DiSCiPl : Debugging systems for Constraint programming
Long term Research (Task 4.2)
Project Nb: 22532

General report on assessment of the tools

Task TWP1.3: Assessment of tools
Deliverable : report D.W.WP1.3 M1.3

April 30, 1999

Task responsible:
T.Cornelissens, OM Partners, tcornelissens@ompartners.com

Co-authors :
V. Dumortier, OM Partners, vdumortier@ompartners.com
G. Fabris, F. Nanni, A.Tirabosco, ICON, {gfabris, nanni, tirabosco@icon.it}
C. Lai, S. N'dong, PrologIA, {claude, stephane@prologianet.univ-mrs.fr)

Table of Contents

1. Introduction

1.1. Contents

The assessment report D.WP1.3.M3.1 is the main deliverable of task T.WP1.3 *"Assessment of the CP debugging tools"* of the DiSCiPl project. Beside the introduction, the report contains following chapters:

- Evaluation of the CHIP debugging tools by OM Partners,
- Evaluation of the CHIP debugging tools by ICON,
- Evaluation of the Prolog IV debugging tools by PrologIA,
- The final considerations.

All "Evaluation" chapters have the same structure. For every application treated, the users wrote a short description and an evaluation report of the debugging tools used, avoiding references to other applications/paragraphs. The intention is that the reader can easily select the applications he is interested in, and can read this section as a separate topic. As a consequence, some remarks appear more than once, certainly when comparing the contributions of the different partners. However, there is no overlap between the applications themselves and the interests of the different end users in the tools are clearly different.
The last chapter presents the final considerations of the end users. Suggestions for amelioration are included.

A detailed documentation or description of the debugging tools developed by Cosytec and PrologIA is not included in this document. The reader not familiar with the DiSCiPl project should first consult the most recent debugging tool documentation written by the tool developers.

1.2. Task description

During the assessment phase, the end users had to accomplish the following tasks:

- Getting used to the debugging tools, including
  - (re)installation of the latest version(s) of the software,
  - following a theoretical training at Cosytec,
  - studying the documentation and examples provided, and
  - starting the first tests on simple academic examples.
- Testing the debugging tools on existing and new real life applications. The tools have been used by novice as well as experienced users. During this testing period, the end users wrote a report containing their experiences and remarks.
- Writing the final conclusions and finishing the assessment report.

1.3. Meeting the needs

During T.WP1.1, the CP debugging needs have been described in the report D.WP1.1.M1.1 – Part 1. Two main categories have been distinguished by the end users:

- Correctness debugging, including the detection of missing/incorrect answers, and
- Performance debugging.

Regarding these debugging needs and the tools on which research was concentrated during the DiSCiPl project, the end users had the following requirements:

- Graphical tools: the end users expected these tools to be helpful for correctness as well as performance debugging. Graphical debugging tools should be able to:
  - Describe the search tree and labeling strategy
  - Show the values of the variables in order to understand domain reduction and constraint failure

- Show relationships between variables in order to describe the constraint semantics, interaction and propagation.
- Allow multiple levels of abstraction to gain a very precise view or a global picture, according to the need of the moment and the subject of the visualization.

It was clear from the reports and demonstrations of the graphical debugging tools delivered at the end of the development phases, that the end user requirements were met, and even more, that a lot more graphical views had been developed than originally foreseen. As a result most of the assessment report is dedicated to the testing of the graphical debugging tools.

- Assertion based and declarative debugging tools: the end users expected these tools to be helpful for correctness debugging. However, due to the complex nature of problems solved by CP programming, it was not clear for the end users what could be expected or required. Experienced CP programmers know that it is very difficult to judge if a solution given by a CP program is correct without graphical representation of the solution. It was decided to consider this part of the project as more experimental and research oriented.
  During the DiSCiPl project, two independent tools, based on a common assertion language designed in the project, have been developed. PrologIA has developed and included an assertion debugger in Prolog IV. The end users discuss this debugger in chapter 4. Also, a prototype of an advanced assertion debugger for CHIP has been developed (CHIPRE) and demonstrated by UPM in collaboration with Cosytec and Linkoping. The results obtained so far with CHIPRE are quite promising, but the current version of CHIPRE is not robust enough to be tested by the industrial end users. However, the impression of the end users is that the assertion debugger could be very helpful to detect and debug different kinds of errors in large, complex and modular programs. More details on the kind of errors potentially detected by CHIPRE can be found in *[1]*and *[2][2]*.

After having used the new debugging tools, the industrial end users conclude that these tools are a real answer to the debugging needs allowing the end users to optimize their new and existing applications.

2. Evaluation of the CHIP debugging tools by OM Partners

This chapter starts with an overview of the applications examined by OM Partners, together with the debugging tools used and the most important remarks of the end users. The following sections contain a detailed problem description and evaluation of the debugging tools used, for each application separately. The last section of this chapter summarizes the conclusions of both novice and expert user.

2.1. Summary of the applications

Academic examples and exercises of the CHIP course

Application: Send more money
Programmer: OM Partners
Tools used: search tree tool
Use of the information: study of the efficiency of the constraint model and labeling routine
Remarks: difficulties with the numbering of the variables in the search tree tool and the meaning of the coloring of the search tree nodes in the search tree.
Shortcomings/extensions: an indication of the amount of shallow backtracking.

Application: Production sequencing (*cycle* constraint and minimization of the setup cost)
Programmer: OM Partners
Tools used: search tree tool combined with *visualize_graph_lines*
Use of the information: study of the constraint propagation due to the *cycle* constraint and of the variable choice in the labeling routine. The visualizer clearly leads to a better and more intuitive understanding of the problem.
Remarks: -

Shortcomings/extensions: -

Application: Rectangle placement (*diffn* constraints, minimization of the surface of the enclosing square)
Programmer: OM Partners
Tools used: *visualize_placement_2d* and *visualize_placement_remains* with/without the search tree tool
Use of the information: study of different labeling strategies. The user became aware of his incorrect interpretation of the *labeling/4* routine.
Remarks: using the search tree tool required quite some change in the labeling routine.
Shortcomings/extensions: the visualize tools used on their own have a different (better) behavior than when used in combination with the search tree tool, due to the fact that new constraints introduced during the search are not taken into account in the search tree tool. Some suggestions for the standalone use of the visualizers are given.

Application: Small scheduling demo (*cumulative* constraint and minimization of the make span)
Programmer: OM Partners
Tools used: *visualize_cumulative_resource*, *visualize_variable* and *visualize_placement_2d (*+ the search tree tool)
Use of the information: the visualizers give a good idea of the real use of the cumulative resource and of the assignment of tasks.
Shortcomings/extensions: Some amelioration of the diffn visualizer is suggested.

Real-life applications

Application: Production scheduling of perishable products, taking into account the non-simultaneous use of pumps and the availability of personnel and machines. Minimization of setup and lateness costs. The existing application has been rewritten by using a cycle constraint.
Programmer: OM Partners
Tools used: *visualize_cumulative_resource* and *visualize_assignment*, combined with the search tree tool + *visualize_graph_lines* in the new version of the program.
Use of the information: the user found a bug in the old formulation of the cumulative resource constraint. When rewriting the program with a cycle constraint, the cycle visualizer helped in setting up the new model. The new program obtained a significantly lower cost for the first feasible solution.
Remarks: some functionality of the debugging tools was not completely clear (e.g. the Done button), more documentation would be interesting.
Shortcomings/extensions: It should be possible to give a user-defined label to the search nodes in the search tree tool and to the items in the visualizers. There is no uniform annotation between the different visualizers. The propagation information is (too) cryptic.

Application: Tank scheduling, taking into account non-simultaneous use of loading points and the production sequence. The sum of setup time and lateness cost has to be minimized. This new application will be taken in production in May 1999.
Programmer: OM Partners
Tools used: *visualize_cumulative_resource*, *visualize_assignment* and *visualize_graph_lines*, combined with the search tree tool.
Use of the information: the search tree tool encouraged the user to look for a better search strategy, and to extend the program with a min_max function. An error in the modeling of the successors in the cycle constraint has been found.
Remarks: there is no debugging help as long as the program does not work.
Shortcomings/extensions: The user would like to have more possibilities to show colors and user defined labels in the global visualizers.

2.2. Send more money

2.2.1. Problem description

One of the first problems considered is the well-known cryptarithmetic puzzle SEND+MORE=MONEY. The problem consists of assigning different digits to the different letters such that the equation holds.
Two alternative formulations of the problem have been tried:

- The first one uses one equation
- The second one uses carries (5 equations)

Constraints involved are alldifferent/1, simple disequations and equations. Labeling is performed using the built-in labeling routine: *labeling(L,0,most_constrained,indomain)*.

2.2.2. Evaluation

Tools used

Search tree tool (*search_labeling/4* as well as annotation by *search_start/2*, *search_number/2*, *search_node/3*). Tools version 15/10/98.

Use of the information

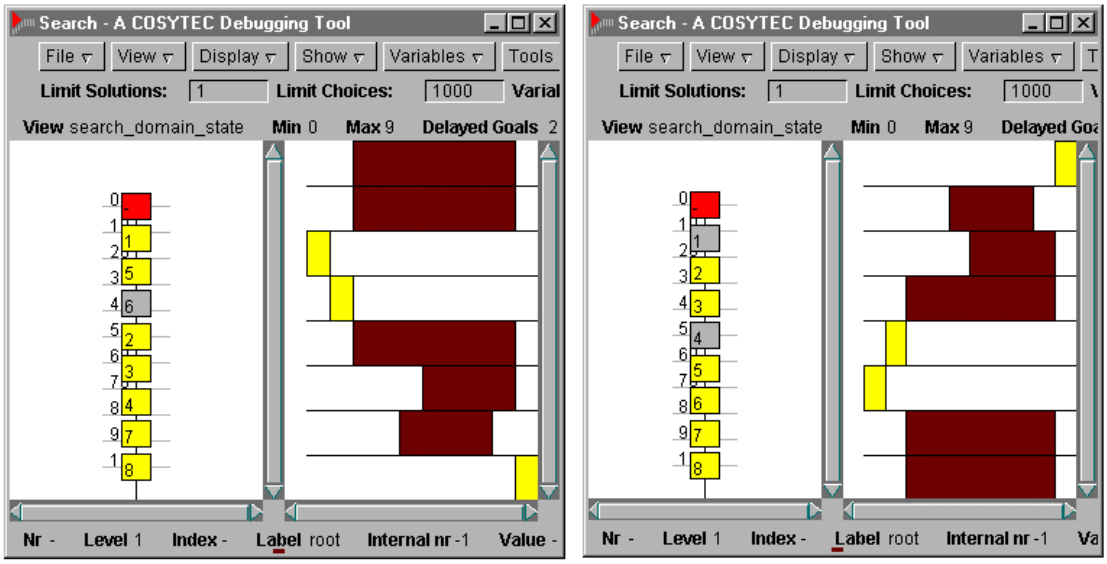


Figure 1 Domain state view; left: labeling order [S,E,N,D,M,O,R,Y]; right: [Y,R,O,M,D,N,E,S]

In the first formulation of the problem (with one equation) the labeling routine was replaced by the special *search_labeling/4* built-in. The search tree obtained contains a single branch, indicating that a solution was found without deep backtracking. The domain state view in the root node shows how initial propagation already instantiates 3 variables (S, M and O with search node indexes 1, 5 and 6; cf. figure on the left). The use of the *most_constrained* parameter in the labeling is reflected in the order of variables in the search tree nodes. It can also immediately be seen that a value choice is performed only once (for variable E with index 2; cfr. gray color in its predecessor node); in all other nodes, the domain of the variable is already reduced to a single value (yellow predecessor node). Whereas if variables are labeled in the order [Y,R,O,M,D,N,E,S], there are 3 nodes (for variables Y, R and D, now

with search node indexes 1, 2 and 5) in which a value choice still has to be performed (cfr figure on the right).

In the second formulation of the problem (with carries) we experimented with annotation via *search_start/2* and *search_node/3*. This allows to make a difference between the variables shown in the search tree and the variables being labeled. E.g.

search_start([S,E,N,D,M,O,R,Y,C1,C2,C3,C4], mylabeling([S,E,N,D,M,O,R,Y]))

only labels the real variables (carries are instantiated due to propagation) but shows information about all variables in the search tool[1]. Or, vice versa,

search_start([S,E,N,D,M,O,R,Y], mylabeling([S,E,N,D,M,O,R,Y,C1,C2,C3,C4]))

allows to concentrate on the real variables (not the carries) within the search tool although labeling is performed on all variables (in the given order).

Remarks

When using *search_start(L, ..)* and *search_node(X,N,indomain(X))*, the position of the variable *X* in the list *L* must correspond to the number *N* associated with *X* in *search_node*. Otherwise, the search view may be blank at non-root nodes. Note that the number *N* has nothing to do with the order in which the variables in *L* are enumerated. It only determines the order (number) of variables within the search tool views.

There may be some confusion about how the variable domain is indicated and how the coloring of nodes according to domain size (yellow for size 1 and gray for size strictly larger than 1) is done : the color/domain indication in a node is related to the *next* variable to be assigned (in the following node), not to the variable instantiated in the node itself. E.g.

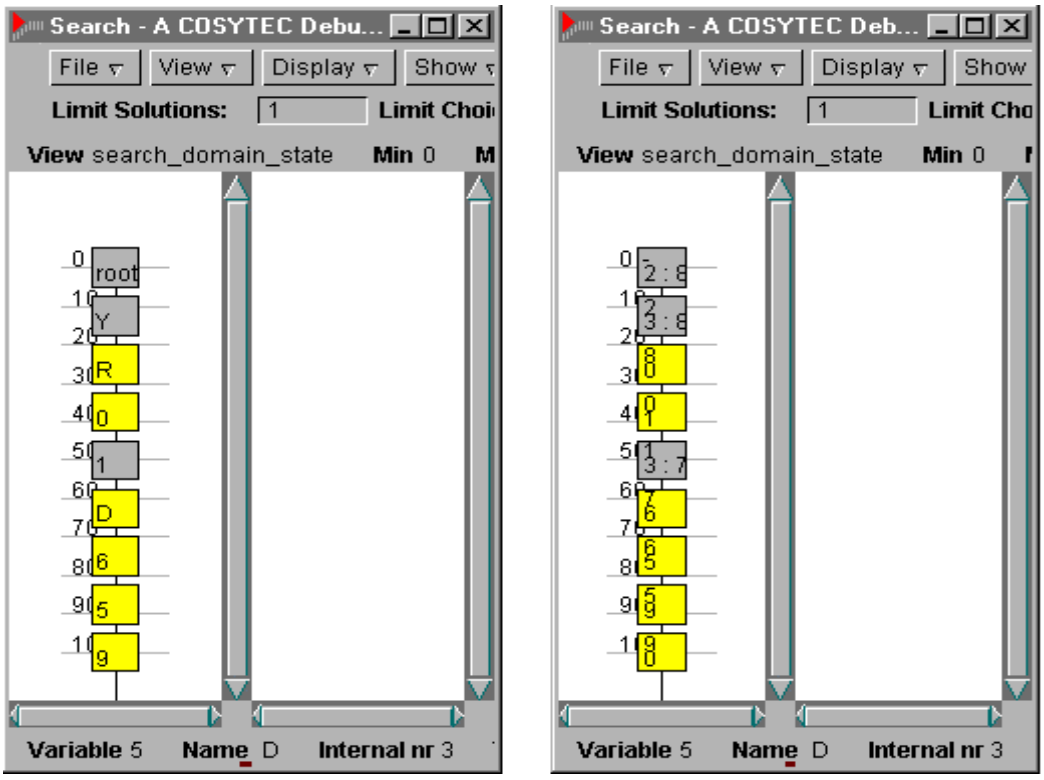


Figure 2 Search node annotation: left: index view - right: domain view

the gray color of nodes 1, 2 and 5 indicates that for the variables in the *following* nodes 2, 3 and 6 (variables Y, R and D) the size of the domain is larger than one (2..8, 3..8 and 3..7 respectively).

[1] Another way to get information about the carries (and still using search_labeling on the real variables only) is to use the visualizer *visualize_variable*.

Another peculiarity concerns the spy functionality. When spying a variable, a line is added in the spy window for each level in the search tree, except for the root level and the level just below it (cfr. figure). So, the spy window remains empty when selecting the root node or one of its subnodes.

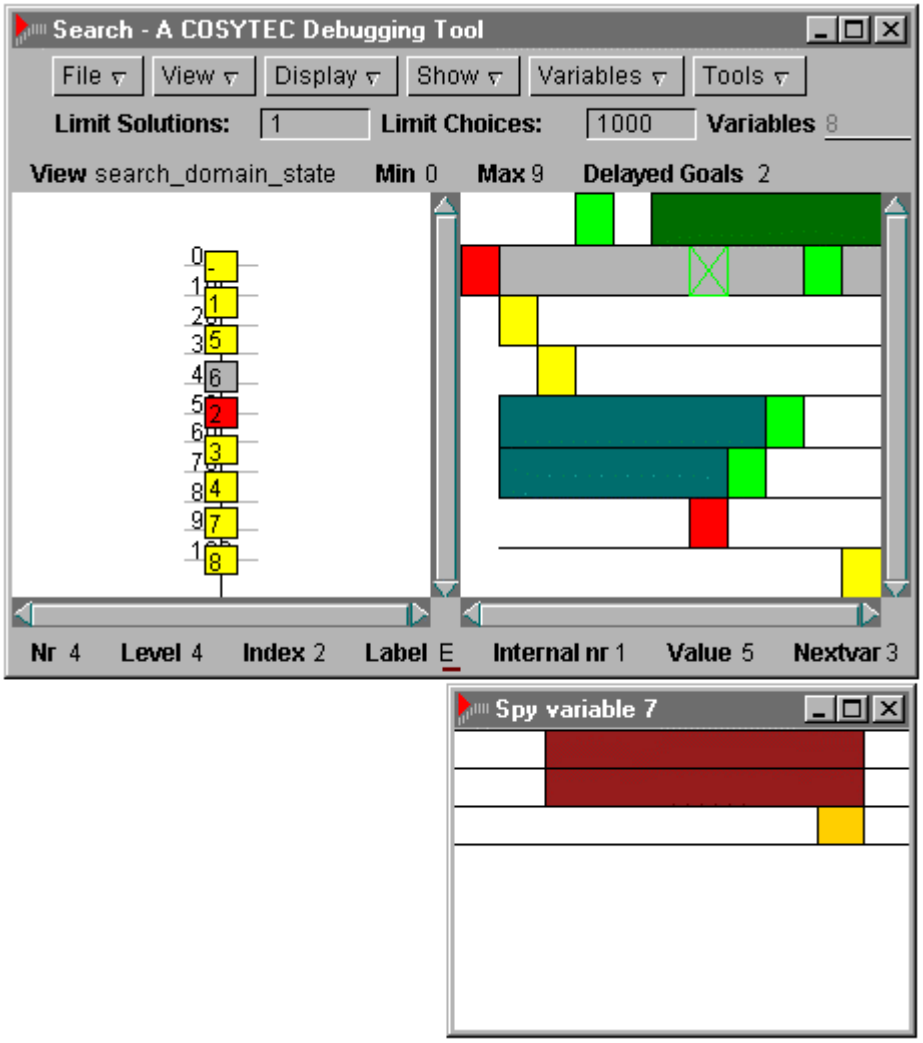


Figure 3 Spy variable 7 (current position is node 5 with index 2)

Shortcomings / possible extensions

The search tool gives no direct information about shallow backtracking, i.e. backtracking within a search tree node, although this information is also useful when comparing different labeling strategies and their propagation. A suggestion is to display the number of backtracks within the node.

2.3. Production sequencing

2.3.1. Problem description

This example is taken from the CHIP course. A company has to decide on the production cycle of different products on one machine. Transition from one product to another incurs a setup cost (non-symmetrical cost). Some product sequences are forbidden. The goal is to find an optimal production sequence, i.e. a sequence that minimizes the total setup cost.
The problem is solved using a cycle constraint:

```
top:-
    data(CostMatrix),
    length(CostMatrix, N),
    length(Ps, N),
    length(Ws, N),
    Ps :: 1..N,
    Ws :: 1..1000,
    Cost :: 0:1000,
    cycle(1, Ps, Ws, 0, 1000000,unused,unused,unused,unused,unused,[Cost,CostMatrix],unused),
    min_max(labeling(Ws,0,max_regret,indomain), Cost),
    writeln(Ps), writeln(Ws).
```

The *Ps* variables specify the successors of products in the production sequence. Labeling is performed on the weight variables (*Ws*), using the *max_regret* strategy.

2.3.2. Evaluation

Tools used

Search tree tool (*search_start, search_node*) combined with the *visualize_graph_lines* visualizer for the cycle constraint. Tools version 15/10/98.

