Implementing Global Constraints as Structured Networks of Elementary Constraints

Dávid Hanák

Budapest University of Technology and Economics
dhanak@inf.bme.hu

CS^2, Szeged

July 1–4, 2002

1. Introduction

CLP

- stands for *Constraint Logic Programming*;
- denotes a family of programming languages used for finding values in various domains satisfying a set of relations (*constraints*);
- has several branches: CLP(B), CLP(Q/R), CLP(FD), CHR;
- is usually embedded into a *host language*, like Prolog.

CLP(FD)

- variables are represented by *finite sets of interger values* and
- connected by the constraints propagating changes in their domains;
- solutions can be enumerated by *labeling*;
- constraints can be *global constraints* and *indexicals*.

```
| ?- A in 4..7, B in 0..10, A*2 #= B, labeling([], [A,B]).
A = 4, B = 8 ; A = 5, B = 10 ; {no}
```

Global constraints as structured networks of elementary constraints

- theory by Nicolas Beldiceanu (SICS);
- implementation in SICStus Prolog by Dávid Hanák (BUTE).

2. Representing constraints as graphs

Initial graph

- an initial graph is generated from the constraint;
- every argument (variable) is assigned to a vertex;
- arcs are generated according to a regular pattern;
- arcs (directed edges) can be unary (!), binary, tertiary etc.;
- *elementary constraints* correspond to arcs.

Elementary constraints

- are easily and quickly tested;
- can be forced to succeed or fail;
- are implemented by *reifiable indexicals*.

Final graph

- includes arcs for which the elementary constraints hold;
- includes vertices which have at least one arc connected;
- is required to satisfy certain properties;
- graph properties are restrictions to the number of arcs, vertices, sources, connected compontents, etc.

3. The description language in theory

Type checking

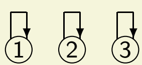
- arguments of constraints are type checked;
- simple data types: int, atom and dvar;
- *collection:* an ordered list of items, each item having a set of labeled attributes;
- some other infrequently used types (list, term).

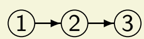
Value restrictions

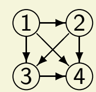
- additional conditions on the values of the arguments;
- *name relop expression*;
- `distinct`(*attribute*);
- `required`(*attribute*);
- and much more...

Arc generators

- input: one or more collections, the items of which correspond to vertices;
- output: arcs connecting the vertices.

loop 

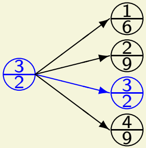
path 

clique(<) 

Example: element constraint

```
Constraint:      element(ITEM,TABLE)
Arguments:       ITEM:  collection(index-dvar, value-dvar)
                 TABLE: collection(index-int, value-int)
Restrictions:    required([ITEM.index,ITEM.value]), |ITEM| = 1,
                 ITEM.index ≥ 1, ITEM.index ≤ |TABLE|,
                 required([TABLE.index,TABLE.value]),
                 TABLE.index ≥ 1, TABLE.index ≤ |TABLE|,
                 distinct(TABLE/index)
Arc generator:   product
Arc input:       ITEM, TABLE
Arc constraint:  ITEM.index[1] = TABLE.index[2] ∧
                 ITEM.value[1] = TABLE.value[2]
Graph property:  narc = 1
```

```
element({index-3 value-2},
        {index-1 value-6,
         index-2 value-9,
         index-3 value-2,
         index-4 value-9})
```



4. Correcting the language specification

Selectors and designators. Assume we have a collection of collections.

- If it is a collection of *sets*, then
  - each set must have unique elements;
  - an element can appear in more than one sets.
- If it is a *partitioning*, each element can appear exactly once altogether.

How can we express this with `distinct(...)`? New concepts:

- selector ::= *name* | selector . *attribute*
  meaning: *for the appropriate values one by one, ...*
- designator ::= selector | designator / *attribute*
  meaning: *for the list of the appropriate values together, ...*

Usage:

- `distinct(SETS.set/val)` – for all sets one by one, values must be distinct;
- `distinct(PARTS/p/val)` – all the values in all the partitions must be distinct.

Arc constraint notation

- `ITEM.value[1]` means *take the* `value` *attribute of the first argument, which is of type* `ITEM` – this is not too fortunate;
- should use something like `Args[1].value` or `Arg1.value` instead.

5. The description language in practice

Constraint definition. A constraint is represented by a clause with 7 arguments. These are:

- the name and arguments of the constraint;
- the list of type checks;
- the list of value restrictions;
- the arc generator input (a list of collections);
- the name of the arc generator;
- the elementary constraint in the form `Args => Body`;
- the list of graph properties to be checked.

Collections

- a collection has the form `{`*Item1* `;` *Item2* `; ...}` where *Item*i is a *record*;
- a record has the form `(`*Att1*`-`*Val1* `,` *Att2*`-`*Val2* `, ...)` where *Att*i is an attribute name and *Val*i is a value;
- the parentheses may be omitted.

```
{ index-1,value-6 ; index-2,value-9 ;
  index-3,value-2 ; index-4,value-9 }
```

Example: element constraint

```
graphfd:global(element(Item, Table),
       [
        Item-collection(index-dvar, value-dvar),
        Table-collection(index-int, value-int)
       ],
       [
        required(Item.index), required(Item.value),
        size(Item) =:= 1,
        Item.index #>= 1, Item.index #=< size(Table),
        Item.value in Table/value,
        required(Table.index), required(Table.value),
        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).
```

6. Version 1: the complex relation checker

Features

- complete type checking (`dvar` is interpreted as `int`);
- full support for selectors and designators;
- partial restriction support:
  - `distinct(...)` and `required(...)`; plus
  - arbitary Prolog calls;
  - `size(...)` is replaced with the length of a collection or list.
- full set of built-in arc generators;
- extensive set of supported graph properties.

Example run

```
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.
```

7. Version 2: the propagator

Embedding into SICStus Prolog

- fitted into the CLP(FD) system of SICStus using the well defined interface;
- this way it can be mixed with "traditional" constraint tools.

Propagation. When the constraint wakes up

- some elementary constraints are known to succeed;
- some are known to fail;
- some of the rest are forced into success or failure.

Example: propagation of the narc $= N$ property

- two sets of arcs: S : known to succeed, U : still uncertain.
- if $|S| > N$, fail;
- if $|S| = N$, force every arc in U to failure;
- if $|S| + |U| < N$, fail
- if $|S| + |U| = N$, force every arc in U to success;
- otherwise can not do anything.

Handling other properties can be a lot more complicated.

Example run

```
| ?- graph_global(element({index-A,value-B},
       {index-1,value-6 ; index-2,value-9 ; index-3,value-2})).
A in 1..3, B in{2}\/{6}\/{9} ? ;
no
| ?- graph_global(element({index-A,value-B},
       {index-1,value-6;index-2,value-9;index-3,value-2})),
     labeling([], [A]).
A = 1, B = 6 ? ; A = 2, B = 9 ? ; A = 3, B = 2 ? ; no
```

Benefits

- a great number of constraints can be described in a dense form using the same formalism;
- the same propagator can handle all of them.

Drawbacks

- it is hard to write thorough propagation for some graph properties;
- some formal descriptions may lead to more complete propagation than others;
- the efficiency of such generic propagator is very low.

8. Conclusions

The relation checker

- verifies the description language itself;
- verifies the formal descriptions of the constraints;
- verification needs proper sets test cases.

The propagator

- validates the completeness of constraint descriptions;
- may serve as a prototype for more effective implementations;
- requires good graph property enforcing algorithms;
- can not be as complete as direct methods.