Complex Constraint Abstraction: Global Constraint Visualization

Helmut Simonis, Abder Aggoun, Nicolas Beldiceanu, and Eric Bourreau

COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex, France
email: Helmut.Simonis@cosytec.com

1 Overview

In this chapter we describe visualization tools for global constraints in the CHIP system. These tools help to understand the behavior of different global constraints, the core feature of the CHIP constraint language. The tools follow a general classification scheme for the use of global constraints in global constraint concepts. Each concept captures the use of a constraint for a particular type of problem, which can be visualized in a specific way. The different visualizers form a class structure of CHIP++ objects, and can be extended or modified by rewriting some callback predicates.

2 Introduction

Debugging of constraint programs take can quite different forms and requires a wide variety of tools. Depending on the host language, the programmer may use a standard debugger or use trace facilities built into the language. It has been reported that debugging with such tools is often time consuming and not very effective [4]. To overcome this problem, ad hoc graphical output has been generated already for the very first constraint programs. In recent years, graphical tools which simplify this task and which make the debugging process more systematic have been developed, in particular for Eclipse [6], OZ [7] and CHIP [9]. We can distinguish the following variants:

– Search tree tools (see chapters **??**,**??**,**??**) represent the search tree generated in a finite domain program in a graphical form and allow user interaction to query the state of the engine at each point. The OZ explorer also provides interactive search facilities, the user can expand nodes in the search tree under his control and thus solve a problem "by hand" while the system performs the constraint propagation. The CHIP search tree tool (see chapter **??**) generates the search tree in a first phase, and allows interaction by re-creating the state of a particular node by re-instantiating variables to certain values.

- The domains of domain variables can often be graphically displayed. Most often this display takes the form of an incidence matrix variables-values, but some systems also use a textual representation.
- Display of the propagation steps in the constraint solver is less common. In the CHIP search tree tool, each propagation step after a value assignment is shown in a graphical form, which indicates the variables affected and the type of update.
- Failure analysis, an important aspect of understanding constraint search behavior, can also be supported by graphical tools.
- The general form of the constraint network is often interesting. Little work has been reported on the display of a constraint network in graph form. The CHIP search tree tool uses an incidence matrix representation, which is particular useful for the representation of non-binary, global constraints.

In this chapter we describe another approach to visualization, the concept based representation of global constraints. Global constraints form the core of the CHIP system, they are high-level abstractions which can be used to express complex conditions between sets of variables in a simple way. They also contain powerful propagation methods to reduce domains and to check consistency. For the development of the visualization tools, the following design aims were stated:

- Multiple use: Each tool must be applicable in multiple instances, so that it can be re-used for many different application problems.
- Use in the search tree tool: The tools must work together with the search tree tool. If a node in the search tree tool is selected, the constraint visualizer should display the state of the constraint in this node.
- Use as debugging tool without the search tree tool: At the same time, the tools will be used outside the search tree tool. An API (application programmer's interface) is provided so that users can update the display at particular points in the application.
- Use as part of application framework: The tools must integrate into the application framework developed by COSYTEC [5]. For simple demos, they can be embedded into applications as sub windows which are defined via the panel package. For complex applications, they can be added as development phase extensions of the application framework.
- Explanation facilities: The tools should try to explain which constraint propagation method has caused a change in the display, so that users can understand problems in models more efficiently.
- Visualize concepts, not code: The global constraints are powerful abstractions, but they do not always correspond to the programmer's view of a particular problem. The same constraint can be used to express many different application concepts. The visualization tools should try to show the constraints in the form of these concepts. This means that one global constraint may have more than one visualization.
- User extendable by rewriting some callbacks: The tools can easily be extended by re-writing/extending some basic callbacks. Rather generic tools can be adapted more closely to particular application domains without major rewrites.

3 Global constraint concepts

In this section we briefly review the difference between global constraints and constraint concepts. Global constraints [1][2] are abstractions which look for a compromise between expressive power and structure. To increase expressive power, the constraint should be as general and flexible as possible, while the structure is needed to allow efficient constraint propagation. In practice, global constraints are usually as general as the methods used inside allow. While this generality increases the usefulness of the constraint and limits the number of different constraint types in a system, it also increases the difficulty of learning and effectively using the constraint. For many applications, it is not necessary for example to understand all 25 pages of the `cycle` documentation to use the constraint. The constraint concepts see the problem from another perspective. Each typical use of a constraint forms a constraint concept, these concepts can be combined in an application in different ways to solve a problem. Some concepts are simply different interpretations of the same constraint, some others are restrictions of a constraint, for example limiting degrees of freedom in some parameter. We will now describe the constraint concepts used in the CHIP environment.