Annotating the program

Labeling is only performed on the weight variables (*Ws*), not on the product successors (*Ps*). In order to show information about *Ws* as well as *Ps* in the search tree tool, the labeling routine is replaced by the following annotated version:

```
    append(Ws, Ps, Vars),
    search_start(Vars, min_max(mylabeling(Ws),Cost) ),
```

with

```
    mylabeling(Ws):-
      search_number(Ws, L),
      labeling(L, 1, max_regret, assign).
```

Use of the information

The search tree tool allows inspecting the propagation. However, it is not straightforward to see how the cycle is being built (interpretation of the instantiation of *Ps* is not that easy). Therefore the search tree tool is combined with the cycle visualizer. The visualizer provides a more intuitive picture of the cycle construction and leads to a better understanding of the problem.

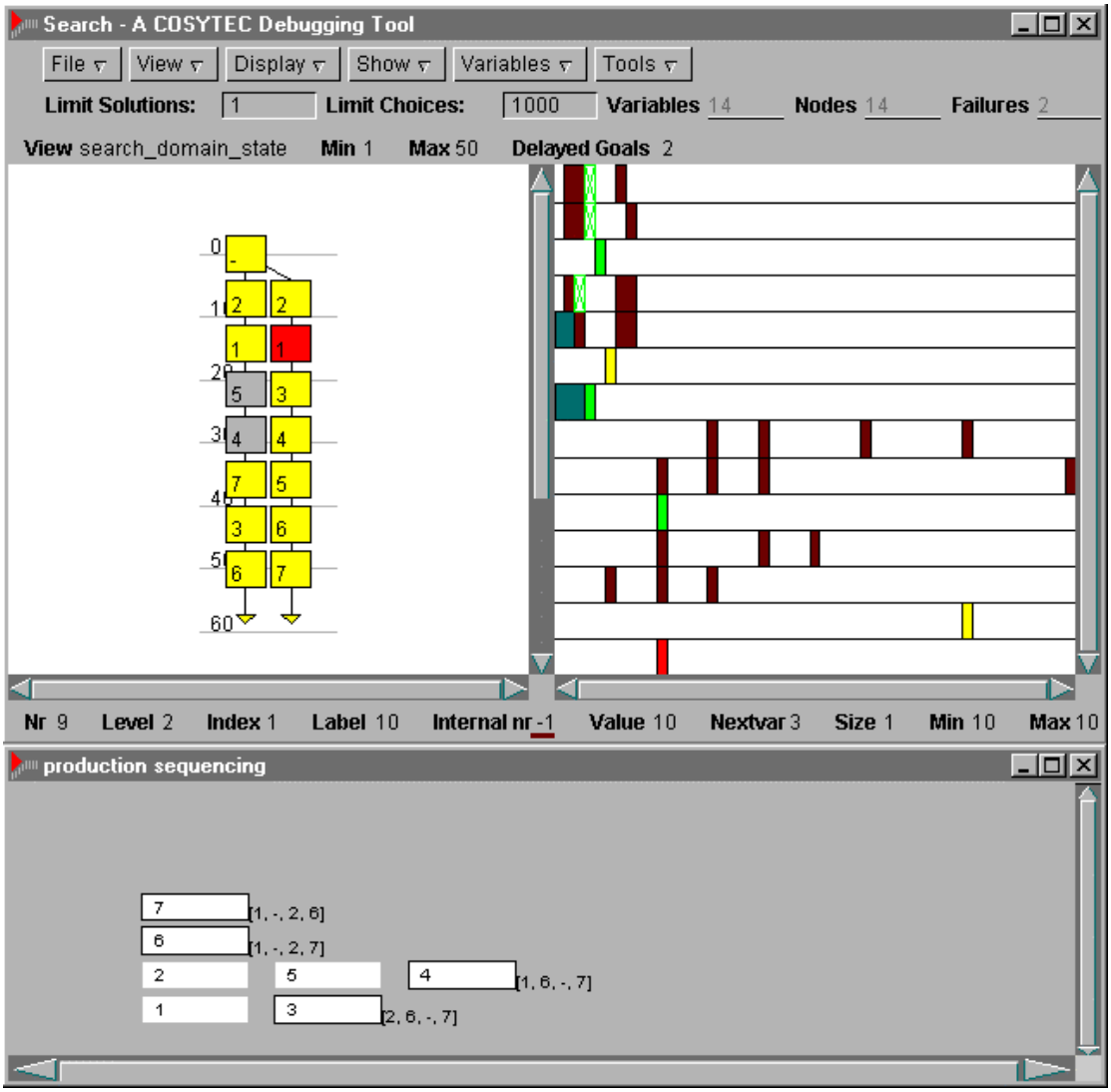


Figure 4 Visualization of cycle constraint

With the given data set, a first solution with cost 115 is obtained without deep backtracking. Afterwards, the optimal solution with cost 110 is found immediately (no value choice needed anymore). Note that during the search for this second solution the variables are labeled in another order (due to the extra constraint of having a cost smaller than 115).

2.4. Rectangle placement

2.4.1. Problem description

Given is a set of rectangles of sizes (n,n), (n-1,n+1), (n+1,n-1),…,(1,2n-1),(2n-1,1). The goal is to find the smallest square in which all the given rectangles can be placed such that there is no pairwise overlap. The rectangles cannot be turned (i.e. height and width cannot be interchanged).
The program uses a diffn constraint. The labeling is important to find a good solution in reasonable time. For n>6, it is very hard to find (and prove) the optimal solution (we stopped the search after taking 1 minute of CPU time). Some different labeling strategies were considered. Out of these, the strategy that performed best consists of first placing the smallest rectangles (rectangles with an "irregular" shape, i.e. a shape that is far from a square). Also, each new rectangle is put as close as possible to the bottom-left corner of the enclosing square, since the aim is to find the smallest enclosing square. This is obtained by labeling on X+Y, with (X,Y) being the coordinates of the bottom-left corner of the rectangle. In most cases, setting X+Y already fixes both X and Y. If this is not the case, labeling proceeds as follows: for a "horizontal" rectangle (width>height), X is labeled (smallest value first); for

a “vertical” rectangle (width<height), Y is labeled (smallest value first). The labeling part consists of the following code:

```
…,
sort(TermList,SortedList,5),
min_max(mylabeling(SortedList), Cost),
…

mylabeling([]).
mylabeling([R|Rs]):-
  myindom(R),
  mylabeling(Rs).
myindom(r(X,Y,_,_,_,PXY)):-
   indomain(PXY),
   ( SX > SY →  indomain(X) ; indomain(Y) ).
```

where TermList is a list of terms *r(X,Y,SX,SY,Surface,PXY)*, each identifying a rectangle: (X,Y) is the bottom-left corner, SX and SY are resp. the width and height of the rectangle, Surface is SX*SY and PXY is X+Y (note: X,Y and PXY are domain variables, SX,SY and Surface are numbers).

2.4.2. Evaluation

Tools used

(1) *Visualize_placement_2d* and *visualize_placement_remains*.
(2) Combination of (1) with search tree tool
Tools version 15/10/98.

Annotating the program (1)

This was a first experiment to use the visualize tool on its own (not in combination with the search tree tool). Annotating the program is straightforward: it suffices to put the *visualize/4* wrapper around the diffn constraint and add *visualize_update* at the end of the *myindom/1* labeling routine. Putting the update at that point allows to follow the rectangle placement step by step, at least if some form of delay is added as well (via *sleep/1* on CHIP for UNIX or via *system(pause)* on NT).

Use of the information

In optimization problems with no or few initial constraints it is important to find a good first solution. Here the visualizer definitely helps to understand and explore different labeling strategies.
After a first solution has been found, the search continues with an upper bound on the cost (i.e. on the size of the enclosing square). The extra constraint is reflected by the appearance of obligatory parts.

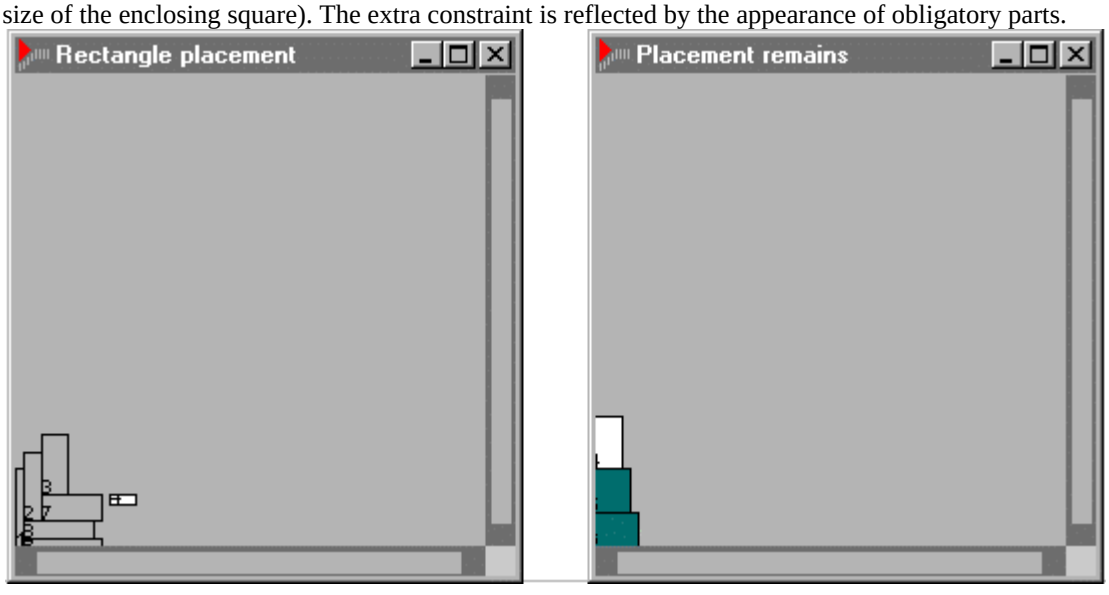


Figure 5 Diffn visualizers (obligatory parts are shown in white)

Bugs found

In a first version of the program, the built-in *labeling/4* was used for determining the term selection (i.e. which rectangle to put first):

min_max(labeling(TermList,5,smallest,myindom), Cost)

However, the visualizer immediately showed that rectangles were still placed in the order of appearance in TermList, so the selection method had no effect. This was due to an incorrect interpretation of the *labeling/4* predicate: the selection method only works if the argument on which the selection is based is a domain variable, not if it is a fixed number.

Annotating the program (2)

The easiest way to debug or examine the program is through the dedicated placement visualizers. However, to obtain more detailed information, the visualizers can be used in combination with the search tree tool. In this case, only the *visualize/4* wrapper around the diffn is needed; the visualizers are updated automatically when a new node in the search tree is selected.

Using the search tool requires some changes to the program, more precisely to the creation of the TermList and labeling thereupon. A term is extended to *r(X,Y,SX,SY,Surface,PXY,Nr)* where *Nr* is the number of the rectangle. The labeling part is changed to:

```
…,
sort(TermList,SortedList,5),
generate_XandYlist(SortedList, XandYlist),
search_start(XandYList, min_max(mylabeling(SortedList), Cost),
…

mylabeling([]).
mylabeling([R|Rs]):-
  myindom(R),
  mylabeling(Rs).
myindom(r(X,Y,_,_,_,PXY,Nr)):-
   indomain(PXY),
   XNr is Nr*2-1,
   YNr is Nr*2,
   ( SX > SY →
            search_node(X,XNr, indomain(X)), search_node(Y,YNr, once(indomain(Y)))
    ;
            search_node(Y,YNr, indomain(Y)), search_node(X,XNr, once(indomain(X)))
    ).
```

A list with the X and Y of each rectangle (*[X1,Y1,X2,Y2,…]*) is constructed (*generate_XandYlist*) and passed as first argument of *search_start*.

Remarks

In order to run the visualize tool it was necessary to load not only the *visualizers* but also the *search* library. Loading the latter opens a window which seems to be needed in order to open other visualizer windows. This problem was due to a bug in the visualize library[2].

When using the visualizer on it's own, labeling on *PXY* and on either *X* or *Y* is sufficient. Whereas in combination with the search tool, both *X* and *Y* have to be enumerated (have to appear within a *search_node*).

Shortcomings / possible extensions

In order to monitor the labeling (rectangle placement) step by step, it should be possible to insert a breakpoint to delay the execution (some kind of *visualize_break* predicate)[3]. The system should then wait for user input before continuing or stopping the execution. Moreover, it would be nice if the user could ask for additional information, such as the domain of a variable, etc. Currently, the user has to

[2] Error 1045 : no connection to display in window_query(window_current, Old_current)

[3] A workaround is to insert the call *system(pause)*, just after visualize_update. Nevertheless, it might be useful to have a more sophisticated kind of *visualize_break*.

determine in advance what has to be displayed; there is no way to gather additional information once the execution has started.

When using the visualize tool on its own, the created windows are not "active": they cannot be moved/resized, no scrolling within the window is possible. This is identified as a restriction of the native NT version of the visualization tools[4].

The visualizers have a different (more interesting) behavior when used on their own, compared to when used in combination with the search tree tool. In the latter case:

- The visualizers do not show the obligatory parts during the search for the optimal solution, i.e. after a first solution has been found.
- The domains shown in the search nodes indicate that after finding a new (better) solution the problem is further constrained (domains are more restricted). However, this domain restriction is not reflected in the domain state view nor in the visualizers for the variables.

The reason seems to be the following: the extra constraint introduced by the *min_max* predicate each time a better solution is found, is taken into account when running the visualizer on its own, whereas it is not when running the visualizer in combination with the search tree tool.[5]

2.5. Small scheduling demo

2.5.1. Problem description

This small scheduling problem consists of 8 tasks, each with a given duration and resource usage. The available amount of resources is also given. The aim is to schedule all tasks such that the overall time for finishing all tasks is minimized.
A cumulative constraint is used to express that the global resource usage at each timepoint should be less than the given limit. Labeling is performed through the built-in *labeling/4*, trying the strategies *smallest* and *most_constrained*.

```
top:-
    durations(Ds),
    resourceusage(Rs),
    length(Ds, N),
    sum(Ds, 0, MaxHorizon),
    length(Ss, N),              /* Ss: starttimes of the tasks */
```

[4] Explanation of the implementers of the tools (H. Simonis): "This is actually a restriction of the native NT version. If the display loop is not running, the resize/move events are queued and not processed. In the Exceed version this is handled differently, as the window manager of Exceed handles these events on the top-level windows. But even there the callbacks to draw the contents of the window are only executed when the display_loop is running. There is no possibility to change this with our current console based application, this would require a multi-threaded full Windows application.

[5] The same code of the constraint visualizers is run inside and outside the searchtree tool. There are no functional differences between the two versions, the search tree tool just call visualize_update after having instantiated the selected node.
The visual difference comes from a more fundamental limitation of the current search tree tool. If during the search new constraints are added in the search procedure, they are not taken into account when later on a state of the search is re-instantiated. The display of the searchtree tool will show the link, where new constraints have been added, in red. This is a warning that the results in the search tree tool do not completely match the real execution.
A typical instance of this problem is the min_max optimization. Whenever a new solution is found, the updated cost is imposed as a new constraint. The global constraint visualizer when run on a node beyond the first solution will not take this new constraint into account and therefore may not show the same behaviour, obligatory parts may be missing, etc.
This limitation is fundamental to the current version of the tool. A solution would require siginificant new functionality in the constraint engine. A fix consists in imposing by hand the correct upper bound on the cost variable in the program to visualize the behaviour of one branch of the min_max search.
Note that the tree view is not affected by this problem, but that all constraint and varible views can show this problem.

ResourceLimit :: 0..4,
Ss :: 0..MaxHorizon,
End :: 0..MaxHorizon,
cumulative(Ss, Ds, Rs, unused, unused, ResourceLimit, End, unused),
min_max(labeling(Ss,0,most_constrained,indomain), End),

2.5.2. Evaluation

Tools used

Visualize_cumulative_resource visualizer and *visualize_variable* to show the starttime variables *Ss* of each task (+ search tree tool). *Visualize_placement_2d* for the diffn to assign tasks (cfr. below). Tools version 15/10/98.

Annotating the program

Annotating the program is simple. It consists of adding the *visualize/4* wrappers, adding the wrapper *search_start/2* around min_max and replacing the labeling routine with mylabeling which calls *search_number/2* before the actual labeling/4.

Use of the information

The cumulative visualizer shows that, in the optimal solution, the resource is used up to its limit during the entire period in which the tasks are running. So it is impossible to add another task using the given resource without increasing the global end time.

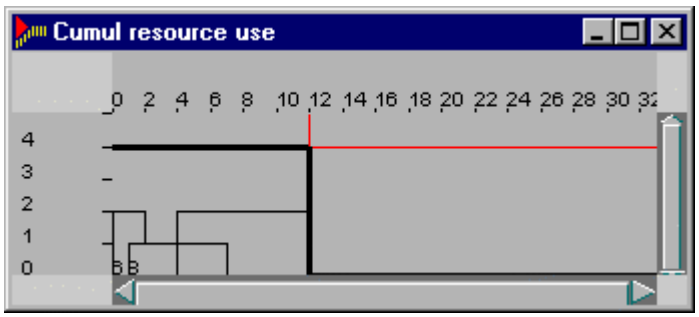


Figure 6 Cumulative resource usage in optimal solution

No obligatory or expected profile appears during the search for the first solution, as the time horizon maximum is set to the sum of the durations of all tasks. However, if the *MaxHorizon* limit is restricted, an obligatory and expected profile appear. One can even find out which task is responsible for the obligatory part by clicking on the part in the cumulative visualizer. This highlights the corresponding line (i.e. task) in the starttimes visualizer.

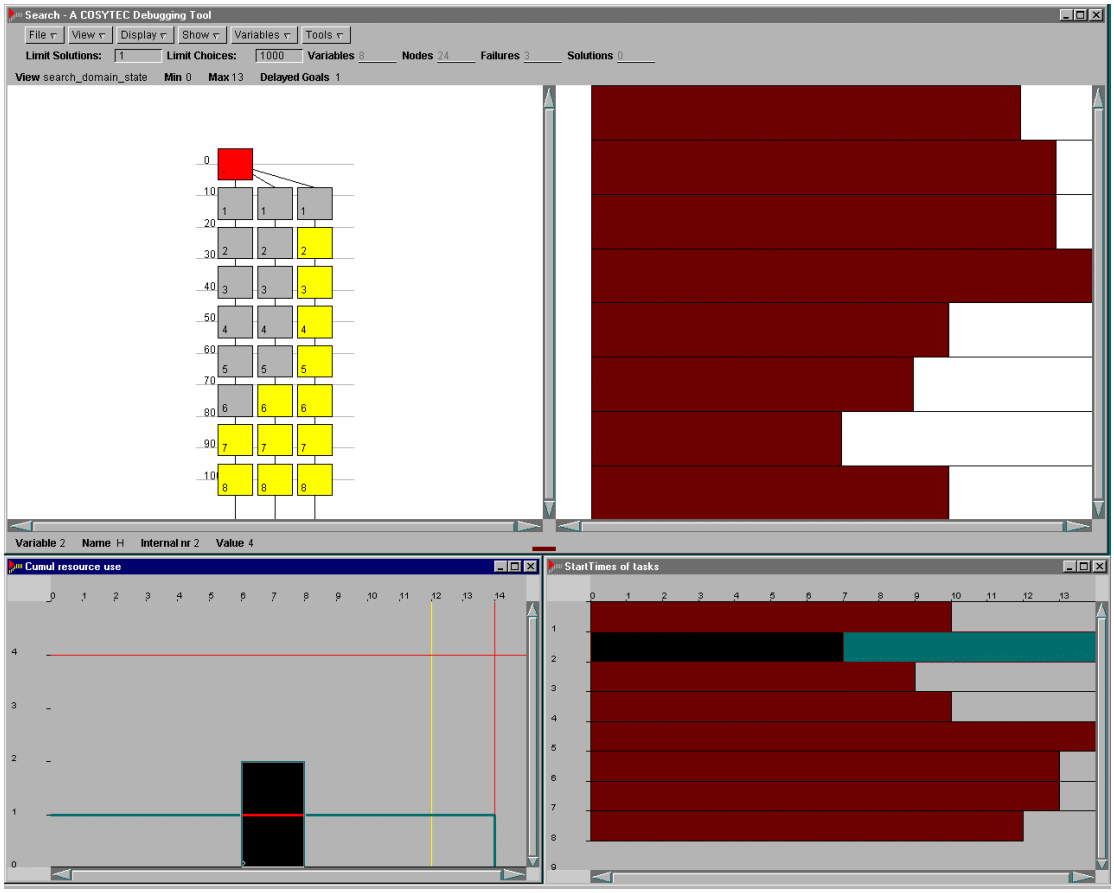


Figure 7 Cumulative resource and variable visualizers

After determining the global end time and the starttimes of each task, a diffn can be used to assign tasks to specific resources and to obtain a more specific view on which resources are used by which tasks. However, note that representing each task by a rectangle [StartTime, Resource, Duration, ResourceUsage] in a diffn imposes additional constraints:

- A task is assigned to a resource during the *entire* duration of the task.
- A task is assigned to *consecutive* resource elements (the variable Resource represents the number of the first resource assigned to the task).

