to the abstract syntax, with the exception of `|COLL|` which should be written as `size(COLL)`, and all the relational operators must be written in Prolog notation;
4. the arc generator input (a list of collections);
5. the name of the arc generator;
6. the elementary constraint in the form *Args* `=>` *Body*, where *Args* is a collection of the arguments of the elementary constraint and *Body* is the constraint itself (`#=` and `#/\` are operators of the host language, basically they mean $=$ and $\wedge$, respectively);
7. the list of graph properties.

Inline collections have a somewhat different syntax than the one in Fig. 2. They can be written as follows (note the difference in the use of commas and semicolons):

– a collection has the form `{` *Item1* `;` *Item2* `;` ... `}`;
– each *Item*i above has the form *Att1*`-`*Val1* `,` *Att2*`-`*Val2* `,` ... where *Att*i is an attribute name and *Val*i is a value.

The lines of Fig. 6 correspond respectively to the lines of Fig. 4, and the definition as a whole should be self-explanatory. However, two things are worth mentioning.

One is that we have chosen to represent the arguments of the elementary constraint as items of a collection. In the body, we need to refer to these items and their attributes. However, there no syntax is defined to access the attribute of a single item of a collection, therefore we call for a trick. `{A}` means a collection with a single item (`A`), therefore `{A}.index` will expand to the index of this single element.

The other point to note that the relational operator `#=` comes from the SICStus CLP(FD) library, therefore, after expanding the selectors, the statement will become a valid CLP(FD) expression. The advantages of this will be discussed in Sect. 4.2.

4 The Implementation

The schema created by Beldiceanu allows us to test whether the relation expressed by a global constraint holds for a given set of concrete arguments. However, it does not deal with the more important case where only the domains of the arguments are specified, but their specific values are unknown. In such a case we need an algorithm to prune the domains of the arguments by deleting those values that would certainly result in a final graph not satisfying the properties. This question is fundamental in practical applications, therefore it is addressed by this section.

```
graphfd:global(element(Item, Table),

       [  Item-collection(index-dvar, value-dvar),
          Table-collection(index-int, value-int)         ],

       [  required(Item.index), required(Item.value),
          required(Table.index), required(Table.value),
          size(Item) =:= 1,
          Item.index >= 1, Item.index =< size(Table),
          Table.index >= 1, Table.index =< size(Table),
          distinct(Table/index)                          ],

       [  Item,Table                                     ],
          product,
          {A;B} => {A}.index #= {B}.index #/\
                   {A}.value #= {B}.value,

          narc = 1).
```

Fig. 6. The `element` constraint in concrete syntax

Development was launched with two goals in mind. The first task was to implement a relation checker, a realization of the testing feature offered by the schema, and a dumb propagator built on this checker. By and large, this task is finished, the results are presented by Sect. 4.1.

The second task was to implement an direct propagator capable of pruning variable domains based on an analysis of the current state of the graph, with the required properties in view. This task is much bigger, the development is still in an early stage. Its current state and features are introduced by Sect. 4.2.

Both tasks are implemented in SICStus Prolog [9], extending its CLP(FD) library by utilizing the interface for defining global constraints. This allows thorough testing of both the program and the theory itself in a trusted environment.

4.1 The Complex Relation Checker and the Generate-and-Test Propagator

The first stage was to implement the complex relation checker, a program that checks whether the relation defined by the global constraint holds for a given set of values, but does no pruning at all. It includes the following features:

- complete type checking (`dvar` is interpreted as `int`);
- full support of selectors and designators introduced in Sect. 3.1;
- support for value restriction with the most frequent statements:
  - `distinct` and `required`; plus
  - arbitrary Prolog calls which must succeed for the restriction to hold;
  - `size(...)` is replaced by the length of a collection or list.
- full set of built-in arc generators;
- extensive set of supported graph properties.

When called, the relation checker is given a constraint with fully specified arguments, and reports the result of the type check, the restriction check, and whether the graph properties hold for the final graph. The output of two example runs can be seen in Fig. 7. In the first case, the first argument appears in the collection passed in the second argument, while in the second case it does not.

```
Testing element({index-2,value-3}, {index-1,value-1 ; index-2,value-3}).
Type checking passed.
Type restrictions held.
Graph properties held.
Relation is sustained.

Testing element({index-2,value-1}, {index-1,value-1 ; index-2,value-3}).
Type checking passed.
Type restrictions held.
Graph properties failed.
Relation is not sustained.
```

Fig. 7. Output of the complex relation checker

The checker was used to test the formal description of several constraints, whether they really conform to their expected meaning, and some errors in their specification have already been discovered (these will not be discussed here).