3.1 Cumulative

Some of the concepts for `cumulative` were already discussed in the original paper introducing the constraint [1]. At the moment, we use the following concepts:

- cumulative resource,
- disjunctive resource,
- bin packing,
- producer consumer,
- redundant projection

The cumulative resource is the most commonly used concept for `cumulative`. The items in the constraints are seen as tasks with `start` time, `duration` and `resource` consumption. The x-axis is the time axis, the y-axis is the resource usage. Limits can be placed on the overall end and/or the maximal resource utilization. The disjunctive resource case is a special case of the cumulative resource with a resource limit of 1, where each task has a resource usage of 1. The bin packing concept for cumulative sees tasks as items to be packed into bins. The x-axis denotes the different bins, the y-axis the height of the bins. Each item is specified by its bin assignment, a width which is equal to 1 and a height which corresponds to the size of the item. Producer/Consumer constraints were introduced in [8] to describe the behavior of consumable resources. Tasks either start from time 0 and block a resource amount up to a time point on the x-axis (producers) or they start from a time point and stretch to the end of the time period (consumers) and consume a given amount of resource. The redundant projection is used to strengthen a constraint model by projecting rectangles of a (2D) placement problem onto one axis.

3.2 Diffn

For the `diffn` [2] constraint, a large number of concepts is known.

- disjunctive tasks,
- machine scheduling,
- assignment,
- placement 2d,
- placement with spare,
- placement 3D,
- placement 3D + assignment

The `diffn` constraint can be used to express disjunctive scheduling problems or multiple machine scheduling problems. The x-dimension denotes time, the y-dimension machine allocation. All tasks have `height` 1 to indicate that they use one machine during their execution. Related is the assignment problem, where tasks fixed in time must be allocated to different machines. The `diffn` constraint can also be used for placement problems. The easiest variant is a two-dimensional placement. More complex are the cases where we can use a spare dimension for relaxation, or where we place objects in a three-dimensional space. The last concept uses four dimensions to describe packing problems with multiple containers. One dimension is used to assign an item to one container, the remaining three dimensions control the placement of the item in that container.

3.3 Cycle

The `cycle` constraint [2] also is used in many different ways:

- oriented graph,
- non oriented graph,
- geographical tours,
- tours with machine assignment,
- tours with fixed times,
- tours with time windows,
- loading/unloading
- scheduling with product dependent set-up times,

In its basic form, it is used to find cycles in oriented or non-oriented graphs. If the locations and distances are derived from geographical data, we want to build geographical tours. Additional parameters allow the introduction of machine assignment to the problem. In other problems, start times are linked to the nodes and we are interested in tour planning with time windows. A significant special case is the tour planning with fixed time windows, for example in the case of railway/aircraft rotations. Another extension handles capacity of the tours together with loading/unloading operations at each node. A completely different model uses the `cycle` constraint to express set-up times in multi-machine scheduling problems.

3.4 Among

We also use the `among` constraint [2] in a number of different ways:

- single among,
- overlapping sequences,
- extending sequences,
- multiple among

In the basic constraint, the number of occurrences of a set of values in a list of variables is limited. This constraint can also be applied to overlapping subsequences or to increasing sequences from a start. In its final form, multiple among constraints on different value sets are handled together in the multiple among concept.

4 Principles of operation

In this section we describe the basic implementation structure of the visualizer tools. If you are not interested in extending/modifying some visualizer tool, you may want to skip this section.

4.1 Class Structure

The class visualize is obtained by adding the `list_entry` attributes to the `visualize1` class. All abstract classes are shown *italic* in figure 1 below, they should never be used by the application programmer. You can add new classes either by deriving a new class from an abstract class, or by specializing an existing class.

The classes `visualize_ltsrsb` and `visualize_srsb` differ mainly in their window layout. The first (see figure 2) has a main drawing area, drawing areas to the left and on top and scrollbars to the right and at the bottom. Most structured displays are derived from this class, the left and top area being used for scales or resource display.

The `visualize_srsb` class (see figure 3) has only one main drawing area which is affected by the scrollbars to the right and at the bottom. It is mainly used for drawing concepts of a simpler form.

4.2 Callbacks

The callbacks are stored in attributes of the objects in the form of predicate calls `pred(X)`, where `pred/1` is the callback name and X is the name of the object affected. Overwriting such a callback with a new routine means that the new code is executed instead of the old one. This requires that the new code should emulate all actions of the old code that you want to keep. Alternatively, the code can be structured in such a way that the new code calls the callback of the parent class before/after executing any of its own routines.

visualize

- *visualize_ltsrsb*
  - *visualize_gantt*
    - visualize_disjunctive_tasks,
    - visualize_machine_scheduling,
    - visualize_assignment,
    - visualize_placement_2d,
    - visualize_placement_remains
  - *visualize_profile*
    - visualize_cumulative_resource,
    - visualize_disjunctive_resource,
    - visualize_redundant_projection,
    - visualize_bin_packing,
    - visualize_producer_consumer
  - *visualize_domain*
    - visualize_variable
  - *visualize_measure*
    - *visualize_stack*
      - visualize_local_stack
      - visualize_global_stack
    - visualize_time
- *visualize_srsb*
  - *visualize_drawing*
    - *visualize_graph*
      - visualize_oriented_graph,
      - visualize_non_oriented_graph,
      - visualize_geographical_tours
      - visualize_graph_lines