These constraints can be relaxed by splitting a task into small rectangles with height 1 and duration 1. For a task of duration *d* and resource usage *r*, *d*r* rectangles have to be created.

In our small example, we just used the simple diffn (one rectangle per task) but increased the resource limit. So the program is extended with:

```
ResourceLimitExt is ResourceLimit*2,
makerect(Ss, Ds, Rs, ResourceLimitExt, Resources, Rectangles),
labeling(Resources,0,most_constrained,indomain)
```

with

```
makerect([], [], [], _, [], []).
makerect([S|Ss], [D|Ds], [R|Rs], RLimit, [Y|Ys], [[S,Y,D,R] | Rects]):-
  YLimit is RLimit - R,
  Y :: 0..YLimit,
  makerect(Ss, Ds, Rs, RLimit, Ys, Rects).
```

(note: this part is performed after fixing the overall end time and the starttimes of each task with the cumulative)

The visualizer shows that, with the additional constraints imposed by the diffn formulation, a larger resource limit is needed. However, splitting one of the tasks into 2 subtasks and assigning each subtask to another resource allows to obtain the original resource limit.

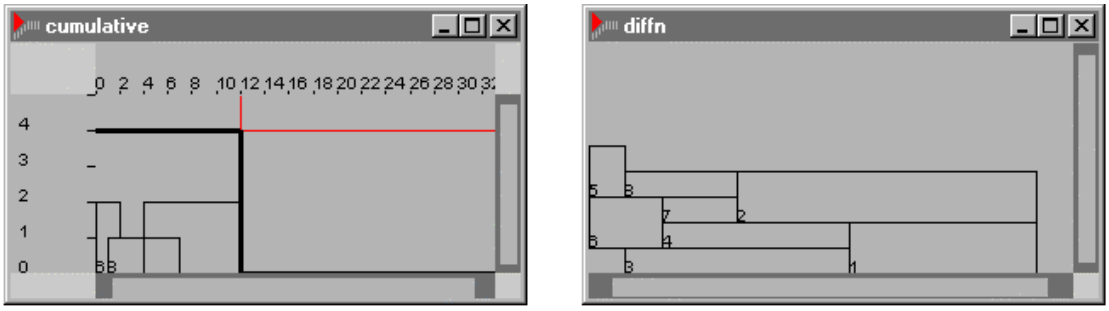


Figure 8 Visualization after finding the optimum solution

Shortcomings / possible extensions

The diffn visualizer *visualize_placement_2d* does not have a legend on the X and Y axis, so the starttimes and resource elements assigned to tasks cannot easily be derived. This legend does appear when using the *visualize_assignment* visualizer (note: this visualizer works although the documentation states that the height of each rectangle should be 1, which is not the case in this example).

Using the *smallest* labeling strategy, the search tree consists of three branches, each representing a solution (with cost resp. 14, 13 and 12). As in the rectangle placement problem, the visualizer (for the cumulative) provides more information when used on its own, compared to when used in combination with the search tree tool. In the case of combined use:

- The cumulative visualizer does not show the obligatory parts nor the expected profile after the first solution has been found.
- Also, the line indicating the upperbound on the end time is not adjusted after the first solution has been found.
- The restriction of variable domains each time a new solution is found is not shown in the variable visualizers nor in the domain state view (it can only be seen from the indication of the domains in the search nodes).

The reason is that constraints introduced during the search (e.g. the constraint added by min_max each time the search for another solution is started) are not remembered by the search tree tool.

When the *MaxHorizon* limit or the *ResourceLimit* is set too small (such that no solution can be found), the cumulative visualizer gives an error and the execution is aborted[6].

2.6. Real-life scheduling application involving setup cost

2.6.1. Problem description

The problem consists of scheduling a number of tasks (about 100 tasks) on 8 machines. Machines are divided in two classes. Each task has to be scheduled on a machine in a given class, depending on the task type. Moreover, the machines allowed for a task are ordered by preference. Some tasks are already fixed on a particular machine. Most of the tasks use a common resource (laboratory personnel). The number of resource instances is limited. Further, each task has a different Bill of Material and the

[6] *visualize(cumulative([H in {0..11}, H in {0..11}, H in {0..11}, H in {0..11}, H in {0..11}, H in {0..11}, H in {0..11}, H in {0..11}], [5, 8, 6, 5, 1, 2, 2, 3], [2, 2, 1, 1, 2, 2, 1, 1], unused, unused, RLimit in {0..4}, End in {0..11}, unused), visualize_cumulative_resource, [winx <- 10, winy<- 700], Cumul resource use)*
call_failed(cumulative(…))
Error 121 : domain variable or natural number expected in domain_info(Ovar_91112, Y1_379, Y2_379, __382, __383, __384) "c:\development\chip5/PLLIB/visualize_profile.pl", line 98: vs_profile_limits(Cumul resource use)

availability of raw material is limited. The schedule also takes into account the unavailability of machines and resources.
The end products produced by the tasks belong to different product families. Transitions between different product families in the schedule involve a setup cost.
For each task, an earliest and latest production time is specified. The earliest time is a hard constraint, the latest time can be relaxed. The scheduling horizon is 0..50000 minutes (about 34 days).
The goal is to find a schedule that minimizes the overall setup cost while at the same time keeping the lateness of tasks under control, also taking into account the preference of machines for each task. A parameter allows weighting the importance of setup cost against the lateness of tasks.
As the search space is quite large, the problem is relaxed to finding a good quality solution (suboptimal) reasonably fast (i.e. within some seconds).

The program already existed before the start of the DiSCiPl project. It includes several diffn and cumulative constraints (besides equations, inequalities and element constraints). A sophisticated labeling strategy is used to find a reasonably good solution. At the time the program was written, the cycle constraint was not yet available. Therefore, reducing setup costs was integrated in the labeling strategy.
In the context of DiSCiPl, the program was rewritten using the cycle constraint. This constraint allows modeling the sequence of tasks on each machine as a cycle, in which the setup costs act as weights.

2.6.2. Evaluation

Tools used

Visualize_cumulative_resource and *visualize_assignment* visualizers, combined with the search tree tool showing the end time and machine variable for each task. Tools version 15/10/98.
In the new version of the program, *visualize_graph_lines* was added to visualize the cycle constraint.

Annotating the program

Adding the necessary annotations for the search tree tool and visualizers required only minor modifications to the program.
In order to use the visualizers, *visualize/4* wrappers are simply put around the *cumulative*, *diffn* and *cycle* constraints. However, some care is needed when the same routine setting up a constraint is called several times (e.g. the same routine may be used to set up a *cumulative* constraint for each resource). In that case, a unique identifier has to be created for each visualizer (based on the information passed to the routine).
Using the search tree tool requires to create the list of search variables and to store some bookkeeping information (in order to obtain the number of the search variable during the labeling phase). Adding the latter information is very easy if the program uses CHIP objects. Each task is modeled as an object instance, containing all information about this task. The number of each task for use in the search tree tool can just be added as an extra data field in the object instance.

Use of the information

Before labeling is started (i.e. just after setup of the constraints), the visualizers already show the position and resource use of fixed tasks (tasks that cannot be moved and dummy tasks used to model unavailability of machines/resources). This allows verifying the correct modeling of fixed parts.

Afterwards, one can check how labeling proceeds. For example, the debugging tools allow checking if tasks are considered in the correct order, e.g. if ordering is based on minimal end time. However, when ordering is based on more complex criteria (e.g. setup costs between task types), the tools do not provide much help. One still has to resort to more complex (external) visualization tools (e.g. a Gantt representation where tasks can be colored according to task type).

By default, the displayed area in the visualizers is limited by the upper domain limits of the associated variables. This yields a general overview of the problem. Moreover, it is possible to zoom in on a specific area in order to obtain more details.

The search tree tool confirms that, in the original program version, the implemented labeling strategy yields a reasonably good solution without backtracking (setup cost: 44, cost of tasks being too late:

3216 => overall cost of 3656). However, the search space is too large to get to the optimal solution. Use of *min_max* with the time-out parameter to limit the search resulted in a (sub-optimal) solution with cost 3390 (setup cost: 26, cost of tasks being too late: 3130), for the given dataset.

As mentioned above, an alternative program was developed that uses a cycle constraint. The cycle visualizer helped in setting up this new model (e.g. helped in identifying the lack of propagation due to errors in the modeling and showed the need to use the origin parameter of the cycle). The labeling part could be simplified by exploiting the cycle: direct labeling of the weights (representing setup costs between tasks) could be used in order to limit setup costs. The remaining labeling code focuses on restricting lateness costs. For the example dataset, the new version of the program results in a first solution with cost 3494 (setup cost: 18, cost of tasks being too late: 3314). This is significantly better than the one obtained with the original program as far as setup costs are concerned (the cycle leads to a better restriction of the search space).

Bugs found

The original program contained an error in the construction of fixed (dummy) tasks to model the unavailable periods of resources, causing no such tasks to be created. The visualizers immediately showed the absence of these tasks.

Remarks

The *diffn* and *cumulative* visualizers do not show an expected profile. It is not clear why this is missing.

With the search tree tool, code following *search_start/2* is not executed until the menu item *Done* (in *File* menu) is chosen.

In the search tree tool, the end time and machine variables of the tasks were specified as search nodes. However, the size of the end time domains is much larger than that of the machine variables. So the domain state view does not give very clear information about the latter. The information can better be obtained through a specific visualizer for the machine variables.

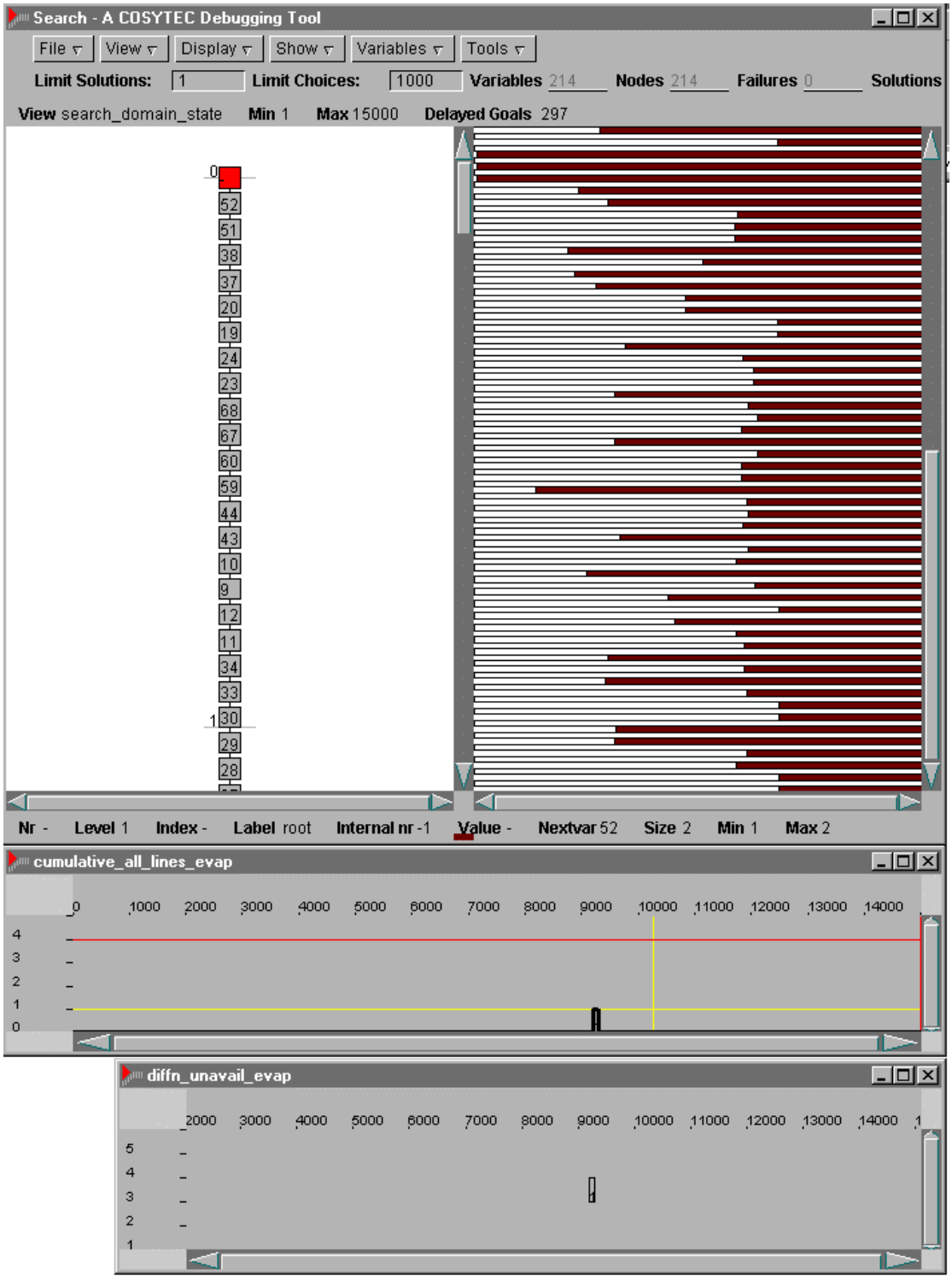


Figure 9 Snapshot before labeling (one fixed task is shown) – no min_max

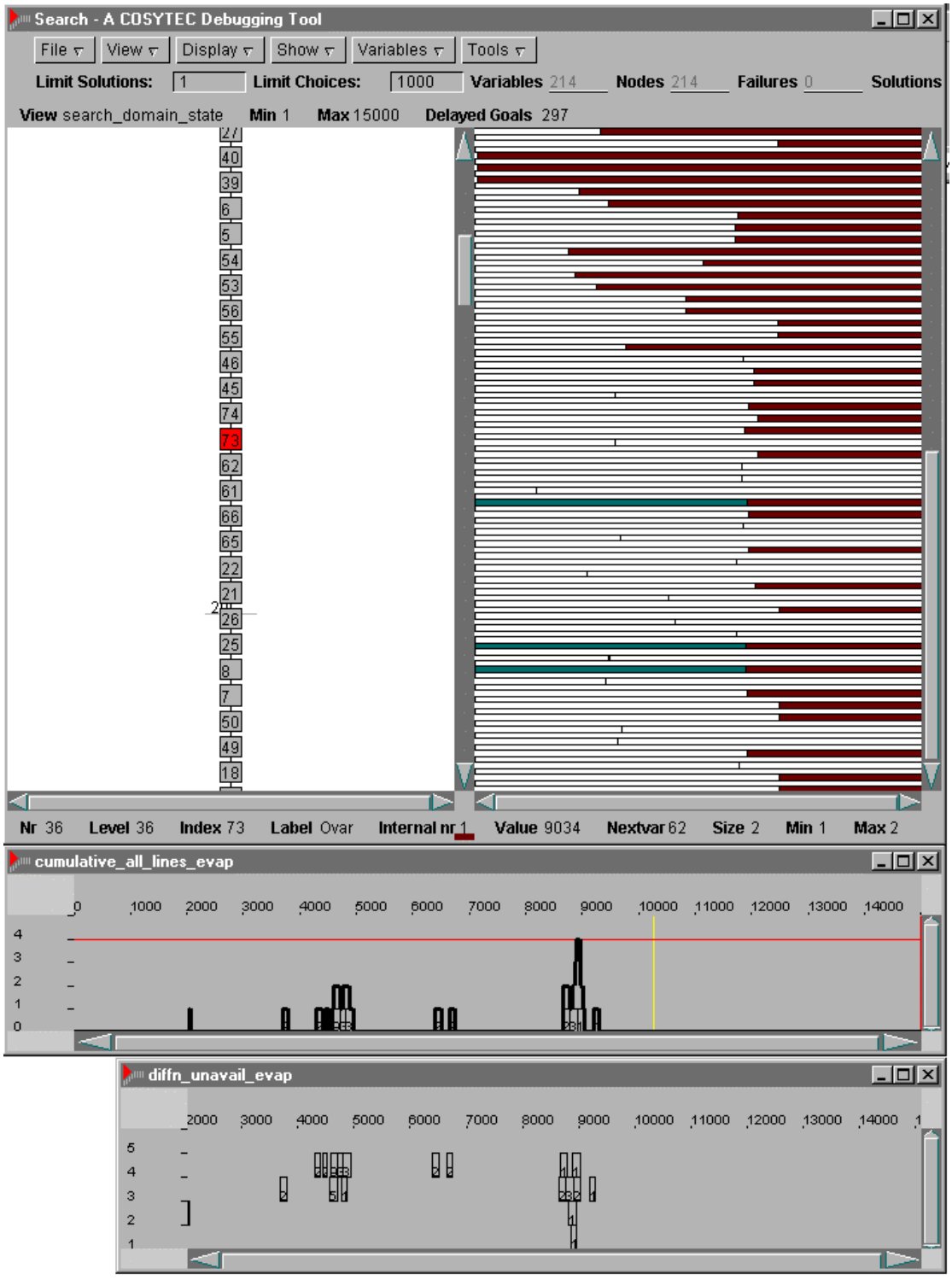


Figure 10 Snapshot during labeling – no min_max (1st solution only)

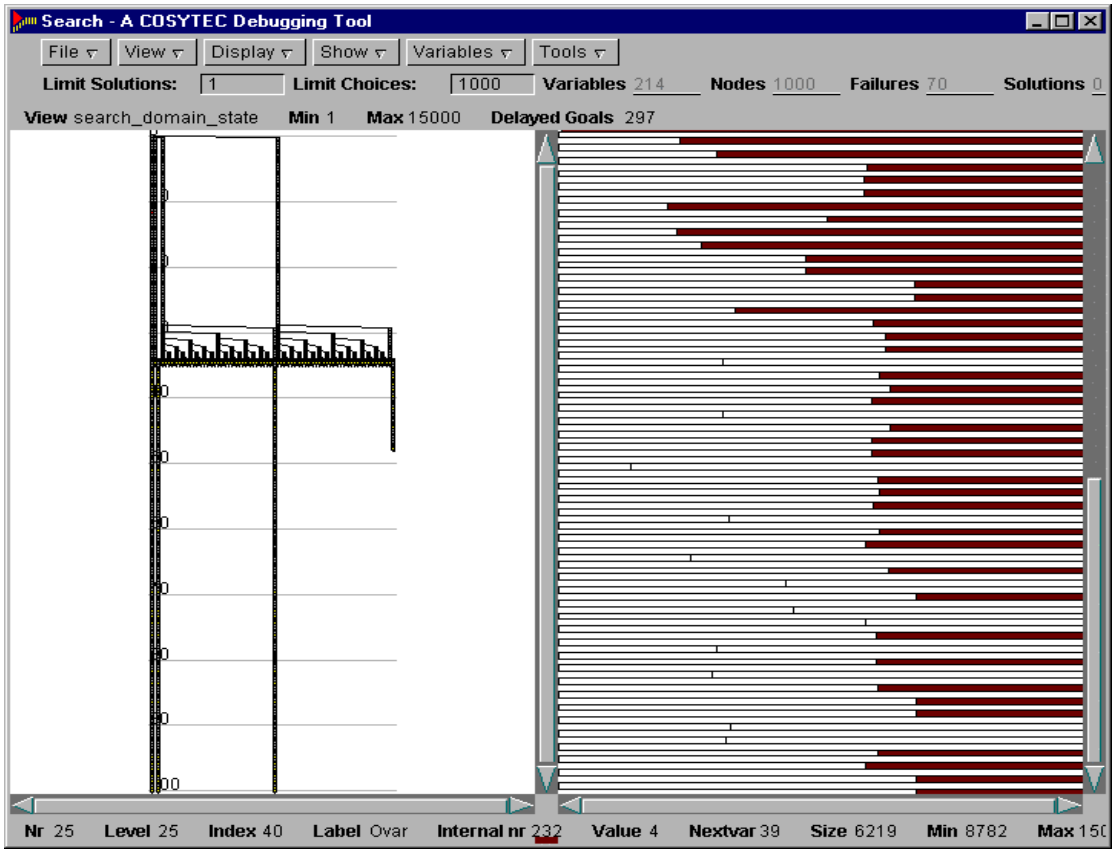


Figure 11 Original program with min_max

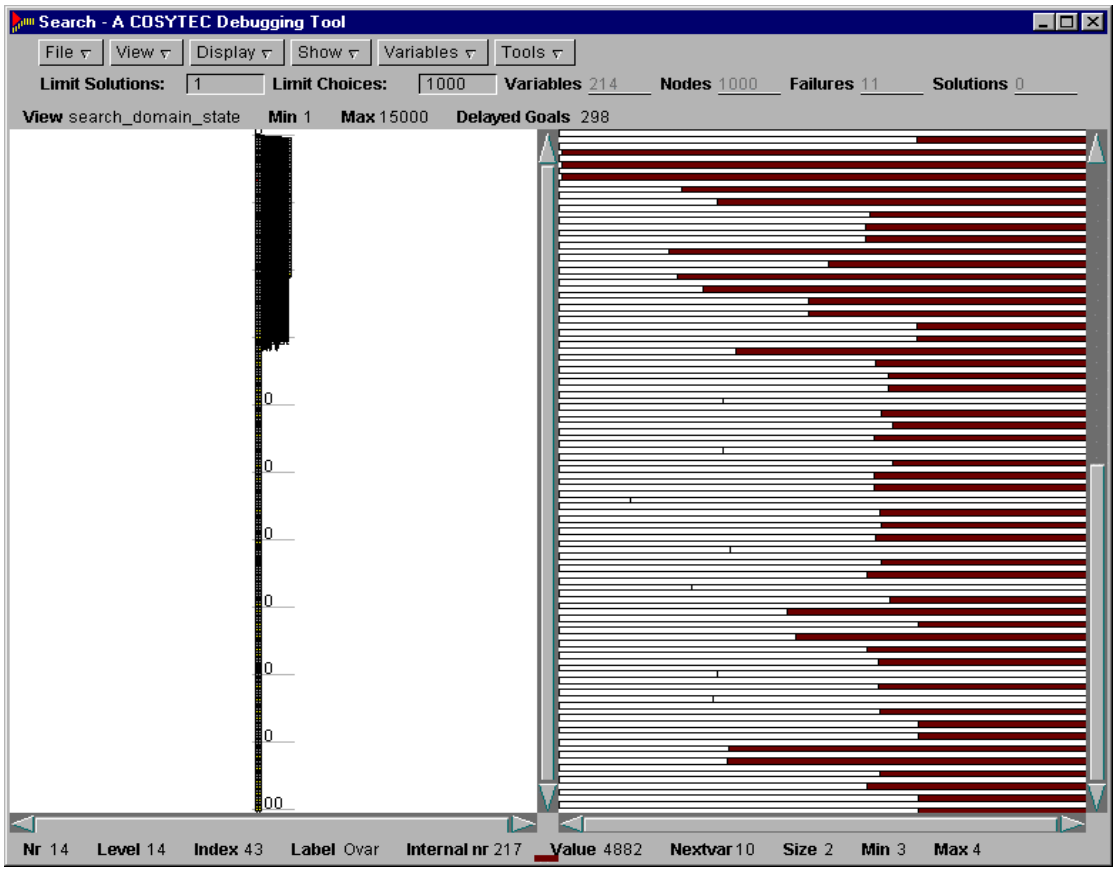


Figure 12 New program with cycle and min_max

Shortcomings / possible extensions

The original program specified a time horizon of 0..50000 (minutes). This size turned out to be too large for the search tree tool and visualizers[7]. Restricting the time horizon to 0..15000 (which is still large enough in order to schedule all given tasks in the example dataset) solved this problem.