The second stage was to amend the relation checker with a generate-and-test propagator. The idea is that whenever the domain of a variable changes, all possible value combinations of the affected constraint's arguments are tested with the relation checker, and only the values that passed the test are preserved. This classical but extremely inefficient method for finding solutions gives us full and exhaustive pruning.

The usage and output of the generate-and-test propagator is the same as that of the direct propagator, which is introduced in the next section (see Fig. 8).

4.2 The Direct Propagator

Generate-and-test propagation is naturally out of the question in any practical applications. The direct propagator is the first step towards an efficient, applicable pruner. Here the line of thought is reversed: we assume that the constraint holds, and from the required graph properties we try to deduce conclusions regarding the domains of its variables.

Propagation in theory. The question that naturally arises is the following: how the changes of domains can be propagated given a graph representation of the constraint. As mentioned in the Introduction, constraints behave like daemons which wake up when the domain of the affected variables change. The propagator – using the programming interface of CLP(FD) – can be set up so that it is notified whenever a constraint wakes up. On these occasions it must check the graph corresponding to the constraint and classify its arcs into three groups:

1. arcs with the assigned elementary constraint being *known to hold* – i.e., they will appear in the final graph;
2. arcs with the assigned elementary constraint being *known to fail* – i.e., they will be left out of the final graph;
3. arcs with the assigned elementary constraint being yet uncertain.

This classification can be completed gradually because the CLP(FD) system is monotonic, which means that values can only be removed from a domain. As a result, a value is removed only if it is definitely not a solution, because it cannot be re-added later.

To propagate the constraint, we have to look at this semi-determined graph and the required graph property together, and try to tell something about the still uncertain arcs. This process is called the *tightening* of the graph. In order to ensure that the graph properties hold, some of

the uncertain arcs must be removed from the final graph, others must be made part of it. This causes the corresponding elementary constraints to be forced into success or failure, thus pruning the domains of the variables. The global constraint finally becomes entailed when there are no uncertain arcs left.

Because of the character of this propagation algorithm, elementary constraints are chosen to be *reifiable* CLP(FD) constraints, as already mentioned in the last paragraph of Sect. 3.2, an example of these is shown in Fig. 6. Reifiable constraints are connected with a Boole variable, and succeed *if and only if* the Boole variable has a value of 1. This use of reifiable constraints has several advantages. For one, a wide range of predefined constraints is available, already at this early stage of development. For another, the algorithm must be able to determine whether an elementary constraint holds or fails, or force it into success or failure, and the Boole variable linked to the reifiable constraints serves exactly that purpose.

To figure out how to tighten the graph at each step, that is, to find the rules of pruning, we need to study each graph property separately. There are simpler properties, such as prescribing the number of arcs, for which finding these rules is not very problematic (see below). A few of these are already handled by the propagator. The are more complex properties, like constraining the difference in the vertex number of the biggest and smallest strongly connected components, the pruning rules for these are a lot more complicated.

Propagation of the `narc = N` property. Let us assume that the required graph property is `narc = N`, where N is a positive integer. Let S be the set of arcs which are known to be part of the final graph, and let U denote the set of the still uncertain arcs. Then we have must take the following action:

- if $|S| > N$, fail, because there are already to many arcs;
- if $|S| = N$, force every arc in U to failure;
- if $|S| + |U| < N$, fail, because N cannot be reached any more;
- if $|S| + |U| = N$, force every arc in U to success;
- otherwise do nothing.

Example run. Running the direct propagator on the `element` constraint is possible because it uses exactly this graph property. An example run can be seen in Fig. 8. The first call determines that the `A-B` element must appear in the list, and we get that `A` must be between 1 and 3, while `B` must be either 2, 6 or 9. The second call we also ask the CLP(FD) environment to enumerate all solutions by labeling the variable `A`, and, as we could expect, we get the three correct solutions.

```
| ?- graph_global(element({index-A,value-B},
       {index-1,value-6 ; index-2,value-2 ; index-3,value-2})).
A in 1..3, B in{2}\/{6} ? ;
no

| ?- graph_global(element({index-A,value-B},
       {index-1,value-6 ; index-2,value-2 ; index-3,value-2})),
     labeling([], [A]).
A = 1, B = 6 ? ;
A = 2, B = 2 ? ;
A = 3, B = 2 ? ;
no
```

Fig. 8. Running the direct propagator

The current implementation can handle four graph properties, these are `narc`, `nvertex`, `nsource` and `nsink`. Fortunately, a large number of descriptions relies only on the first two, thus many different constraints can already be propagated. Without going into details, such constraints are `among`, `disjoint`, `common`, `sliding_sum`, `change`, `smooth`, `inverse`, and variants of these.