Fig. 1. Visualize class structure

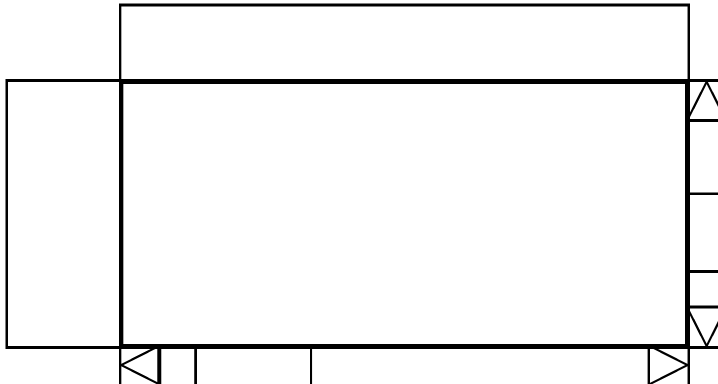

Fig. 2. Visualize_ltsrsb window layout

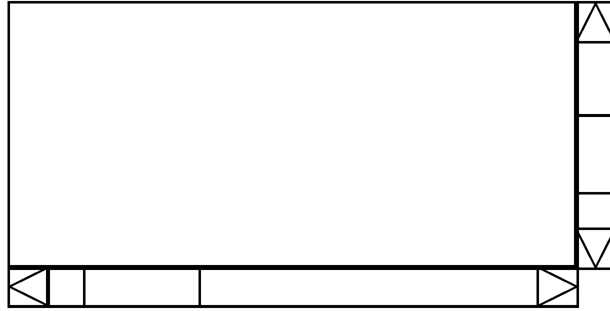

Fig. 3. Visualize_srsb window layout

create_windows This code creates the object, sets attributes and creates all windows of the visualizer. It does not draw in the new windows.

resize_callback This code is executed if the back window of the visualizer is resized. This is used to keep the scrollbars, title and resource windows in constant size after a resize.

reset_callback This code is executed when an existing visualizer is re-created during program execution. It should make sure that default values are set correctly.

init_callback (user code) The init callback is responsible to calculate the displayed area for the constraint. This area should be stored in the `x1, y1, x2, y1, ix1, iy1, ix2, iy2` attributes. The code should not attempt to change the vdc of the windows, that is done in another part of the program. Most visualizers want to modify this code.

draw_callback (user code) The callback is responsible to draw the visualization. The drawing can use all attributes of the visualizer, but most important will be the items attribute which contains the current set of items. These values must not be modified by code.

resource_draw_callback (ltsrsb only) This code draws the resource scale. The default is a numeric scale dependent on the vdc space of the visualizer. This callback only exists for ltsrsb subclasses.

title_draw_callback (ltsrsb only) This code draws the title scale. The default is a numeric scale dependent on the vdc space of the visualizer. This callback only exists for ltsrsb subclasses.

4.3 API

This section describes how to use the visualizer library as an application programmer.

Creation A visualizer object is created by a wrapper around a constraint call inside a CHIP program. The wrapper has the general form

```
visualize(Constraint, Visualizer_type, Attribte_list, Name)
```

The first argument is the call to the constraint, the second argument is the type of visualizer used (an atom), the third is a list of attribute value pairs of the form `A<-B` where A is an attribute of the visualizer and B is the value used. The last argument is the name of the visualizer object. If the constraint call is executed only once in the application, this name should be an atom. If the constraint call is executed multiple times in a recursive loop, then a new name must be generated each time. A typical example is the call

```
visualize(cumulative(Start, Dur, Res, unused, unused,
            Limit, End, unused),
        visualize_cumulative_resource,
        [winw<-500, winh<-400],
        cumulative1)
```

which generates the `cumulative_resource` visualizer `cumulative1` working on a `cumulative` constraint and setting values for the window width and height in the tool.

When the constraint is called, the following actions are performed:

- The program generates the visualizer object of the correct class.
- It then creates all windows required.
- All the parameters are set either to their default value or (if given) to the value specified in the attribute list.
- The constraint is posted. If the constraint fails immediately, the state of the constraint at this time is shown. If the constraint delays, nothing is drawn. This particular mechanism is used to allow failure analysis at set-up time.
- The internal constraint number is remembered. This number is created internally in the constraint solver and is increased every time a new constraint is posted. Remembering the value allows us to refer to a constraint from the search tree tool.
- The constraint call is remembered by storing the call as an variable attribute in the object. Whenever the domain variables in the call change, the new information can be extracted and displayed. If the program ever backtracks over the creation of the constraint visualizer, no further updates of the display are possible.

Note that the objects are not removed when the query terminates. If the same program is run again, the system recognizes the names of existing visualizers and attaches the constraints to the existing windows. In this way, settings on window positions and sizes are not lost from one query to the next. During a debugging session, the same code can be executed multiple times without destroying and recreating the visualization tools.

Update To update the tools, two predicates have been defined. Note that the visualizer tools are also updated automatically inside the search tree tool. In that case, none of these predicates need to be called in the application program.

`visualize_update`

This predicate updates all visualizer tools that are active at this moment. This is the normal way to update the display at a given point in the search.