When adjusting the scale of a visualizer (zooming in on some subarea), the original legend is just torn out (the number of indications remains the same; cfr. figure below). A suggestion is to refine the legend when zooming in (increasing the number of indications).

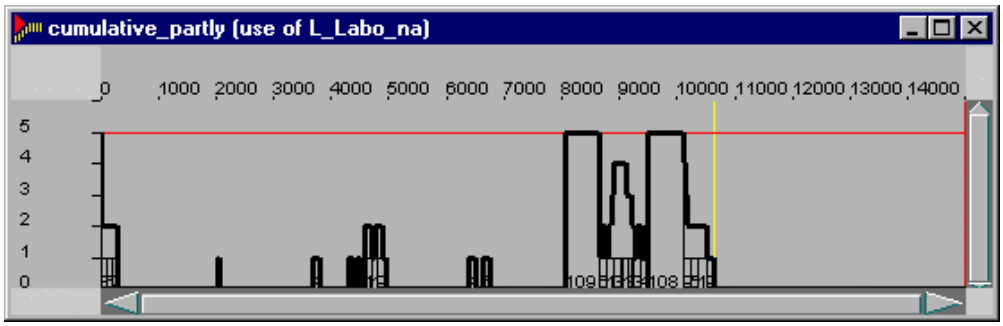


Figure 13 Default display

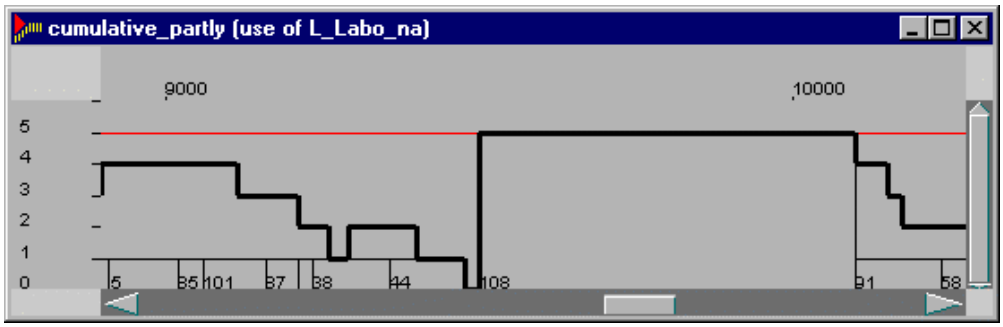


Figure 14 Display after zooming in

When dealing with large problems (large data sets), it is not straightforward to interpret what is shown in the search tree tool. The annotation of search nodes gives no clear indication to which variable (and, in this example, which task) a node corresponds. Maybe a variant of the *search_node* predicate could be provided, in which the user can specify a label for the node, e.g. *search_node(X,N,'task3',indomain(X))*. An extra option "Show User Label" in the search tool could then annotate the search node with this label. In this way the user could annotate search nodes with any information he finds useful.
Also in the visualizers there's a lack of useful annotation. E.g. in this particular scheduling problem, it would be interesting to see the task type, in order to find out if tasks of the same type are grouped such that setup costs are minimized. Maybe the *visualize* wrapper could be extended with an extra argument, namely a list of labels corresponding to the list of rectangles in the diffn, or to the lists of variables (starts, durations,ends,…) in the cumulative (it's not clear what to do with the cycle constraint). Even better would be if the different rectangles in the diffn visualizer could be colored based on some characteristic of the rectangle.

There is no uniform notation (numbering) of tasks over the different visualizers; the number of a task in a visualizer depends on the occurrence of its associated variable(s) in that particular constraint. Hence,

[7] Execution of the tools was aborted with the following error message: "Error 1009 : not a valid VDC in window_vdc(0, 0, 50001, 214); "c:\development\chip5/PLLIB/search_vdc.pl", line 32: search_domain_vdc_set(1, 50001, 214, 297)" for the search tree tool, and "Error 1009 : not a valid VDC in window_vdc(0, 0, 50001, 5); "c:\development\chip5/PLLIB/visualize.pl", line 384: visualize_create_ltsrsb(cumulative_all_lines_load)"

clicking on some item (rectangle) in a particular visualizer highlights items with corresponding numbers in other visualizers, but these do not necessarily correspond with the same task.

The propagation events view in the search tool may be very useful for the implementers of CHIP but is too cryptic for an end-user of the system.

2.7. Tank scheduling application

2.7.1. Problem description

The problem described in this section is a tank planning + shipment problem. A task consists of the following steps: creation of a certain product quantity in a reactor, storage of the product within a tank, and loading (shipment) of the product out of the tank using some loading machine. As soon as the reactor step is finished, the product has to be put into some tank (fill action). This tank is chosen among a pool of available tanks. Of course, tanks have a limited size and a tank should not contain different products at the same time. At some point, (part of) the tank content has to be loaded (empty action). The choice of loading machine depends on the tank. Loading is only possible during working hours.

The problem has been solved in different phases:
The first goal was to schedule the fill, empty and loading actions, keeping the reactor steps fixed and taking into account all tank and lateness constraints. The program involves several cumulative and diffn constraints (on tanks and loaders). Alternative formulations are possible to take into account the tank capacity:

- a diffn in 3 dimensions (each item in the diffn models the storage of a product in a tank, lasting from fill to empty, after the product has come out of a reactor step), or
- a cycle implicitly involving a cumulative (cfr. arg 12 of the cycle; this cycle models the sequencing of fill and empty actions on tanks).

The second goal was to take also the reactor planning into account. On the reactor there are major setup times between different product families. As usual, it is difficult to determine the relative importance of setup time versus lateness.

2.7.2. Evaluation (Part I)

In this part a solver has been made for the tank assignment. The reactor steps are fixed.

Tools used

Visualize_cumulative_resource, *visualize_assignment* and *visualize_graph_lines*. Also combination with the search tree tool showing the start time, end time and machine variable for each task. Tools version 15/10/98.

Use of the information

The cycle visualizer shows how fill and empty steps on the tanks are scheduled. E.g. in the figure below (note: a fill and its corresponding empty step have subsequent numbers), the sequence on tank 8 (cfr. upper line) is $fill_1(22)$-$empty_1(21)$-$fill_2(28)$-$fill_3(32)$-$empty_3(31)$-$empty_2(27)$; the tank is empty after step $empty_1(21)$ and after $empty_2(27)$. Unfortunately, the step numbers shown in the cycle do not reveal which product is involved. So, it's difficult to check whether a tank never contains a product mix. In fact, the program initially contained a bug with respect to this constraint. This bug was only detected through an external visualization tool (part of our own application) where coloring of tank steps on product type is possible.

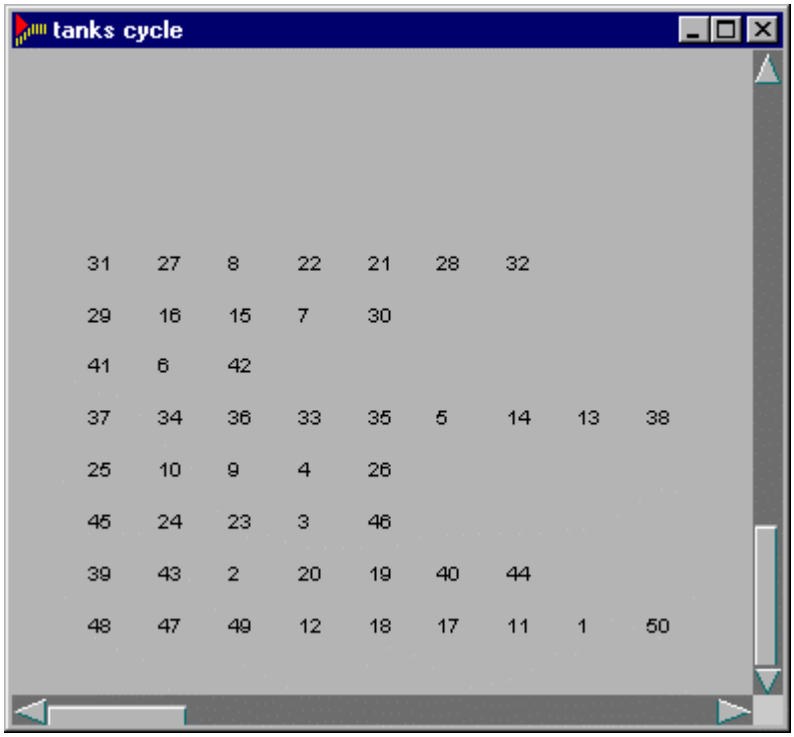


Figure 15 Cycle visualizer on tanks

The cycle visualizer only provides information about the relative sequence of fill/empty steps. It does not indicate their absolute position in time. The latter can be derived from the diffn visualizer on tanks. However, a problem is that step numbers do not correspond in the different visualizers (cfr. below).

A cumulative visualizer shows the workload of the loaders. It can be derived at what times the maximum capacity is reached. The diffn visualizer gives more information about which loaders are used at what times.

Remarks

One version of the program includes a diffn with 3 dimensions which cannot be visualized.
The cumulative visualizer on tanks only shows when there are simultaneous fill or empty steps on different tanks. There is no way to visualize how much of the tank capacity is used at any time.

The visualizer windows should not be closed during a debugging session, otherwise an error occurs[8].

There is still no support for syntactic and semantic errors, nor for errors due to data inconsistency. Especially the latter are very annoying, since it is very difficult to know if the inconsistency is due to the data or to a modeling error. The following error is shown when the cycle constraint is broken:

?- top.
cycle(1,[16, Ovar in {1,3..15,17..20}, Ovar in {1..2,4..15,17..20}, Ovar in {1..3,5..15,17..20}, Ovar in {1..4,6..15,17..20}, Ovar in {1..5,7..15,17..20}, Ovar in {1..6,8..15,17..20}, Ovar in {1..7,9..15,17..20}, Ovar in {1..8,10..15,17..20}, Ovar in {1..9,11..15,17..20}, Ovar in {1..10,12..15,17..20}, Ovar in {1..11,13..15,17..20}, Ovar in {1..12,14..15,17..20}, Ovar in {1..13,15,17..20}, Ovar in {1..14,17..20}, Ovar in {2..15,17..20}, Ovar in {1..15,18..20}, Ovar in {1..15,17,19..20}, Ovar in {1..15,17..18,20}, Ovar in {1..15,17..19}], [0, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}, Ovar in {0,72,9999}], 0, 10000, 0, [1], unused, unused, unused, unused, Cost in {0..10000}, [[0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [9999, 0, 72, 72, 72, 0, 72, 0, 0, 0, 72,

[8] Error 1025 : no window with this name in window_current(diffn_tanks) "c:\development\chip5/PLLIB/visualize.pl", line 462: visualize_reset_ltsrsb(diffn_tanks)

0, 0, 0, 0, 72, 72, 72, 0, 0], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0, 0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 72,0, 72, 72, 72, 0, 72, 72, 72, 0, 0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0,0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 0, 0, 0, 72, 72, 0, 72, 0, 72, 0, 0, 0, 0, 0, 0], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0, 0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 0, 0, 0, 72, 72, 0, 72, 0, 72, 0, 0, 0, 0, 0, 0], [9999,0, 72, 72, 72, 0, 72, 0, 0, 0, 72, 0, 0, 0, 0, 72, 72, 72, 0, 0], [9999, 0, 72,72, 72, 0, 72, 0, 0, 0, 72, 0, 0, 0, 0, 72, 72, 72, 0, 0],[9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0, 0, 0, 0, 72, 72], [9999, 0, 72, 72, 72, 0, 72, 0, 0, 0, 72, 0, 0, 0, 0, 72, 72, 72, 0, 0], [9999, 72, 0, 0, 0, 0, 0, 0, 72,72, 0, 72, 0, 72, 0, 0, 0, 0, 0, 0], [9999, 0, 72, 72, 72, 0, 72, 0, 0, 0, 72, 0, 0, 0, 0, 72, 72, 72, 0, 0], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72,72, 0, 0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0, 0,0, 0, 72, 72], [9999, 72, 0, 0, 0, 72, 0, 72, 72, 72, 0, 72, 72, 72, 0, 0, 0, 0, 72, 72], [9999, 72, 0, 0, 0, 0, 0, 0, 72, 72, 0, 72, 0, 72, 0, 72, 0, 0, 0, 0, 0, 0],[9999, 72, 0, 0, 0, 0, 0, 0, 72, 72, 0, 72, 0, 72, 0, 0, 0, 0, 0, 0]]])
no
-?

Figure 16 Error message and resulting 'no' answer of CHIP, due to the violation of a cycle constraint

Shortcomings / possible extensions

When clicking on a node in the search tree, the visualizers are automatically updated to reflect the situation at that point. However, this is only the case if *all* variables that determine rectangles in the visualizer are made search tree nodes (e.g. for a diffn visualizer, a rectangle is defined by both the machine *and* start or end time of a step involved; having only the machine variables as search nodes does not suffice). The need to introduce extra search nodes complicates the views in the search tool. The view is mixed up with information that may be irrelevant for the user. Moreover, in the domain state view, the domains of machine variables are much smaller than those of start/end times of steps, and the view is scaled with respect to the largest domain (cfr. figures below). More details can be obtained by zooming in, at least if the difference in domain sizes is not too large.

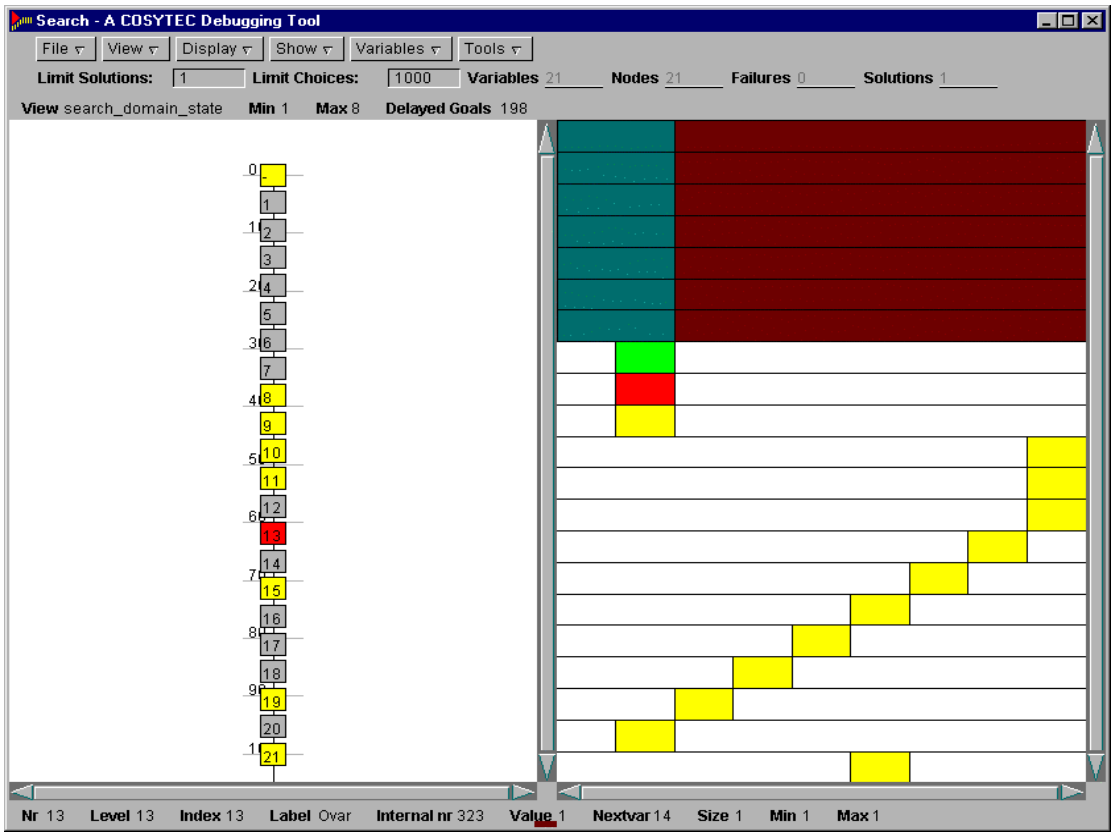


Figure 17 diffn visualizer with only machine variables as search nodes

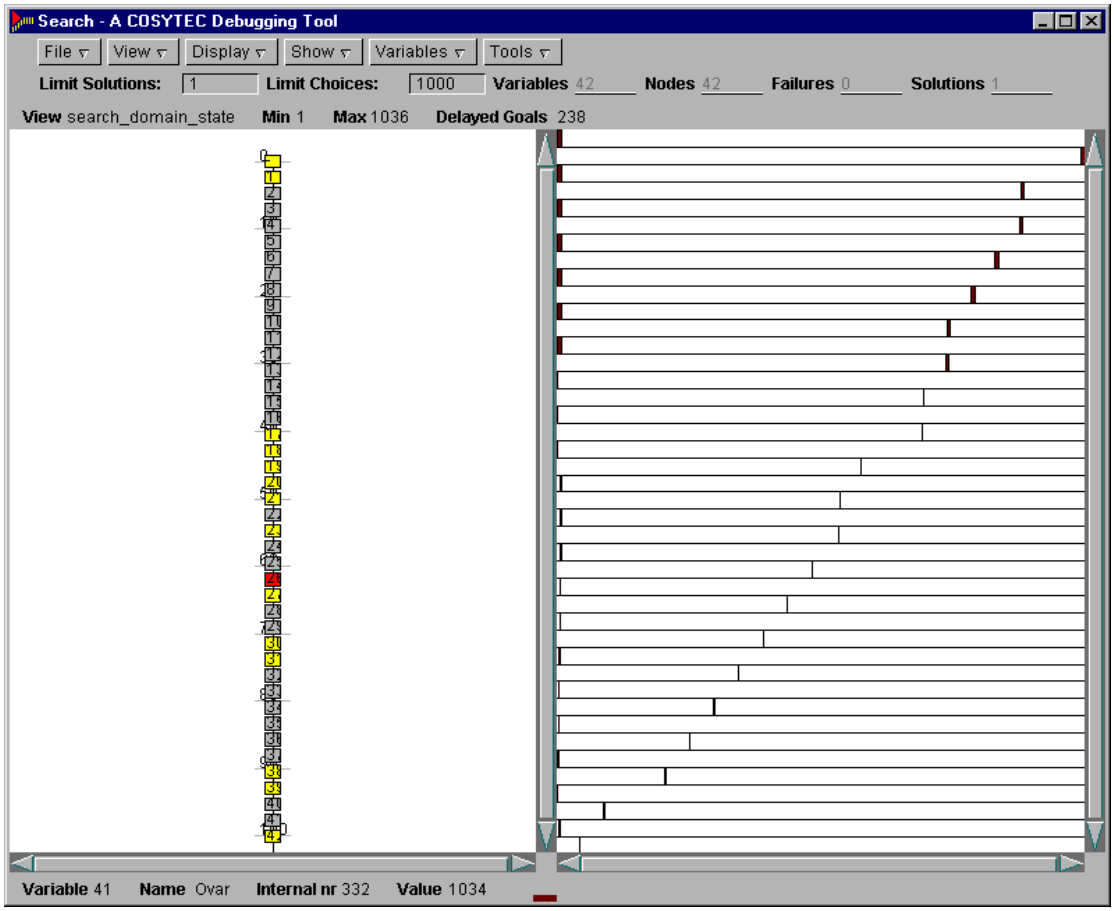


Figure 18 diffn visualizer with both machine and start variables as search nodes

The user should have the possibility to label the steps in the visualizers with more useful information, e.g. the step identification and the product involved.