Current work is concentrated on the perfection of the propagation of these properties, and on the study of the `nscc` (number of strongly connected components) and related properties, which are also heavily used in the existing descriptions. The rest of the properties are only required by a minority of the constraints.

This propagator, although still not efficient enough to be useful in practical applications, may serve as a prototype for more effective implementations. A few thoughts on this issue are shared in the next section.

5 Future Work

Pruning rules for more of the graph properties are to be worked out. The existing rules also need to be improved in certain cases. This will be the objective of an international project hopefully starting in Autumn 2003.

Using reifiable constraints as elementary constraints poses a problem: they do not necessarily provide a pruning as strong as expectable. Such a case can be seen on Fig. 9. What we see here, is that 1 is not excluded from the domain of `A`, although it could be. The problem is that forcing the and-ed elementary constraint of `element` (Fig. 6, lines 12–13) into failure is not enough to do that. We would get better pruning if we could write something like:

arc exists $\Longleftrightarrow$ *indices are equal*
arc exists $\Longrightarrow$ *values are equal*

But implication does not conform with the concept of elementary constraints. This problem requires further study.

```
| ?- graph_global(element({index-A,value-B},
       {index-1,value-6 ; index-2,value-2 ; index-3,value-2})),
     B #= 2.
B = 2,
A in 1..3 ?
```

Fig. 9. Weak propagation of reifiable constraints

Efficiency matters need to be considered more carefully when implementing further propagators. One way to increase efficiency, as suggested by Beldiceanu, could be to abandon the thought of a common propagator, that is able to parse such descriptions and prune in run time, and implement a *pruner algorithm generator* instead. This generator would take the description and convert it into a piece of code that does the pruning. This would shift the execution of complicated graph algorithms into compile time, where efficiency is a smaller issue. How this can be done must be worked out yet.

6 Conclusions

The paper began with the introduction of a theory first described in [8] that enables us to represent global constraints as regular graphs of the same elementary constraint. It was shown how the

definition of a global constraint looks like, what restrictions and requirements may appear in it, and how the representative graph is built by it.

Then the concrete syntax of the language used by the implementation was presented. First, attention was drawn to two problems with the ADL specification, and solutions to them were suggested, too. Second, the concrete syntax itself was illustrated.

The last part of the paper introduced the actual results of my research through the programs developed as case studies. One of these was the complex relation checker capable of testing whether a constraint holds for a given set of values. The other, based on this, was the generate-and-test propagator, which implements exhaustive propagation for a large number of graph properties, but with very low efficiency. The result of the third, more interesting approach was the direct propagator, which was considered as a step towards efficient algorithms of constraint pruning. This deals with semi-determined constraint graphs and the enforcement of uncertain arcs in order to satisfy the required graph properties.

Acknowledgements

I would like to thank Nicolas Beldiceanu for his theory and his ideas on the propagation of graph properties. Peter Szeredi, my tutor always directed my attempts at research and writing with patience yet with great tenacity. Thanks are also due to the anonymous reviewer whose remarks led to major improvements in the paper.

References

1. Jaffar, J., Michaylov, S.: Methodology and implementation of a CLP system. In Lassez, J.L., ed.: Logic Programming: Proceedings of the 4th International Conference, Melbourne, MIT Press (1987) 196–218 Revised version of Monash University technical report number 86/75, November 1986.
2. van Hentenryck, P., Michela, L., Perron, L., Régin, J.C.: Constraint programming in OPL. In Nadathur, G., ed.: Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99). Volume 1702 of Lecture Notes in Computer Science. (1999) 98–116
3. ILOG: ILOG Solver 5.1 User's Manual. ILOG s.a. `http://www.ilog.com`. (2001)
4. Loia, V., Quaggetto, M.: Embed finite domain constraint programming into Java and some Web-based applications. Software— Practice and Experience **29** (1999) 311–339
5. Smolka, G.: Constraints in OZ. ACM Computing Surveys **28** (1996) 75
6. van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge, Massachusetts (1989)
7. Diaz, D., Codognet, P.: A minimal extension of the WAM for clp(FD). In Warren, D.S., ed.: Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, The MIT Press (1993) 774–790
8. Beldiceanu, N.: Global constraints as graph properties on a structured network of elementary constraints of the same type. In: Principles and Practice of Constraint Programming. (2000) 52–66
9. Swedish Institute of Computer Science Uppsala, Sweden: SICStus Prolog User's Manual. (2003) http://www.sics.se/isl/sicstuswww/site/documentation.html.