`visualize_draw(X)`

This predicate updates only the visualizer with the name X. This predicate is useful to display the state of some constraint at different points during the search process, for example to compare the effect of one asignment step.

5 Interface to search tree tool

The CHIP search tree tool is described in chapter **??**. Its function is to capture and visualize the form of a search tree generated by a CHIP search routine. In the most simple case, a built-in search procedure (labeling) is replaced by another built-in, which automatically calls the correct primitives. In a more complex situation, a user written search routine must be annotated with some special predicates to identify how the search should be displayed. The interface from the search tree tool to the global constraint visualizers is transparent. If a visualization tool is active during the search, it will be updated whenever a new node in the search tree is selected. The state of the constraint will be displayed for the node which has been selected.

We now describe what is happening in detail when a node is selected. When the callback on the selection of a node is executed, the following steps are performed:

- The state of the search in this node is re-instated. For this, the path from the root to the node is scanned to find the variable bindings that must be restored. By constraint propagation, the state of all domain variables and constraints is updated.
- The node specific information is displayed in the search tree tool. Depending on the selected view, this may be the display of all variables or of all constraints, etc.
- All currently enabled visualizers are redrawn in the current state of the search.
- The bindings of all variables are undone, we fail until the root of the tree.

6 Use outside search tree tool

The global constraint visualizers can be used outside the search tree tool. In that case, the programmer is responsible of calling the update routine (`visualize_update/0`) at the right point in the search process. If called at every point in tree search, i.e. after each `indomain`, the complete evolution of the constraint is shown. This is useful to obtain a first impression of the behavior of the constraint, but a more detailed analysis will be required to understand failures or missing propagation at some particular step in the search.

7 Cumulative Visualizers

We present different visualizer tools in detail. For space reasons, we can not discuss all visualizers that have been implemented, we restrict ourselves to typical examples.

7.1 Cumulative Resource

Probably the most often used visualizer represents cumulative resources. In this tool, we show three types of objects, profiles, limits and tasks. Figure 4 shows the cumulative resource visualizer for a machine in a scheduling example from the Alvarez benchmark set [3]. We will now explain the different components shown in the diagram.

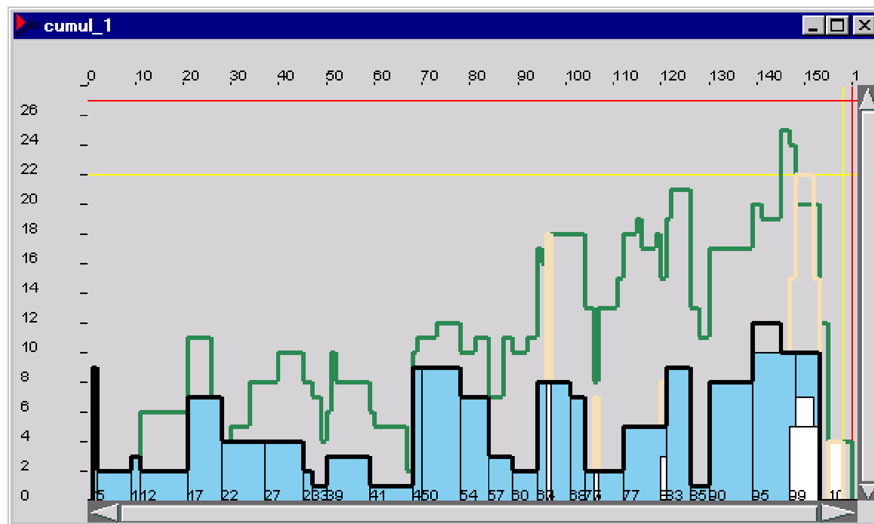

Fig. 4. Cumulative resource view

Profiles There are three different profiles that can be generated from the tasks in the constraints. Each of these profiles approximates the final resource profile, that is generated when all tasks are fixed. The fixed profile (black) is generated from all tasks which are already assigned, i.e. for which start, duration and resource use are integer values. The profile adds up the resource use for each time point. It is given by the function

$$y(t) = \sum_{\substack{i=1| \\ start_i \leq t < start_i + duration_i}}^{n} resource_i$$

where t ranges over the interval

$$[\min_{i=1}^{n} start_i, \max_{i=1}^{n} start_i + duration_i]$$

The obligatory part profile (white) is generated from all tasks for which an *obligatory part* is known. This is the case if the latest start and the earliest end of a task form a non-empty interval. This means that the task is known to be active in this time interval, even if neither start nor duration is fixed. The obligatory part profile is always above the fixed profile, and always below the overall resource limit. This is one of the necessary conditions that the cumulative constraint checks for satisfiability. The profile is given by the formula

$$y(t) = \sum_{\substack{i=1| \\ start_i^{max} \leq t < start_i^{min} + duration_i^{min}}}^{n} resource_i^{min}$$

The notation $start_i^{max}$ denotes the maximum value in the domain of the *start* variable of task i.