There is no uniform notation (numbering) of steps in the different visualizers (step numbering depends on the occurrence of the step variable(s) within each particular constraint). E.g. the cycle visualizer and diffn visualizer use a different annotation for the same fill or empty step.

When zooming in within the visualizers, the legend on the axis is not refined accordingly.

Another nice-to-have would be the ability to link several related visualizers (with same x-axis), such that scrolling / zooming in one visualizer (along the x-axis) triggers a corresponding action in the related ones.

The end user still would prefer to have more debugging support in case of data inconsistencies.

2.7.3. Evaluation (Part II)

The reactor steps have to be scheduled in such a way that the setup time + lateness cost is minimized and that the tank planning (see Part I) is still feasible.

Tools used

Additional *visualize_assignment* and *visualize_graph_lines* in combination with the search tree tool. Tools version 15/10/98.

Use of the information

In this case the cycle visualizer and diffn visualizer both show the sequence of the production steps on the reactor. The diffn visualizer better reflects the time aspect of the problem, the cycle visualizer better reflects the number of tasks already placed. Unfortunately, the steps shown in the visualizers do not reveal which product is involved nor how high the associated setup and lateness cost is. The search tree view, however, allowed to detect that the node numbers in the cycle constraint were not defined correctly. Once the constraints on the reactor scheduling were modeled correctly and the first feasible solution had been found, the search tree view also was a real help during the optimization of the solution. The format of the search tree helped the user to detect trashing and showed at which point the program performed (useless) backtracking.

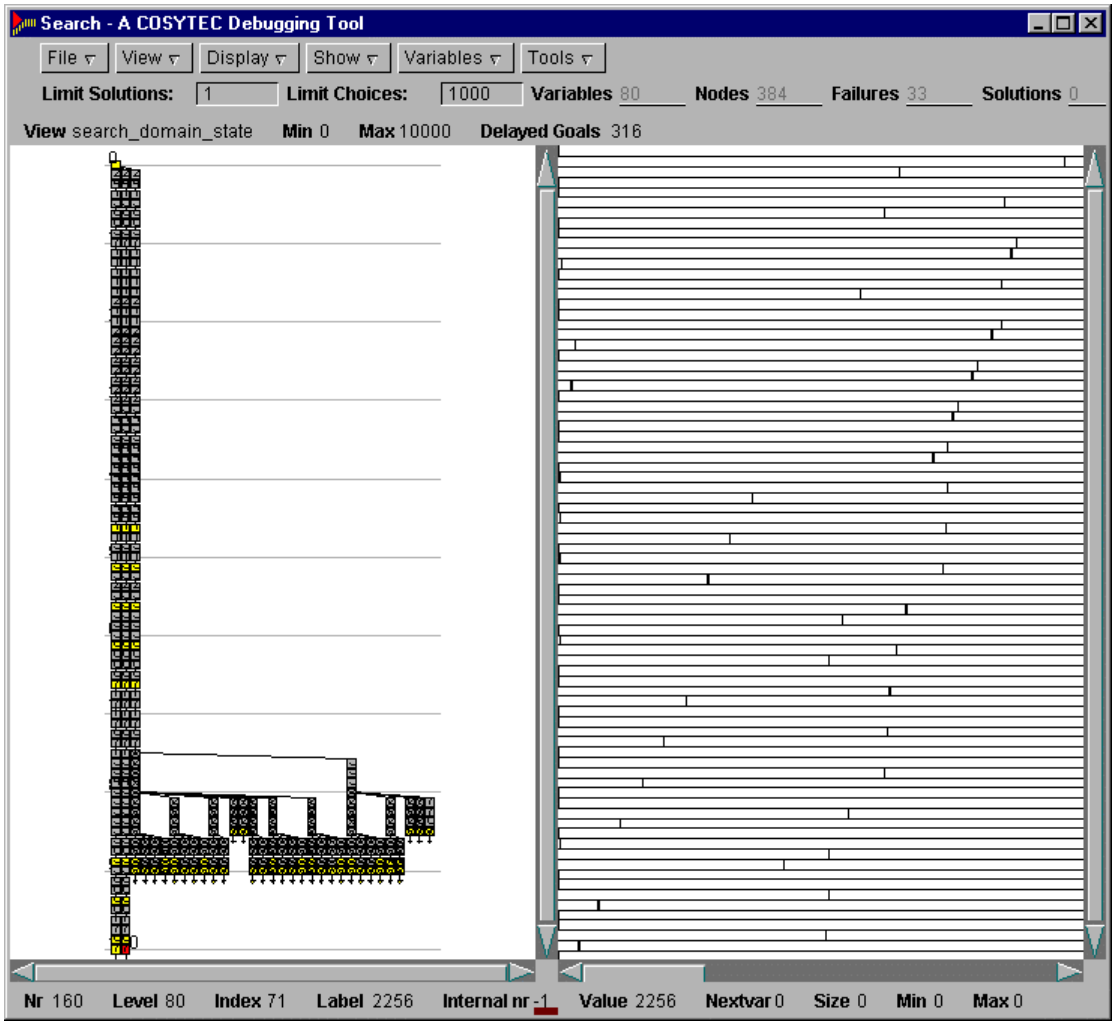


Figure 19 The (incomplete) search tree view reflecting useless backtracking (trashing).

The trashing could be avoided by changing the labeling routine. The users realized that it has no sense to try every successor of a node. It was sufficient to try, within the same product family, only the step (i.e. successor) with the earliest duedate. The search tree obtained in the final version of the program is shown below. The ameliorated labeling routine allowed the user to obtain an optimal solution in a reasonable time.

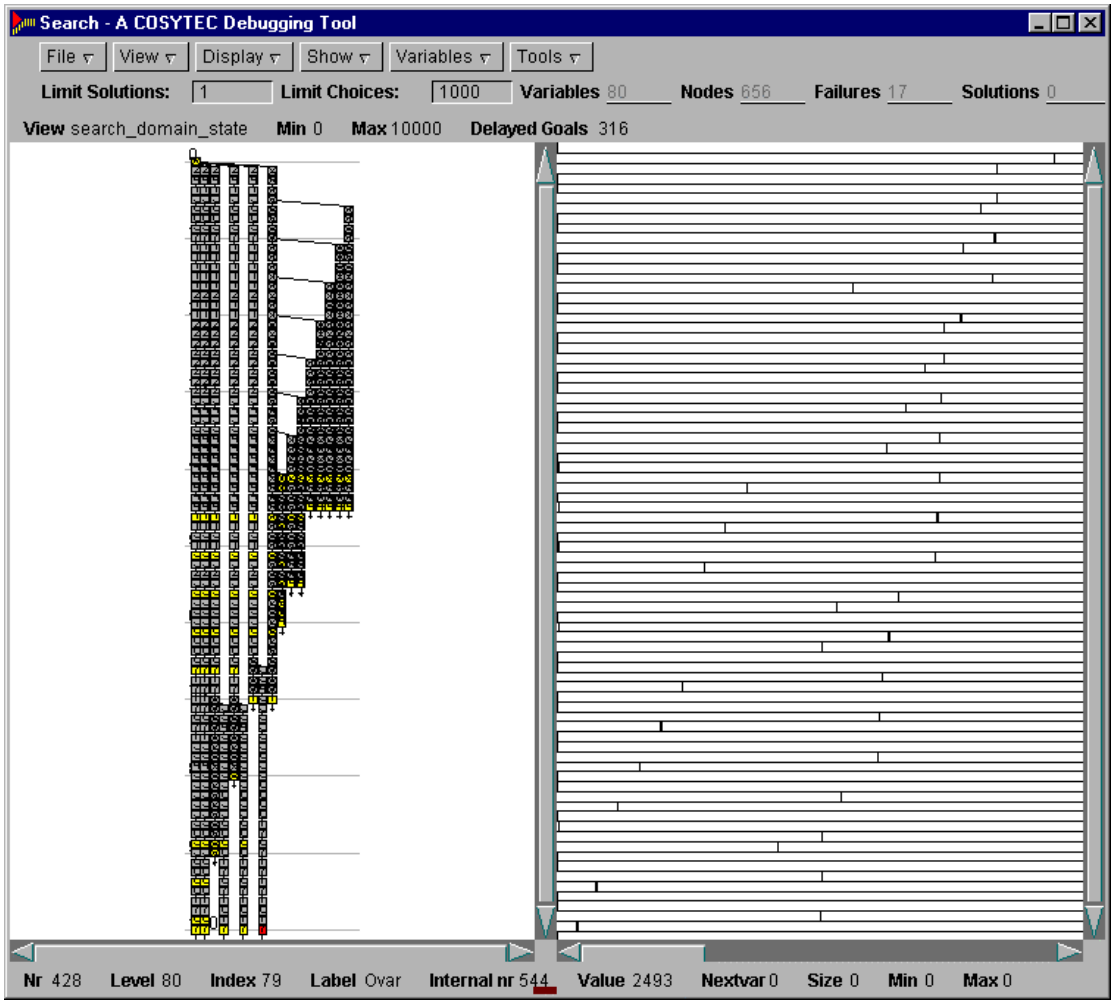


Figure 20 The complete search tree obtained for the final version of the program

Shortcomings/extensions

The end user would like to have more possibilities to show application dependent information in the global visualizers.
The user finds it still difficult to compare the domain state views at different nodes of the search tree. Also the difference in variable ordering in the different branches is not easy to follow. Perhaps a new view could help.

2.8. Conclusions

2.8.1. Novice user experience

Profile of the user

Five years of experience with Prolog programming. Knowledge of constraint logic programming. No experience with programming in CHIP (use of advanced global constraints).

Experience with the tools

GUI evaluation

The graphical representation of CLP problems through the search tree tool and global visualizers is what a user needs to understand and explore different search strategies. The overall view (shape of the search tree) already gives an intuitive picture of how well the search strategy works (what is the degree

of propagation). Afterwards, the user can inspect the execution in more detail, by clicking on the nodes of interest (comparing different nodes) and investigating the different views. The tool allows complete navigation through the tree, not just between adjacent nodes.

The domain state and update views in the search tree tool are quite easy to understand and give information about how much propagation occurs in each step. Working out the propagation view requires some more effort from the user. The propagation events view is even less accessible for an end-user, but may no doubt provide essential information to the implementers of the system.
A shortcoming with respect to search node annotation is that annotation is limited to predefined properties (variable index or level or value, size of the domains,…). Especially when dealing with large problems, the user should have the possibility to define an application specific node labeling.

The global constraint visualizers offer graphical representations that are tailored to the specific problem at hand (to a specific use of the constraint). The fact that they can be used in combination with the search tree tool results in a powerful toolset. A drawback is that there is no uniform notation (numbering) of objects over the different visualizers, which makes it more difficult to compare these views. Moreover, as with the annotation of search nodes, the user would like to have a more intuitive notation that provides useful information for the particular problem he's considering.

Learning the tools
Program annotation in order to use the search tree tool is quite simple. However, some care is needed with respect to variable ordering (the search node number must correspond to the position of the variable in the list given to *search_start* or *search_labeling*). Keeping track of the search node number requires minimal effort when using CHIP objects (in that case the number is just an extra field in the object structure); it's not necessary to extend or build datastructures to link the numbers with the variables.

Adapting the program to use the global visualizers was very straightforward, as it only involves adding a simple wrapper around the global constraints. Even standalone use of the visualizers (not in combination with the search tree tool) is easy: it suffices to use *visualize_update* and some kind of break mechanism (such as *system(pause)*) to follow program execution.

User considerations
The user is quite impressed by the information and possibilities offered by the tools. They provide both an overview on the problem as well as detailed information. Setting up the tools to work on a given problem is very easy. It requires only minor annotations in the source program. Even without much experience, useful information can be obtained. For some of the views tough (such as the propagation view), help from an expert user is desirable.
Some shortcomings:

- Constraints introduced during the search are not remembered by the search tree tool (e.g. the extra constraint introduced by *min_max* after a first solution has been found). This results in a wrong image in the domain state view (the domain may actually be smaller than shown). The correct domain can only be seen in the indication of the domains in the search nodes.
- Combining the global constraint visualizers with the search tree tool has the advantage that the visualizers are automatically updated when navigating through the search tree. However, it also requires that *all* variables determining items (e.g. rectangles) in the visualizers are included as search tree nodes. These extra search nodes complicate the views in the search tool, mixing up the relevant information with irrelevant details.
- Standalone use of the visualizers may provide more information than in combation with the search tree tool, e.g. if the predicate *min_max* is involved. The reason is that the search tree tool (cfr. above) does not remember constraints introduced during the search.
- Scrolling within the visualizer windows is not so handy: precise dragging is missing.
- Zooming in should also refine the legend of the visualizer.
- Some visualizers (e.g. diffn, *visualize_placement_2d*) do not have a legend on the X and Y axis.
- As mentioned above, there is no uniform notation (numbering) of objects over the different visualizers.
- The following minor bugs should be fixed: abort when visualizers are closed because some *visualize* routines do not check this (cfr. tank scheduling application), abort of cumulative visualizer if no solution can be found (cfr. small scheduling demo), need to include the search

library even for standalone use of the visualizers (cfr. rectangle placement problem), absence of lines for the root level and the one just below in the spy window (cfr. send more money example).

2.8.2. Expert user experience

Profile of the user

Nine years experience on constraint programming, i.e. 3 years using CHARME (Bull), 6 years using CHIP (Cosytec). No Prolog background. Knowledge of classical OR techniques.
The user has used CHIP in several (running) industrial applications, in particular as a solver for real-life short term scheduling applications. The user is a fan of global constraints [SC] and is convinced that they are the main reason for successful real-life CP programs.

Experience with the tools

GUI evaluation
The GUI of the search tree tool is very powerful and contains interesting views on the search tree space.
Although most real-life problems still have to be scaled, the number of variables and constraints that can be used in the search tree tool is big enough to test valuable cases. The search tree itself gives an immediate idea on the efficiency of the labeling routine. Interpreting the domain state views, is often more difficult, due the fact that variables often have a very different domain size. E.g. the domain for a Machine is typically 1..10, where a Start time variable has typically a domain of 0..20000.
Comparing the domain states at different nodes is not easy (especially at success leafs, when all variables are instantiated). The user also had some problems finding out the change in variable ordering in the different branches in case of large problems.
The GUI's of the global visualizers are very good for a "more application oriented view" on the behavior of your application. These visualizers make it possible to control:

- where on the planning horizon a task is placed and in which sequence (from left to right, randomly, etc.),
- if the global constraints take into account fixed tasks correctly,
- if variable lists like successors and alternative cycles in the cycle constraint are defined correctly,
- etc.

However, very quickly there is a need to extract more information from these visualizers. Although the end user is aware that it is not the intention to compete with dedicated application GUI's, an amelioration could be:

- User defined labeling and/or coloring of the variables
- Consistency between the labeling of variables in the different visualizers
- Possibility to add more information to the visualizers.

Learning the tools

Due to the documentation, demos and short training given by Cosytec, it was relatively easy to start using the debugging tools. Once the user understands the annotation method, it can be reused relatively easy. However understanding the meaning of the information shown as well as learning how to use the different views of the debugging tool took some time. A debugging tool methodology would be helpful.

User considerations

The user is very enthusiastic about the new debugging tools. They give the user an excellent idea on what is going on, and encourages him to optimize the search strategy. Without these tools, a lot of text tracing and pencil + paper work was needed.
However, some problems defined as debugging needs are not yet solved:

- The program has to run before the debugging tools can be used.
- Due to data inconsistencies, a correct CHIP program still stops with the not very explanatory message: 'no'. The user knows that such data inconsistencies can be avoided by extended control on the input data, but some help in detecting the errors would still be nice.

The user is convinced that writing new CP programs in such a way that they can be executed with/without debugging tools, facilitates the maintenance of these programs.

3. The evaluation of the CHIP debugging tools by ICON

This chapter starts with an overview of the applications examined by ICON, together with the debugging tools used and the most important remarks of the end users. The following sections contain a detailed problem description and an evaluation of the debugging tools used, for each application separately. The last section of this chapter summarizes the conclusions of both novice and expert user.

3.1. Summary of the applications

Academic examples and course exercises

Application: N Queens problem using *alldifferent* or *among*
Programmer: ICON
Tools used: Search tree tool, Local stack -, Global stack -, Utime visualizer
Use of the information: study of the constraint propagation mechanism and labeling strategy
Remarks: some bugs and inconveniences in the search tree tool are found. The memory consumption of the search tree tool has been studied.
Shortcomings/extensions: A strong demand to link the internal representation index with source code names. The possibility to allow the user to define his own views. A global constraint visualizer for *among*.

Application: Ship loading problem
Programmer: ICON
Tools used: Search tree tool and *visualize_cumulative_resource*
Use of the information: extensive study of the propagation due to the cumulative**.**
Remarks: some bugs and inconveniences in the search tree tool and global visualizer are found. It is difficult to derive information about failures caused by the *cumulative* constraint
Shortcomings/extensions: proposal for the visualization of the "induced nodes" and for more information in the failure_nodes

Real-life applications

Application: Layout of mechanical objects part 1: graph coloring
Programmer: ICON
Tools used: Search tree tool
Use of the information: detection of duplicate disequality constraints and of symmetrical solutions, verification of the quality of the implemented search strategy and amelioration, reduction of the search tree.
Remarks: study of CPU and memory usage of the search tree tool
Shortcomings/extensions: detailed suggestions on the search tree tool views

Application: Layout of mechanical objects part 2: physical layout
Programmer: ICON
Tools used: Search tree tool , visualize_cumulative_resource
Use of the information: detection of inappropriate constraint awakening and useless propagation via the propagation view. Detection of an excessive number of element constraints via the incidence matrix. Speeding up of the application.
Remarks: some bugs found in the search tree tool in combination with the visualizer
Shortcomings/extensions: The interaction between the global visualizers and the search tree tool could be better.

3.2. N_queens puzzle

3.2.1. Problem description

This is the classical problem of placing N queens on a N*N chessboard in such a way that there is no mutual check.
Used constraints: *alldifferent* in a first formulation, *among* in a second one.
The labeling strategy assigns variables starting from the middle of the board; this is achieved by reordering the variables before the search and by choosing the values starting from the middle of the domain.

3.2.2. Evaluation: alldifferent

Tools used

Search tree tool
Local stack Visualizer, Global Stack Visualizer, Utime Visualizer

Annotating the program

The labeling routine was replaced by the special search_labeling/4 built-in, together with the annotations search_start/2, search_number/2, and search_node/3 in the assignment.
Annotating the code for search tree debugging was easy, except for understanding how to impose an order on variable visualization different from what defined in the labeling heuristic. More precisely, we spent some time to understand the relation between the index associated to each variable, the insertion order of the variables in *search_start*, and the visualization order of the variables in the *State* and *Update* tool views.
The needed transformations of the annotated code for reordering were as follows:

```
...
reorder(L, LL),
search_number(LL,Merge),
search_start(LL,labeling(Merge))
```

became:

```
...
search_number(L,Merge),
reorder(Merge, MergeOrdered),
search_start(L,labeling(MergeOrdered))
```

In fact, we discovered that to change the visualization order of the State view matrix rows the variation of the order of *search_start* variables was not enough, because the index in *search_number* also has to conform to the sequence in the first argument of *search_start*.

Use of the information

The variables/domains incidence matrix is surely the optimal representation for this problem, and shows immediately the constraint propagation effect on domains.

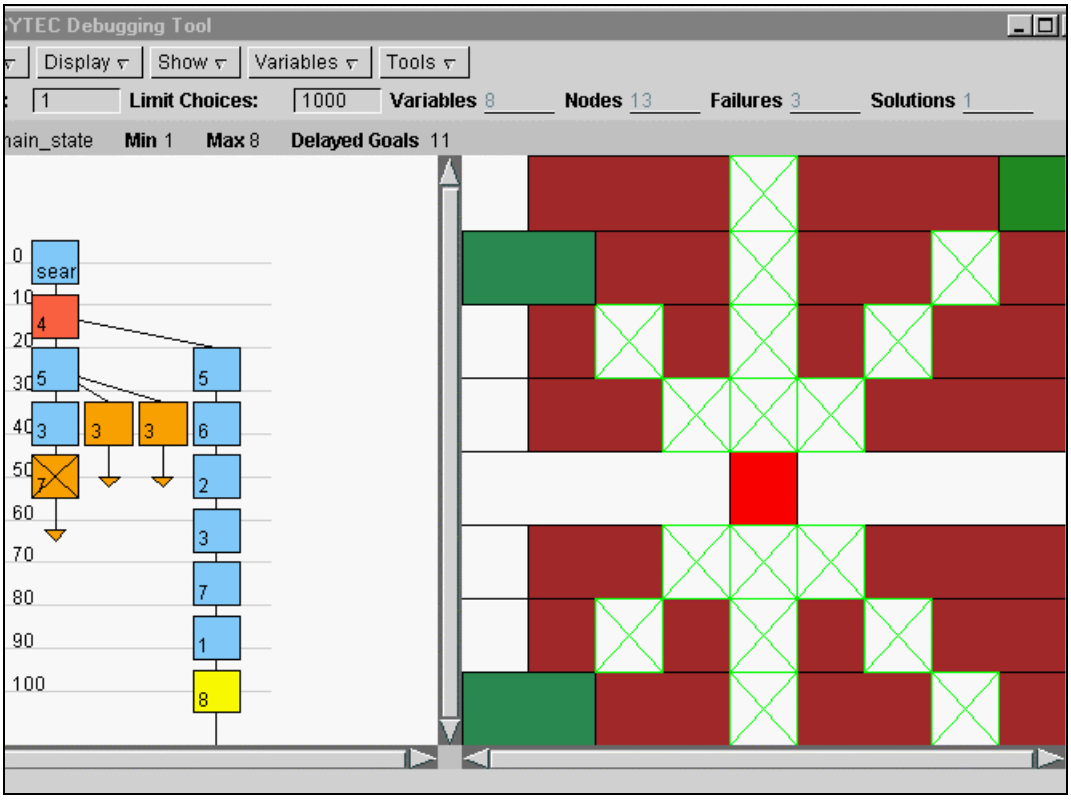


Figure 21 8_queens, *alldifferent*: domain state view

The CPU time visualizer, *utime*, reveals that for some instances of the problem the applied strategy wastes more time for the first solution.

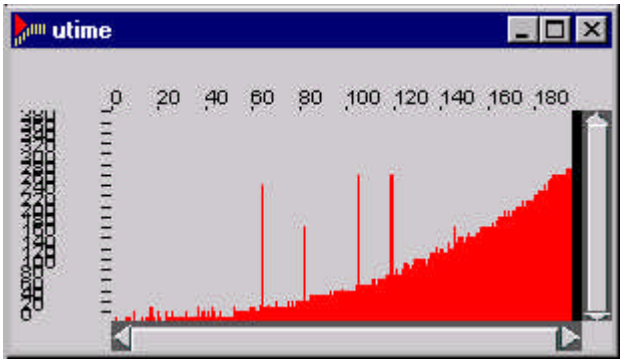


Figure 22 N_queens, *alldifferent*: utime visualizer screen shot

The search tree tool is very useful to analyze how the solver behaves in these critical cases. For instance, for N=68 we have the following search tree:

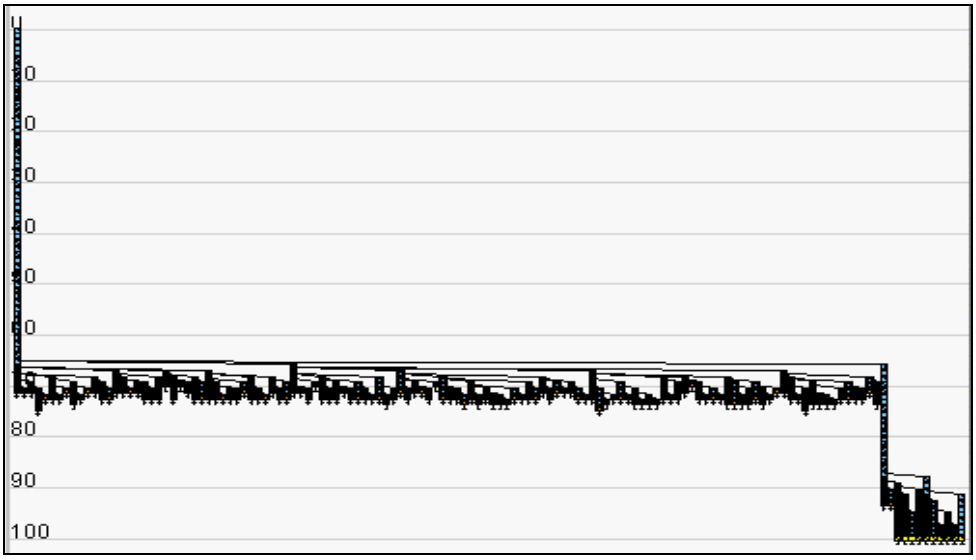


Figure 23 68_queens, *alldifferent*: search tree

Many deep backtracks are shown for the first solution, while the others are found quickly.
This analysis recommends using partial evaluation strategies, similarly to what Cosytec tried in the queen_credit problem for the "exceptionally hard instances[9]".

Remarks

Bugs found

- If a variable is spied, the menu item *Unspy Var* returns an error when the same variable is selected on the search tree[10]. Moreover, closing the spy-window for that variable and then selecting the *Unspy All* menu, the execution aborts with an error message[11].
- If the chip process is run with 40000 Kb of local memory, the local memory visualizer causes an exception[12].

Memory usage

During the Search Tree Tool evaluation, the variation of memory consumption has been evaluated.
The graphics for global and local stack usage are presented here, for N ranging from 8 to 200, without the Search Tree Tool.

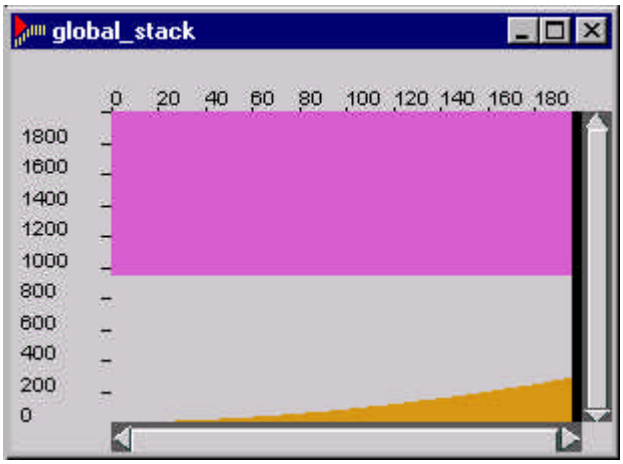


Figure 24 *alldifferent*: global stack usage

[9] See also the *credit* library in ChipV
[10] *Warning: undefined procedure called - seach_delete_spy(7)*
Error in event : callback failed search_menu_pressed(search_panel, search_var, search_var_nospy)
[11] *Error 1025 : no window with this name in window_current(search_spy_7) "indexed", line 0: search_reset_spies([7])*
[12] *Error 1009 : not a valid VDC in window_vdc(0, 0, 200, 40000) "g:\Program Files\CHIP V5/PLLIB/visualize.pl", line 373: visualize_create_ltsrsb(local_stack)*

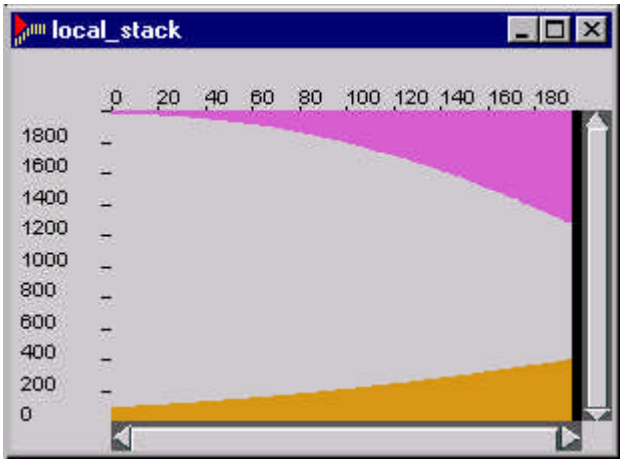


Figure 25 *alldifferent*: local stack usage

In the following table the measures without and with the Search Tree tool are compared for local stack, global stack, heap and trail, related to some problem sizes and for the first solution of the puzzle. A multiplication factor F shows the required increase of the dimension in order to use the Search Tree tool.

	Local			Global			Heap			Trail		
N	W/out	W/	F	W/out	W	F	W/out	W/	F	W/out	W/	F
8	49	137	2.8	9	3	0.3	1008	1270	1.3	2	6	3.0
68	132	806	6.1	44	48	1.1	1008	1555	1.5	82	114	1.4
111	209	1320	6.3	101	107	1.1	1008	1375	1.4	220	274	1.2
186	377	2259	6.0	255	264	1.0	1008	1453	1.4	618	711	1.2

Figure 26 *alldifferent*: comparison of memory statistics

The data highlights the local stack and heap increase, while other indicators remain roughly stable.

Other remarks

- With the Search Tree Tool usage, no predicate following *search_start* is executed until the item "Done" on the File Menu is chosen, since the debugging tool grabs the computation. So this command should be issued if one wants to change the board dimension or the tool termination conditions without restarting the application.
- If the index of some nodes of the Search Tree is not coherent with the order of the variables in the first argument of *search_start*, some nodes in the tool show an empty State view matrix.

3.2.3. Evaluation: among

Tools used

Search tree tool
Local stack Visualizer, Global Stack Visualizer, Utime Visualizer

Annotating the program

As in the *alldifferent* case

Use of the information

Just like the previous example, the N_queens puzzle *among* version was run to find the first solution. This time N ranges from 8 to 100 (instead of 200), as we detected longer computing times and larger memory utilization with respect to the *alldifferent* formulation.
The CPU time graphic shows that also in this case the most difficult instances of the puzzle are the ones with N=68 and N=86. The solution time is incremented by a factor of 10 in the *among* form.

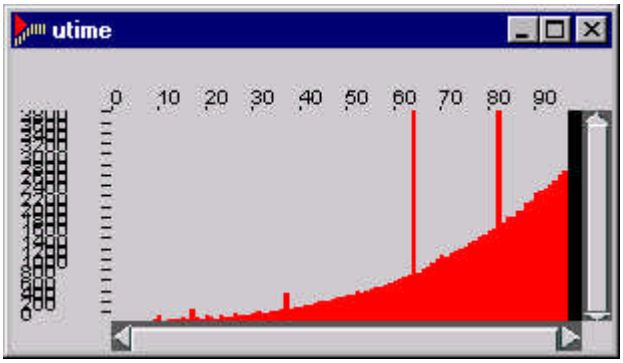


Figure 27 N_queens, *among*: utime visualizer screen shot

The Search Tree tool analysis for N=68 shows the same kind of branching. This denotes that the *among* approach of the problem, even if displaying a different constraint awakening and domain reduction, does not avoid the pitfalls of *alldifferent*. Therefore, this confirms what was supposed during the *alldifferent* analysis: these are hard instances of the problem, independently from the formulation.

Remarks

Memory usage

The *among* modeling generates a greater memory waste, which is about twice the previous case. The heap remains constant even now.

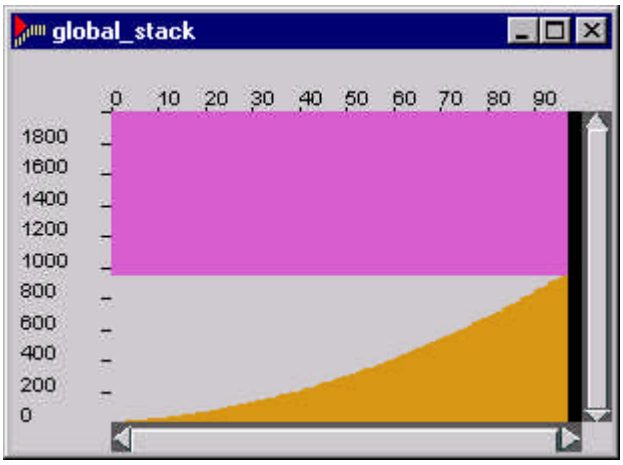


Figure 28 *among*: global stack usage

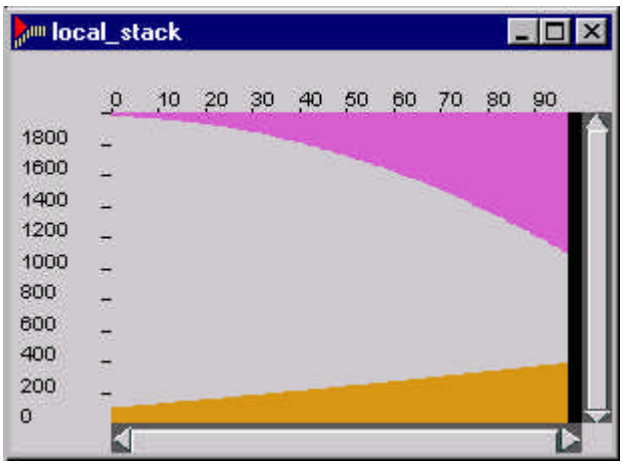


Figure 29 *among*: local stack usage

	Local			Global			Heap			Trail		
N	W/out	W/	F	W/out	W	F	W/out	W/	F	W/out	W/	F
8	59	227	3,8	15	15	1,0	589	1277	2,2	9	15	1,7
68	234	1476	6,3	468	445	0,9	589	1581	2,7	410	459	1,1
111	376	2387	6,3	1165	1100	0, 9	589	1416	2,4	1075	1156	1,1
186	660	3984	6,0	3127	2935	0,9	589	1518	2,6	2983	3121	1,0

Figure 30 *among*: comparison of memory statistics

Shortcomings / possible extensions

- The table views, e.g. *propagation*, are less handy when they do not link the internal representation index with the source code name. In other words, when pointing a row or a cell of the table it would be useful to refer to the variable/constraint used in the source, else the view may sometimes be cryptic. Better if a mouse click highlights the related code (in something like a specialized code editor) or if moving the mouse on a cell causes a tip to pop up with the variable name or the constraint definition.
- The 2D domain representation is quite immediate and useful. As an extension, a more general solution could be suggested, where the user has the possibility to define new views (the current ones of the Search Tree tool could be the default ones). The idea is to give users the possibility to define the representation which depicts better their problem: as a matter of fact, only in some cases a variable/domain incidence matrix fits, for instance in problems like N_queens where the constraints reflect a 2D relation.
- A global constraint visualizer for *among* is missing.

3.3. Ship loading

3.3.1. Problem description

This example is taken from the Chip demos. The problem consists in finding a schedule that minimises the time to unload and load a ship. The work contains a set of 34 elementary tasks, characterised by a given duration and a given number of necessary people.
Used constraints: disequalities (#>=, #<=) for precedence constraints, in order to balance loads on the ship, and a *cumulative* constraint because of the limited amount of resources (i.e. people) shared by the different tasks.

3.3.2. Evaluation

Tools used

Search tree tool, with *search_number*/2, *search_node*/3, *search_start*/2, *search_labeling*/4 adornments. *Visualize_cumulative_resource*.

Annotating the program

Annotating the program did not present any problem. In a former specification we tried the cheapest annotation method (the conversion of *labeling*/4 in *search_labeling*/4) but we found that this way the search strategy is distorted by the exclusion of the *min_max*.

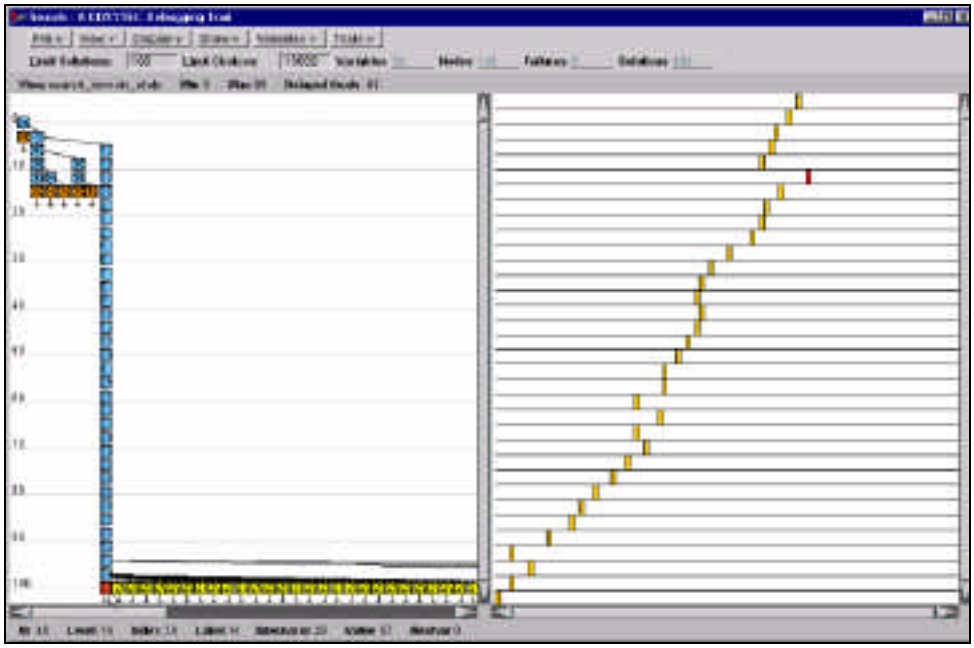


Figure 31 Domain state view, only *search_labeling*/4

The unaltered strategy behaviour can be observed annotating the program in detail with *search_number*/3, *search_node*/3, *search_start*/2.

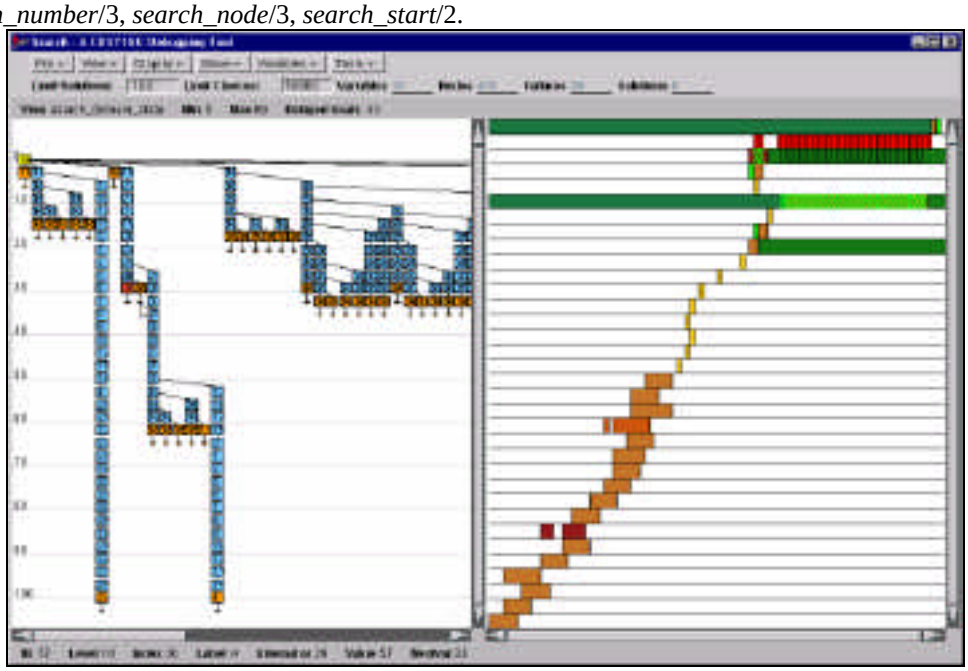


Figure 32 Domain state view, detailed annotation

Use of the information

In this problem there is one cumulative constraint contemporary to many simple ones. Thus it is not straightforward why a trace was found unsuccessful even before arriving at the Search Tree leaves: for which reason can a job not be assigned? Is it because of too few resources, or cost overflow with respect to a previous solution, or because precedence would not be achieved? Every time the user wants to understand the reason of a Search Tree failure, the *Propagation* or *Propagation events* views have to be inspected in order to verify the state of the constraint system.

Even in this situation the variables/domains incidence matrix representation suits to depict directly the precedence relations between the tasks and the pertaining propagation, whereas it is not apt to represent the relations on resource usage or request, as settled in the *cumulative* constraint. Adorning the program

with the global constraint visualizer for *cumulative* focuses the evaluation of the goal on the Search Tree node under analysis, so the corresponding variable can be left out of the inspection.