If the domains for the start variables are very big, no obligatory parts will exist, but if by constraint propagation the domains are restricted, the obligatory parts start to appear and are taken into account in the constraint propagation. When all tasks are fixed, the fixed task profile and the obligatory part profile are identical and describe the total resource use for all tasks.

We display another profile in the cumulative resource visualizer, the *expected resource use* profile (gray). For this profile we distribute the resource use of a task over the total time period when it can be placed, from the earliest start to the latest end. This approximation of the resource use gives us some indication of resource requirements, even if no obligatory parts exist. The profile is given by the formula

$$y(t) = \sum_{\substack{i=1| \\ start_i^{min} \leq t \\ t < start_i^{max} + duration_i^{max}}}^{n} \frac{resource_i^{min} * duration_i^{min}}{duration_i^{max} + start_i^{max} - start_i^{min}}$$

The expected profile does not always stay below the overall resource limit and the final profile may be below the expected profile in some intervals. But the profile

is still useful, as it provides important negative information. If the expected profile is zero for some time period, we know that no task can be scheduled in this interval.

Limits For both the overall end and the overall resource limit variable we display the minimum and the maximum values as red and yellow (vertical and horizontal) lines. These limits are also used to determine the overall drawing area for the constraint.

Tasks To indicate the space taken up by each task, they are displayed as gray rectangles as soon as they are completely fixed. If an obligatory part exists, it is displayed as a white rectangle. All tasks are drawn from the bottom of the display, possibly obscuring other tasks. It is unfortunately not possible in the general case to place them as rectangles which do not overlap and which fit under the resource profile.

7.2 Bin Packing

The bin packing visualizer is derived from the cumulative resource visualizer. In this particular case, all task duration values are equal to one. The fixed part profile is generated in the same way as for the cumulative resource tool. As all the task duration are one, the concept of obligatory parts does not exist. Instead, we can place all tasks non-overlapping under the resource profile. For this, we define an additional domain variable for each task in the visualizer, its y-coordinate in the visualizer display. The x-coordinate is given by the `start` variable in the `cumulative`, the width is one and the height is equal to the `resource` value in the `cumulative`. We then state a `diffn` constraint to place the task rectangles without overlapping. Each time we update this visualizer, we again solve the small constraint problem placing the tasks. Figure 5 shows an example bin packing visualizer. All tasks have been placed and are drawn underneath the generated profile. It is interesting to note that we use the constraint system itself to visualize some constraint. We use this feature for other visualizers as well.

8 Diffn Visualizers

The `diffn` constraint has a number of different uses, but the most common is the 2D placement concept. An interesting dual visualizer is the `placement_remains` tool, which shows all tasks which are not yet drawn in the placement visualizer.

8.1 Placement 2D

The 2D placement visualizers shows a set of 2D rectangles in an area which is defined by the minimal and maximal values in the domains of the rectangles.

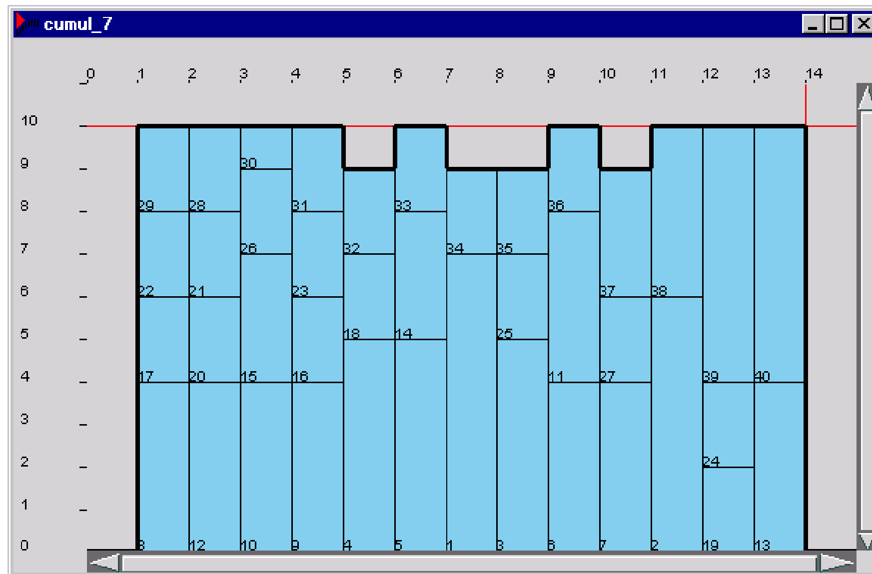

Fig. 5. Bin packing example

If a rectangle is fixed, i.e. x, y, *width* and *height* values are integers, then the rectangle is drawn in gray. If an obligatory part exists, defined in analogous way to the obligatory parts in `cumulative`, then the obligatory part is drawn in white. If an object is selected in another visualizer, the area in which it can be displayed in the `diffn` will be highlighted during the selection. We only use the bounds on the coordinates for this purpose, eventual holes in the domain are not shown. Figure 6 shows a typical visualizer example. Three rectangles are already placed, and there are two obligatory parts shown.

8.2 Placement remains

The 2D placement tool is very useful if many tasks already have been placed. But in the very beginning no tasks will be placed. How can we get an idea about the rectangles that have to be placed? The `placement_remains` tool provides this information. It displays all rectangles which are not completely placed. Rectangles for which an obligatory part exists are drawn in white, all others are drawn in gray. When we select one of the rectangles, the area in which it can be placed in the placement problem will be highlighted in the `placement_2D` visualizer.

The layout of the rectangles in this tool again is controlled by a constraint program, which itself uses a `diffn` constraint to place all remaining, unassigned rectangles. We do not enforce the domain limits of the rectangles at this point and only ensure that the rectangles are not overlapping. Figure 7 shows a snapshot of the `placement_remains` tool.

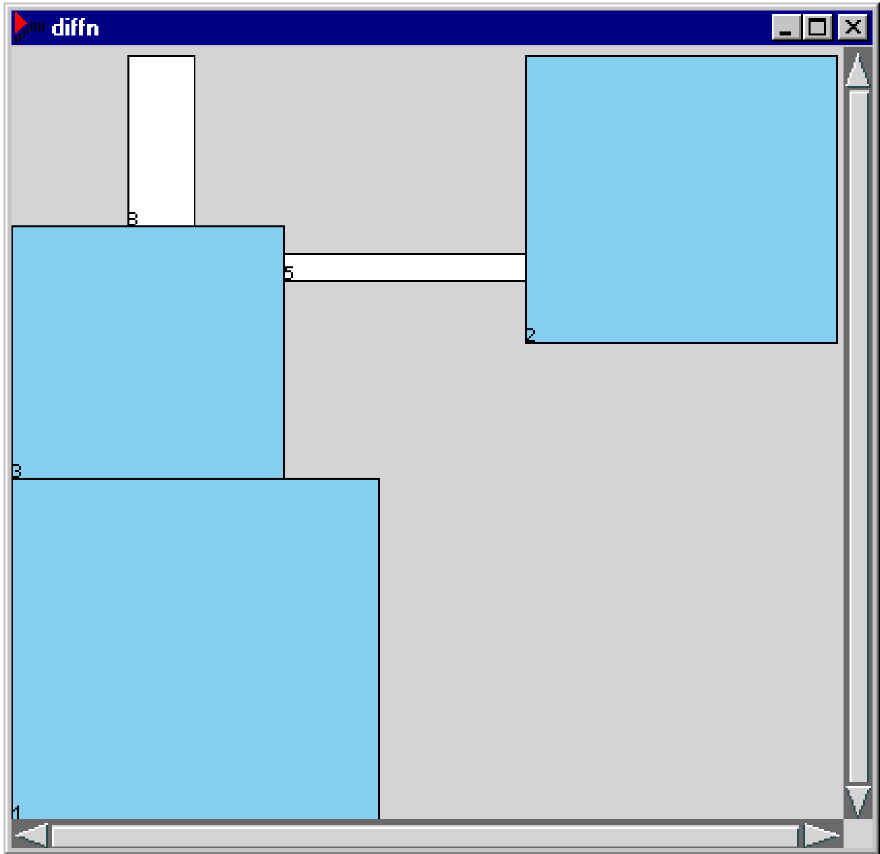

Fig. 6. 2D placement

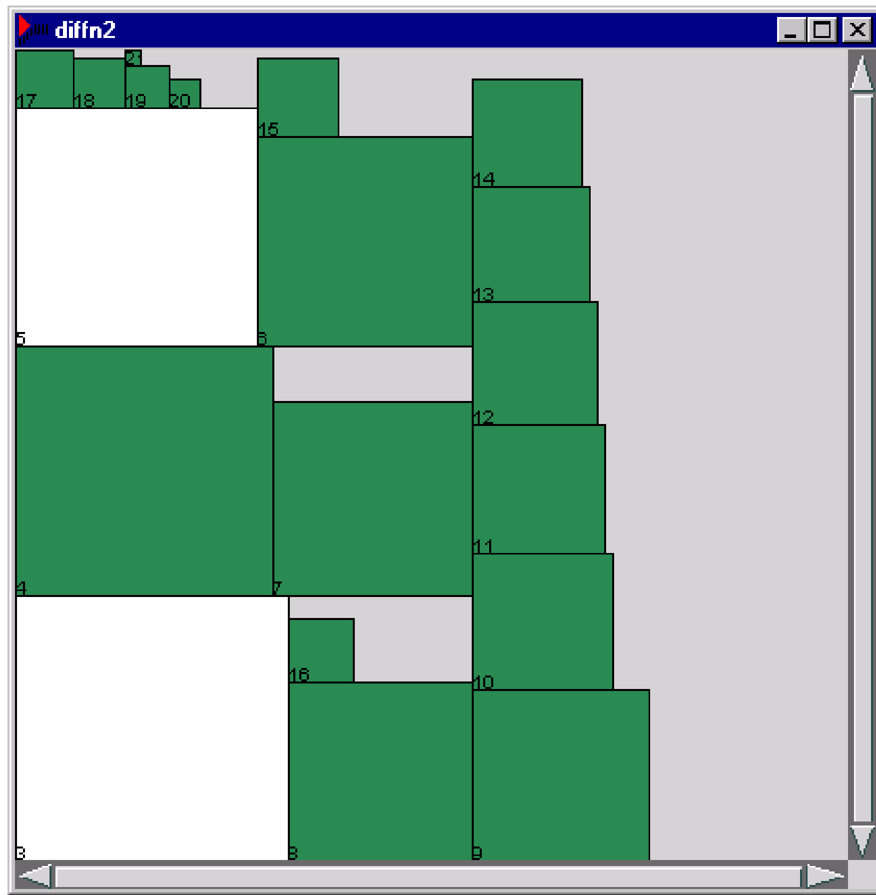

Fig. 7. Showing unplaced items

9 Cycle Visualizer

We show now two quite different interpretations of the `cycle` constraint, one based on a geographical representation, the other on the idea of lines in the directed graph defined by the constraint.

9.1 Geographical tour

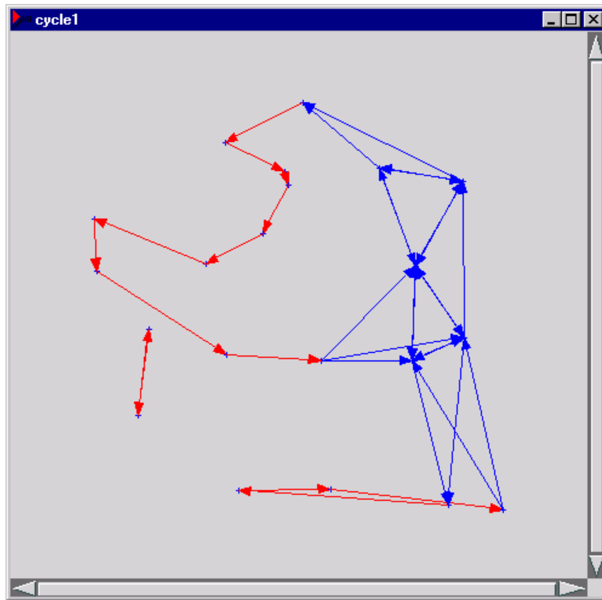

Fig. 8. Geographical tour visualizer

In this tool, the directed graph given in the `cycle` constraint is shown by placing the nodes at co-ordinates which are given by the user as an additional attribute. Typically, these locations will correspond to some geographical location. The connections between nodes are drawn as arrows. If there is only one successor, the arrow is drawn in light gray. If there are still multiple possible successors, the arrows are drawn in dark gray. An additional