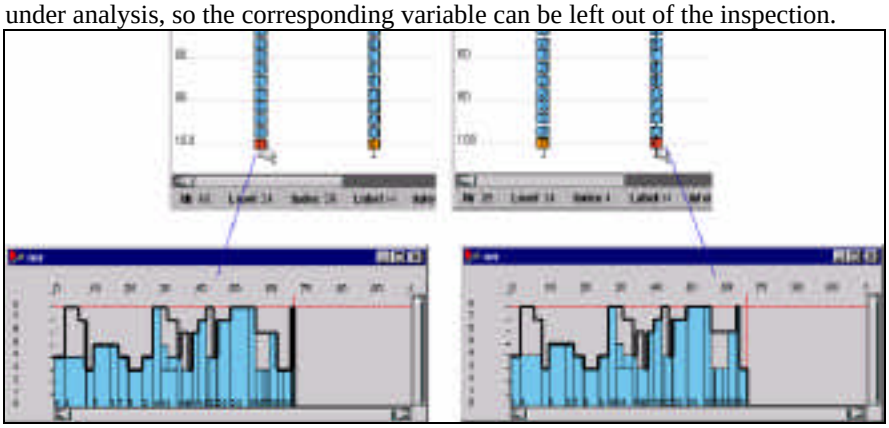


Figure 33 Cumulative visualizer correspondent to two different Search Tree nodes

Remarks

- If the user requests the visualisation of the legend (menu *Tools*, item *Show Legend*) in the views *Domain Update*, *Propagation*, *Incidence Matrix*, and *Variables*, the system crashes if the legend window is already displayed[13]. More generally, there is a kind of flicker between the legend and the main window: when the legend pops-up the main window disappears.
- The Global Constraint Visualizer windows must remain open during a program run, otherwise *visualize_update* will generate an exception[14]. This is true both for combined GCV/Search Tree interaction and GCV stand-alone mode. *Visualize_update* should simply succeed, and *visualize_show(Win)* could output a warning message on stderr.
- When using a cumulative constraint, the GCV wrapper will accept only a domain variable for the *End* parameter in the constraint specification, while without the wrapper a fixed number works too. If a number is used, the program aborts with an exception[15].
- The GCV cumulative visualizers seem to abort when there is no solution, e.g. when there is no solution because of the limits in the constraint specification[16].

Shortcomings / possible extensions

- On the Search Tree visualizer left pane it would be useful to differentiate the nodes where many variables become ground. With a good search strategy, a variable assignment grounds many other variables. On the search tree this property is completely lost: it is simply translated into node chains where these "induced" variables are chosen and assigned with their sole admissible value. In other words, the node that caused a big reduction appears like any other one, i.e. like any other choice/value pair not implying domain reductions. Even the nodes in such a chain look like any other node in the tree. Currently the user can find out these nodes only by means of the explicit observation of a null domain reduction in the Search Tree Tool right pane. The analysis of these cases can unveil strong variable dependencies, possibly not taken into account in the problem modeling or in the search strategy. In our opinion, any solution that differentiates graphically these nodes is worthy, as the user can immediately visualize how the problem complexity is greatly reduced (and then explore the details and the reason of the reduction).
- As noted above, it is not immediate to identify the *min_max* failure nodes, which can be settled by the detailed visit of every failure node in a *Propagation* view. This fact is troublesome when the tool is used for performance debugging, since the programmer is not helped in detecting the interesting nodes. Generally, it would be helpful if the user could somehow label the nodes that

[13] *Error 1025: no window with this name in window_current(search_legend).*
[14] *Error 1025 : no window with this name in window_current(sample_3) "g:\Program Files\CHIP V5/PLLIB/visualize.pl", line 462: visualize_reset_ltsrsb(sample_3)*
[15] *Error 121 : domain variable or natural number expected in domain_info(Ovar_1149880, Y1_6447, Y2_6447, __6450, __6451, __6452)*
[16] *Error 121 : domain variable or natural number expected in domain_info(Ovar_1149880, Y1_6447, Y2_6447, __6450, __6451, __6452)*

satisfy some condition. For instance, the user could tag with a particular colour any node that fails because of a certain set of constraints[17].

- The bugs of global constraint visualizers on update and window naming should be fixed.

3.4. Layout of mechanical objects: graph coloring

3.4.1. Problem description

This is an existing application developed at ICON. The problem consists in the determination of the configuration of mechanical pieces, which must respect several mutual relations. An instance of the problem is given by:

1. a set of objects to configure;
2. a set of relations to be satisfied.

Configuring an object means setting up its elements. These elements can be disposed with a set of physical assembling constraints. The relation that must hold between the objects (different for any problem instance) is satisfied by the identification of the elements to be included in any object.

The problem is divided in two main parts. This section is devoted to the first one, which consists in the identification of the mechanical elements to be inserted in each object, in order to satisfy the set of relations between them. This problem is practically a "graph coloring" with a complex specification, where the graph is induced by the set of relations between the objects. The goal is to find a colouring of the graph that minimises the number of used colours, which is given by a *min_max*. The domain variable selection strategy is *most_constrained*, and the value generation starts from the domain minimum value. Used constraints: disequalities (#\=).

The size of problem instances varies from one to several hundreds of objects and relations. Just as an example of the complexity of the constraint problem, on a small instance with 10 objects and 12 relations, 926 disequality constraints are posted on 65 variables to define the corresponding graph.

3.4.2. Evaluation

Tools used

Search Tree tool

Annotating the program

There were no major problems annotating the program for the Search Tree tool. The *min_max Timeout* parameter was included in our production application because an optimal solution is not always possible for any instance of the problem. We found that with the Search Tree tool this parameter is useless, as can be replaced by the GUI field *Limit Choices*.

Attempting to include in the tree the cost variable (which of course is not comprised in the labeling) required some extra work on the *search_start* adornment.

Use of the information

With the tool we found a first benefit without any particular intervention on the code, as two facts were immediately granted: the number of generated constraints and the number of variables. As a matter of fact, these two values vary a lot depending on the problem instance, and characterise well how hard is the solution.

The variables/constraints incidence matrix was very useful in finding disequalities (#\=) on variable pairs that were duplicated during the constraint generation phase.

[17] A failure node is characterized by the fact that the following choices fail for any possible value, but each failure could infringe different constraints.

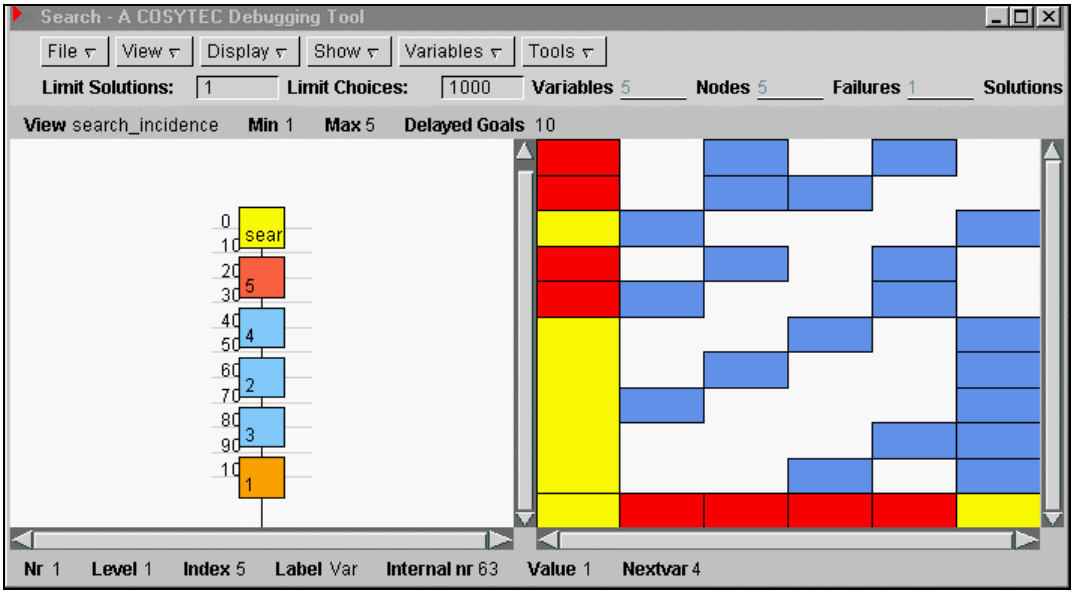


Figure 34 Layout first phase, incidence matrix

This event was already investigated during the past code development (of course without the Search Tree tool), and it is due to the complexity of the generation of constraints composing an instance of the problem. The number of duplicated constraints did not seem too high to condition in a heavy way the computation; thus this item was left out during the development. Anyway, this is an optimisation of the program easily discovered with the Search Tree tool (at least for small instances).
With the tool we verified that the implemented search strategy is useful to find quickly good solutions. However, even for medium problems the optimal solution requires the exploration of wide trees. For instance, the next picture reports a tree exceeding over 5000 nodes (restricted with the tool parameter *Limit Choices*).

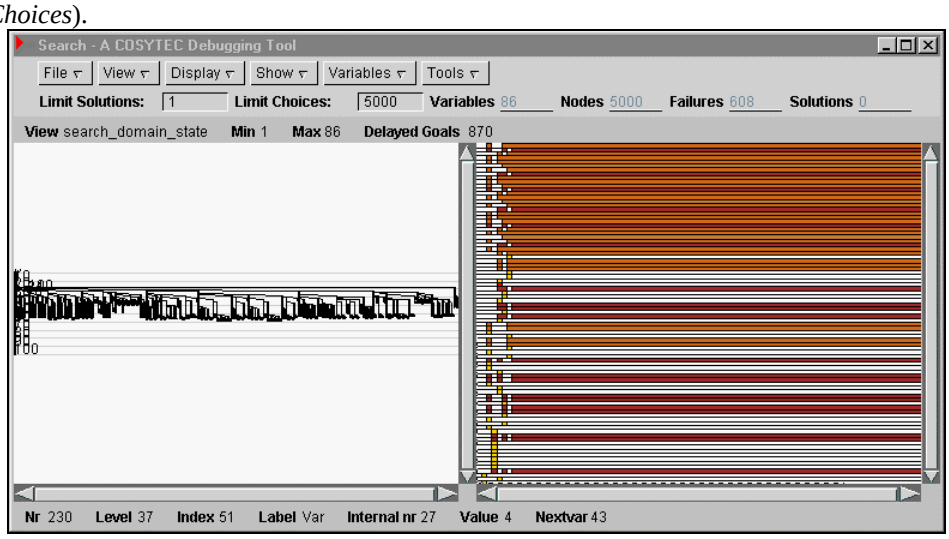


Figure 35 Layout first phase, domain state, limited search

During the analysis of small problems with the search tree, a direct consequence of the labeling came out. In fact, the labeling does not avoid the generation of solutions that are invariant for permutation of the variables (e.g., for any tree-colour solution to the graph colouring, there are three equivalent solutions).
In order to recognise this behaviour, we manipulated the tree view by collapsing some branches.

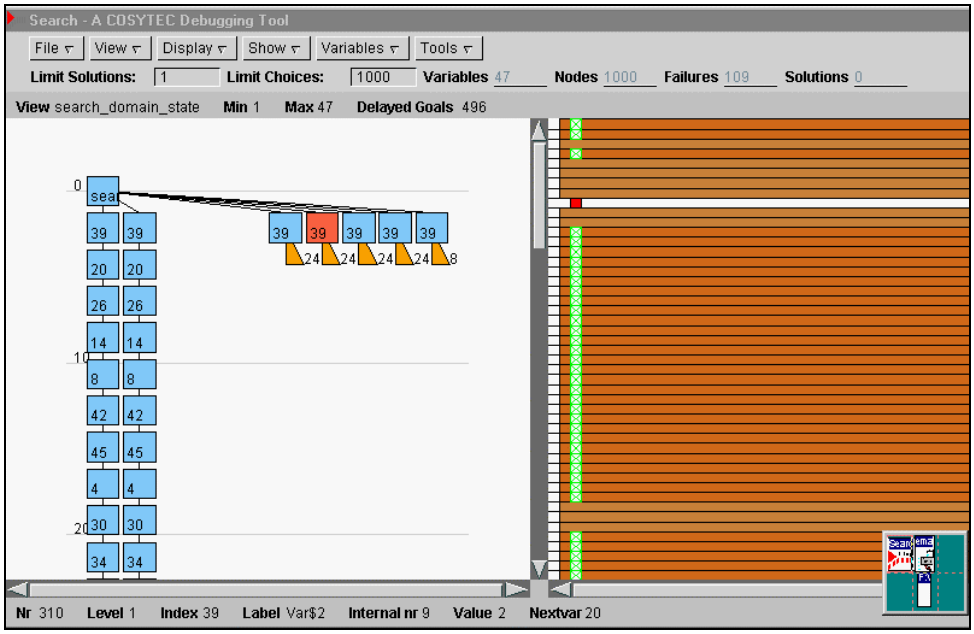


Figure 36 Layout first phase, collapsing branches

In the picture, collapsed failure branches are shown. These branches are equivalent, as after the first domain variable assignment they lead to the same propagation, differing only for a permutation of the assigned values. This is suggested also by the fact that all the subtrees have the same dimension (24 nodes, except the last one, which was interrupted by the choice limit).
In the next picture, the two found solutions are shown, each with the same value for the first variable. In addition, the first failure branch with the same first variable (index 39, value 1) is expanded. The search for a solution could stop even after the first failure branch exploration, as it represents the optimality proof of the previous solution: the following subtrees are equal to the expanded one, unless for the assignment order; for instance, the next branch starts with the variable 39, value 2.

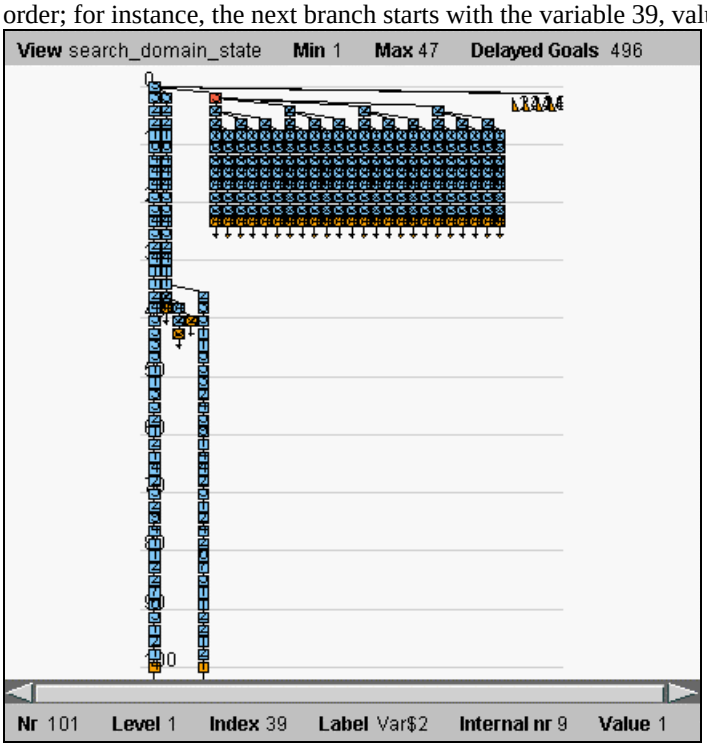


Figure 37 Layout first phase, first solutions found

This problem was considered even in the development phase, but the need for a quick near-optimal solution led to *Timeout*-limited search without making the generation function heavier. The actual

generation is based on the selection of most constrained variable, which is given with *indomain*/1 the minimum available domain value.

```
…
search_start(Ls,min_max(outlab_stv(Ls,Cost), Ls, 0, Num, 0)),
…

outlab_stv(Ls,Cost):-
        maximum(Cost, Ls),
    search_number(Ls,Merge),
        labeling(Merge, 1, most_constrained, assign_stv),
    nl, nl,
        write('Solution found with '),
    write(Cost),
        writeln(' positions'),
    tell(solution_stv),
writeln(Ls),
    told.
assign_stv(t(X,N)):-
        search_node(X,N,indomain(X)).
```

On average this strategy seems to lead quickly to a solution that uses few colours (therefore frequently optimal, even if the proof requires a deeper exploration of the search tree).
We modified the generation function, so to avoid the permutation effect above described. The labeling function then became:

```
search_start(Ls,min_max(delete_new(Ls,Cost,[0]), Ls, 0, Num,0))
…...

delete_new([],_,_).
delete_new([H|T],Cost,CostP):-
    maximum(Cost,[H|T]),
    delete(t(X,N),[H|T],R,1,most_constrained),
    search_node(X,N,assign1(X,CostP,CostPNew)),
    delete_new(R,Cost,CostPNew).

assign1(X,C,C):-
    C\=[0],
    member(X,C).
assign1(X,C,[H1|C]):-
    C=[H|_],
    H1 is H +1,
    X=H1.
```

With the Search Tree tool it was then possible to see the reduction in the tree.

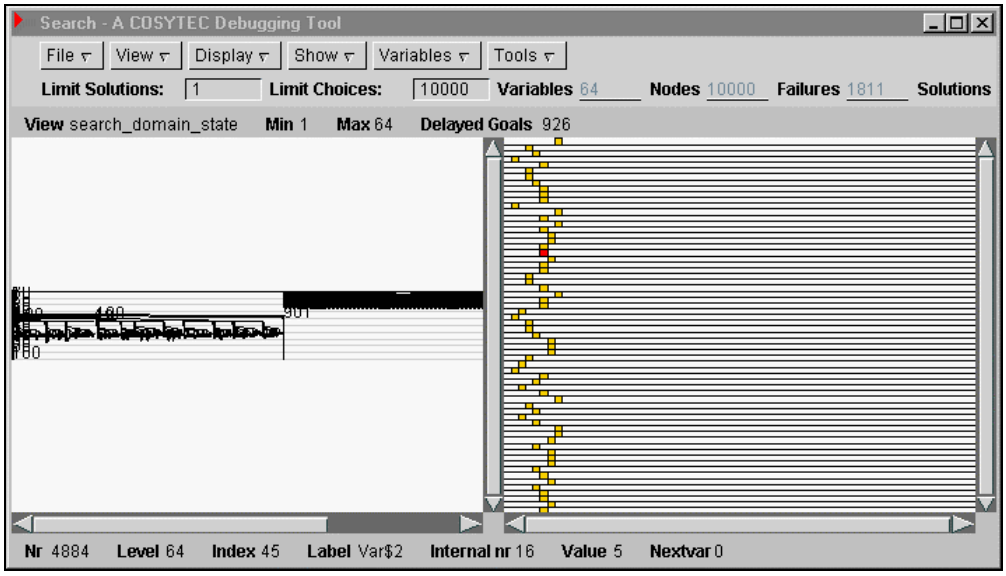


Figure 38 Layout first phase, example with old labeling scheme (outlab_stv)

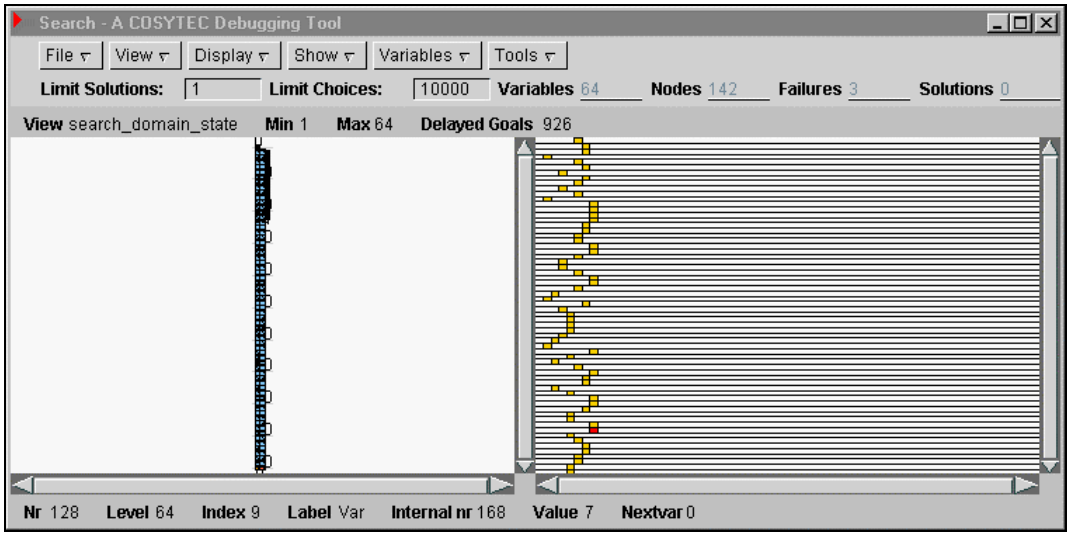


Figure 39 Layout first phase, example with new labeling scheme (delete_new)

Indeed, we noted that the instances of the problem vary greatly. Consequently, the same generation function does not show a uniform behaviour; i.e. it is not always better than the others, at least on execution times. In the following table, computation time for a good solution is compared on the two generation functions presented above, on two examples.