attribute, `show_domain`, controls the display. If the domain of a node is larger than this value, no successors will be drawn. This helps to identify small, restricted parts of the graph without drawing large numbers of connections to clutter the display. Figure 8 shows an example. A number of nodes are already assigned, there is one closed cycle in the bottom left corner, but the domains of other nodes consist of rather small sets of nodes. By constraint propagation, all links between nodes that are far apart have already been removed.

9.2 Graph lines

The next visualizer uses a different representation of the `cycle` constraint, which is based on the idea of a graph line. A line starts with a node, which has not yet been chosen as the successor for another node and ends with a node for which the successor is not yet given. Initially, all nodes form lines of their own. Whenever a variable is assigned, two lines are merged or a line is closed and a cycle is created. When all variables are assigned, a given number of cycles will have been generated. Figure 9 shows an example, which displays seven lines and one completed cycle. The cycle (drawn in gray) consists of two nodes, numbered 9 and 20. Five lines consist only of one node each. The last node in the line is drawn in white and outlined in black, with the domain of the successor variable printed at the end of the line. In the current state, no line can be closed, as for each line the start node is not in the domain of the end node. The next assignment step will therefore consist in the merging of two lines. At each display, the layout of the lines is re-calculated. This visualizer can quickly give a good idea how the

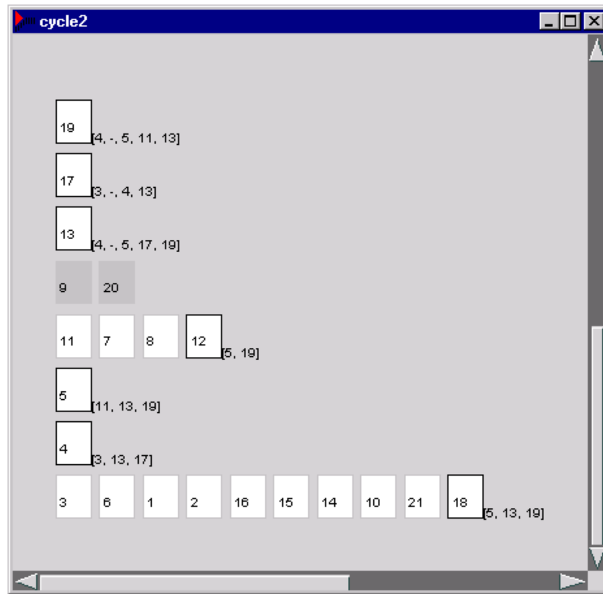

Fig. 9. Graph lines visualizer

assignment routine is working and whether constraint propagation is forcing the merging of lines.

10 Interaction between Visualizers

When considering multiple visualizers at the same time, it is useful to find the relation between the different items displayed. Figure 10 shows an example. We have a two `redundant_projection` visualizers (similar to the `cumulative_resource` tool described above) on the left, one `diffn` `placement_2d` tool one the top right and one `diffn` `placement_remains` tool at the bottom right. Selecting a rectangle in the `placement_remains` tool highlights its position in the other tools. At the moment, the link is done only by argument position. This interaction will be extended to allow customized links between entries in different visualizers.

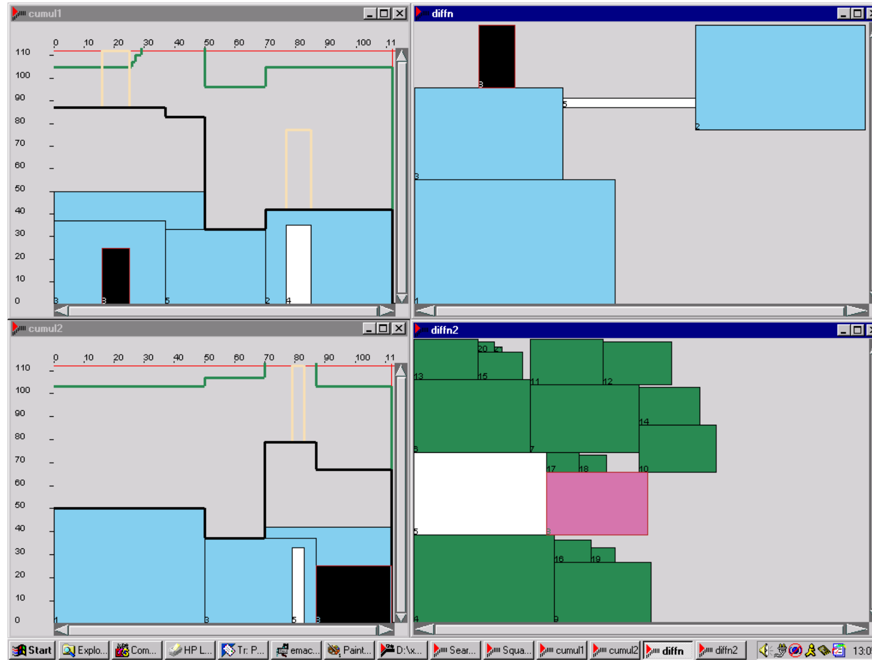

Fig. 10. Visualizer Interaction

11 Conclusions

In this chapter we have described different visualizers for global constraint concepts. One global constraint, like `cumulative`, can be used in multiple contexts, to solve different types of constraint concepts. To understand the behavior of the constraint, it is best to describe the propagation in terms of these concepts as well. The global constraint visualizers are all derived from a fundamental class of visualizer objects. The user can overwrite different attributes and methods

to adapt the tool to his particular needs. Working together with the search tree tool or as a standalone utility, these visualizers allow a better understanding of the global constraints and an easier exploitation of their functionality.

12 Acknowledgement

The work presented here is part of COSYTEC's work package in the DiSCiPl project and was developed using ideas from a number of consortium partners, in particular from UPM. Feedback from a number of early users at COSYTEC was also very valuable.

References

1. A. Aggoun, N. Beldiceanu, Extending CHIP in Order to Solve Complex Scheduling Problems, Journal of Mathematical and Computer Modelling, Vol. 17, No. 7, pages 57-73,Pergamon Press, 1993.
2. N. Beldiceanu, E. Contejean, Introducing Global Constraints in CHIP, Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994.
3. N. Beldiceanu, E. Bourreau, D. Rivreau, H. Simonis. Solving Resource-Constrained Project Scheduling Problems with CHIP, Fifth International Workshop on Project Management and Scheduling, Poznan, Poland, April 1996.
4. M. Fabris et al., CP Debugging Needs and Tools, In Proc. Of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97, Pages 103-122, Linköping, Sweden, May 1997.
5. P. Kay, H. Simonis. Building Industrial CHIP Applications from Reusable Software Components. In Proceedings of the Third International Conference on the Practical Application of Prolog, Paris, France, April 1995.
6. M. Meier, Debugging Constraint Programs, In Principles and Practice of Constraint Programming, page 204-221, Cassis, France, September 1995, Springer, Lecture Notes in Computer Science 976.
7. C. Schulte, Oz Explorer: A Visual Constraint Programming Tool, Proceedings of the Fourteenth International Conference On Logic Programming, Leuven, Belgium, pages 286-300. The MIT Press, July 1997.
8. H. Simonis, T. Cornelissens, Modelling Producer/Consumer Constraints, In Principles and Practice of Constraint Programming, Cassis, France, September 1995, Springer, Lecture Notes in Computer Science 976.
9. H. Simonis, A. Aggoun, Search Tree Debugging, Technical Report, COSYTEC SA, October 1997.