		With STtool		Without STtool	
		Old lbl	New lbl	Old lbl	New lbl
Example 1: 12 objects / 10 relations [926 constraints / 64 vars]	9 colours solution	23654		50	
	8 colours solution	401	32777	50	190
	7 colours solution	17546	431	1863	491
Example 2: 11 objects / 17 relations [870 constraints / 86 vars]	8 colours solution	26458	21911	60	391
	7 colours solution	491	471	80	5237

Figure 40 Layout first phase, comparing the labeling

It is worth noting that in the second example the two generation functions seem to adhere to what pointed out by the Search Tree tool, while without the tool the old labeling is faster. This seems to suggest that with the tool the execution time is not (always) comparable.

In the first example is evident that the new labeling is quite apt. The search tree is thinner: two solutions with 8 and 7 colours are found straight away, and the optimality is proved for 7 colours.

Remarks

Memory usage
The variation of memory consumption and the solution time has been evaluated on two examples, with and without the Search Tree tool. The generation function is the original one (labeling + indomain).

Example 1: Number of constraints: 926
Number of vars: 65
Solution limit: 1
Choice limit: 10000
In the first problem the application finds three solutions within 10000 nodes. For the third solution there is no optimality proof within this limit. Time values are in seconds, memory values in KBytes.

Sol.	Cost	Time with tool	Time without	Global mem with tool	Global mem without	Local mem with tool	Local mem without
1	9	22.332	0.050	77 GStack 4497 Heap	43 GStack 1680 Heap	5683 LStack 502 Trail	367 LStack 65 Trail
2	8	0.371	0.060				
3	7	16.975	1.672				

Figure 41 Layout first phase, example 1 statistics

Example 2: Number of constraints: 496
Number of vars: 47
Solution limit: 1
Choice limit: 1000
In this example, the application finds two solutions within 1000 nodes, and moreover the second solution is proved optimal

Sol.	Cost	Time with tool	Time without	Global mem with tool	Global mem without	Local mem with tool	Local mem without
1	7	11.726	0.040	44 GStack 2308 Heap	26 GStack 2121 Heap	3043 LStack 235 Trail	292 LStack 40 Trail
2	6	0.341	0.050				

Figure 42 Layout first phase, example 2 statistics

In a real world problem with 21 relations on 27 objects, 11473 disequality constraints are generated. Even with 80MB of local memory and 10MB of global memory the use of the Search Tree tool was impossible, as the local memory was found insufficient and the process aborted.

Other remarks:
- The possibility of an overall view of the generated constraints is interesting. It would be nice if for big problems (thousands of constraints, hundreds of variables) some utilities were available to group and reorder the incidence matrix view rows. For instance, in the present application it would help a lot grouping (and counting) disequalities on the same variables in order to discover how many useless constraints are generated.
- The execution time does not seem to be a parameter that can be analysed directly with the Search Tree tool (i.e., no straight indication comes out on how the application will usually behave).
- To analyse the application on real world situations (medium/big problems) even 100MB of memory are not enough.
- In the first example shown here, the third solution (cost=7) is not immediately visible: many subtrees must be collapsed to see the nodes after the 2000^{th}. Some utility would be helpful to navigate through wide trees, for instance to detect quickly the solutions or other interesting nodes.
- In problems that use the *min_max* predicate there is not a clear representation of the factors that influence the computation while searching any solution following the first one.

3.5. Layout of mechanical objects: physical layout

3.5.1. Problem description

This section is devoted to the analysis of the second phase of the mechanical object layout application described previously in detail.
Given the set of objects to configure, the second part of the problem is aimed at the physical layout of the elements in each object, meeting a fixed set of physical constraints (which are invariant for any

instance of the problem). In this phase there is no need of an optimal solution, whereas it is accepted the first solution properly mapping the elements identified by the first phase.
This stage involves the use of a *cumulative* constraint to define the physical constraints that mechanical elements must respect. Other used constraints are *among*, *element*, and *alldifferent*.

3.5.2. Evaluation

Tools used

Search tree tool, with *search_number*/2, *search_node*/3, *search_start*/2.
Global Constraint Visualizer : *visualize_cumulative_resource*

Annotating the program

The visualizer wrapper is quite simple, but the particular usage of *cumulative* required that the (useless) argument *End* was added to the constraint, as the only available visualizer was *visualize_cumulative_resource*.
For any object to configure there are two *cumulative* constraints, whose visualizers must be uniquely named and placed on the screen.
The code:

cumulative(StLs, DurLs, ResLs, unused, unused, 1, unused, unused).

was modified in:

End :: 0..88,
incval(lateral,Clique),
*WinY is (Clique-1)*90,*
sprintf(Name,"lateral_%d",[Clique]),
visualize(cumulative(StLs, DurLs, ResLs, unused, unused, 1, End, unused),
visualize_cumulative_resource,[winy<-WinY,winw<-350,winh<-70],Name).

Use of the information

The goal of the physical layout phase is to find a placement for the mechanical elements. For any object there are two *cumulative* constraints that express the physical assembling limitations. Each object element is represented with a task that uses one resource. Its “duration” is the space that could cause a placement conflict if owned by another element. The *among* constraint captures the optional distribution of the elements in particular areas of every object.
The first analysis was operated, with both Search Tree tool and Global Constraint visualizer, on a small example constituted of 7 objects and 7 relations. This way all the visualizer windows could be seen together with the search tree.

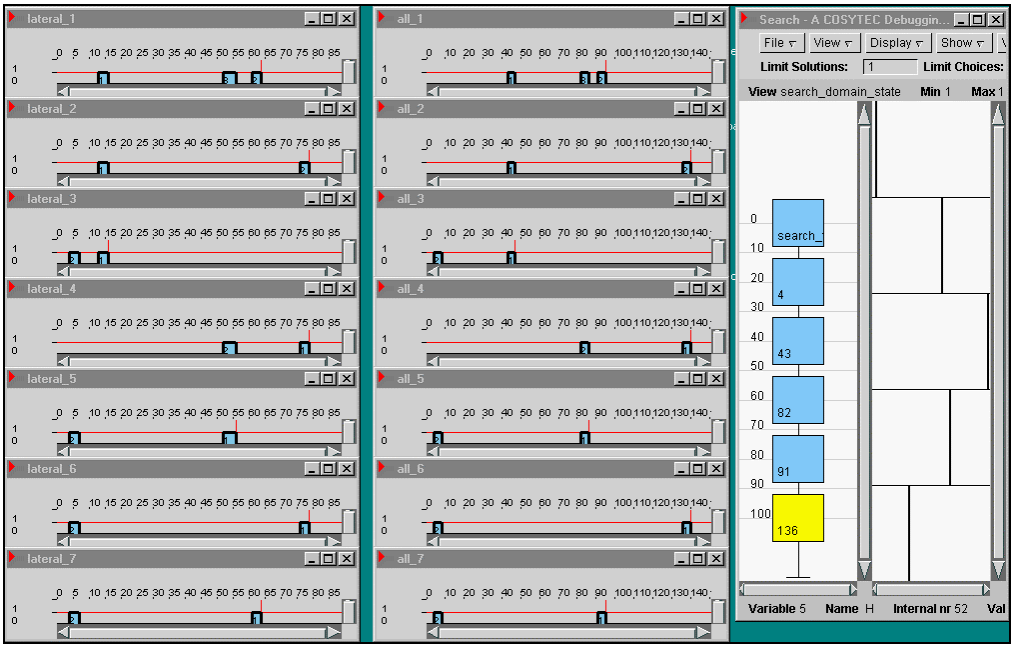


Figure 43 Layout second phase: *lateral_x*, *all_x* and Search Tree windows

For any object there were two paired windows (named *lateral_x* and *all_x*), representing the two physical constraints.
As it can be seen the picture, there were five elements to place (corresponding to the nodes in the tree) that could be joined differently within the objects. For instance, in the *all_1* window there were three elements, whereas in the *all_3* window there were two. A graphic representation of the element positions resulted interesting, even if the *cumulative_resource* representation is not completely adherent to our domain[18].
We verified on the search tree that for small problems there is no backtracking at all when the variables are assigned values that are compatible with the physical constraints and there are no *among* additional constraints. If an optional distribution is added, then the labeling initially leads to compatible assignments that do not comply to the added *among* constraint.

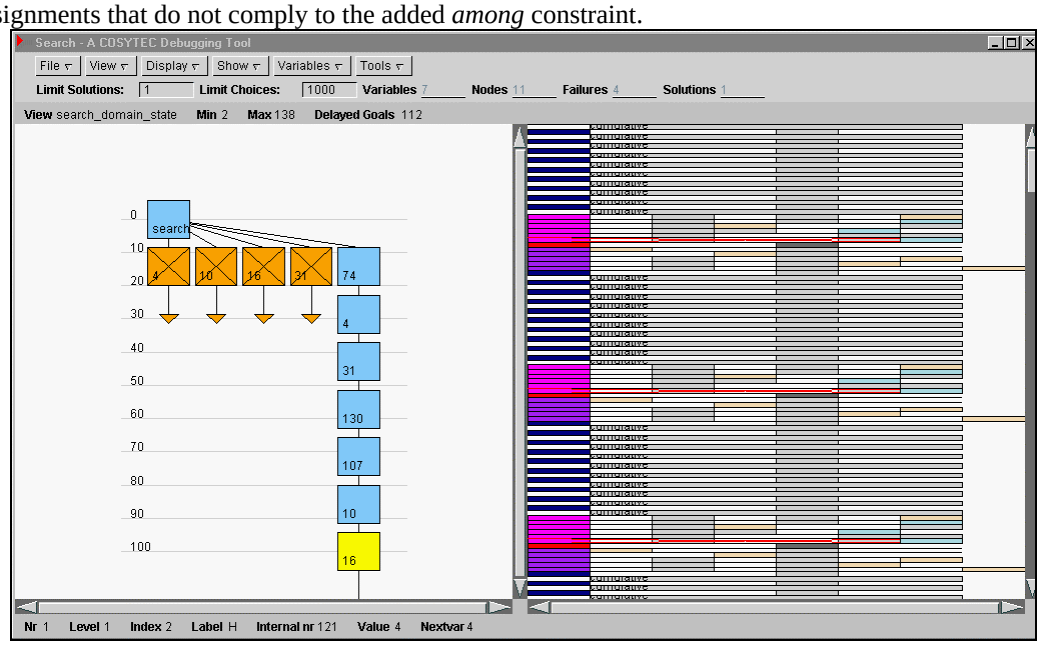


Figure 44 Layout second phase: failures due to *among*

In order to detect the reason for these failures, the *Propagation* view was quite helpful. For many problem instances we could find that there is no solution at all, due to the extra constraint *among*.
Then we analysed the program behaviour on real examples, in order to understand which is the most critical phase of the element placement. The example was constituted of 118 objects with 44 relations. During the second phase, 2089 constraints on 138 variables were generated. Obviously, we could not keep the first code annotation that would have created 236 windows. Hence, we changed the code in order to visualise only a set of selected objects, without the Search Tree tool. The *visualize_update* function let us follow the placement of the elements in the objects during the computation.

[18] In fact, this constraint models tasks on time intervals, whereas our model refers to elements and areas. Obviously, there are efficiency reasons why a *cumulative* constraint was used instead of a *diffn*.

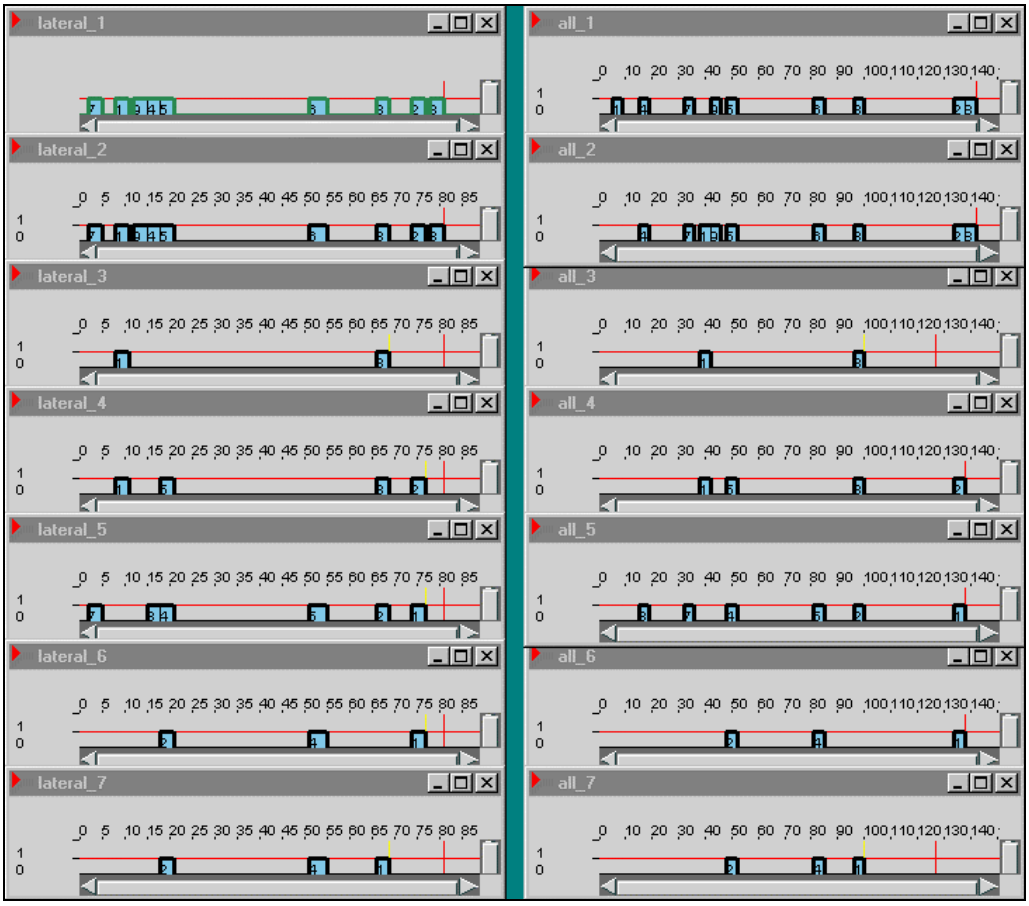


Figure 45 Layout second phase: elements placement

In the presented example the computation failed, but we could not understand the reason since the set of selected objects was meaningless. Hence we went back to the Search Tree tool to monitor all the computation, but we had an "out of memory" error even with 90MB of Local Stack.
On smaller problem instances, a detailed analysis of the *Propagation* view resulted advantageous, since it revealed a great number of inappropriate constraint awakening and useless propagation:

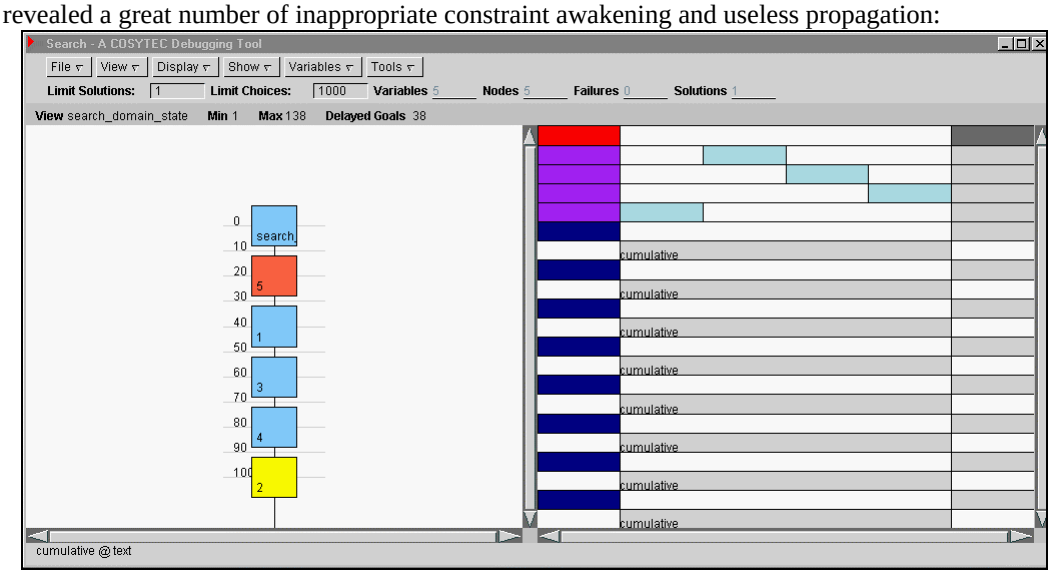


Figure 46 Layout second phase: *Propagation* view

Thanks to the *Incidence Matrix* view, we could find the excessive number of *element* constraints. It is worth noting that the piece of code responsible for this behaviour was developed a long time ago by other programmers, who do not maintain anymore this application. Identifying this anomaly would have been hard with standard debugging techniques.

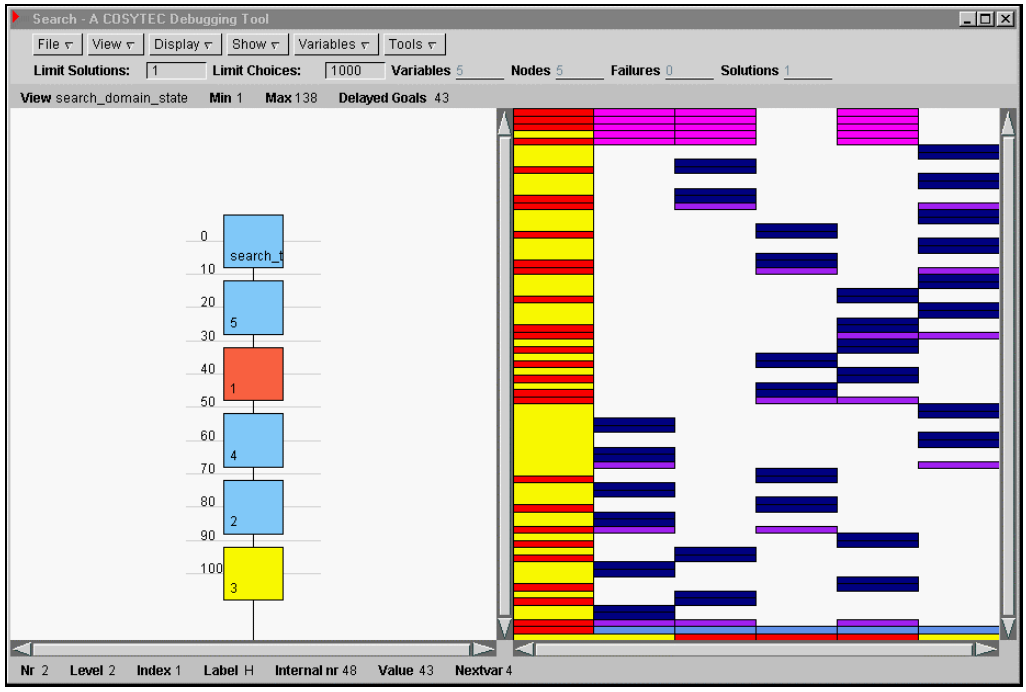


Figure 47 Layout second phase: *Incidence Matrix* view

Hence, with the Search Tree tool analysis the constraint generation anomalies were fixed. Specifically, even if there were no modeling errors, we modified the generation of the *element* constraints to speed up their definition.
The *Propagation* and *Propagation Events* analysis was a demanding job, but it allowed to understand at a deep level the real Chip behaviour when awakening the constraints.

Remarks

Bugs found

- *Visualize_cumulative_resource* will complain[19] if the cumulative constraint *End* argument is a constant. It must be a domain variable instead.
- Using the Search Tree tool with a Global Constraint visualizer, if one window is closed and the computation restarted, the window is not recreated and the execution aborts with an error[20].

Other remarks

- The association between tasks in different visualizer windows is based only on the domain variable order, as cited in the *cumulative* constraint. The index depends from the insertion order in the constraint, but the same task (variable) can appear in more than one constraint, with every possible ordering. A mouse selection does not identify a task, but instead the task position in the constraint specification. On other visualizer windows, the highlighted task will be the one in that position and not (as one would expect) the same task. The two things coincide only when the start lists in any constraint stick to the same ordering.
- It would be helpful if a mouse click on a search tree variable (node) highlights the connected tasks in the visualizer windows, provided that the variable appears in the related constraints.
- In the *Propagation* view there are constraints that involve both traced and non-traced variables. We had some trouble when relating the displayed constraints and variables with the source program ones.

[19] *Error 121 : domain variable or natural number expected in domain_info(Ovar_1149880, Y1_75, Y2_75, __76, __77, __78)*
[20] *Error 1025 : no window with this name in window_current(lateral_2) "g:\Program Files\CHIP V5/PLLIB/visualize.pl", line 462: visualize_reset_ltsrsb(lateral_2)*

3.6. Conclusions

The following two sections sum up the experience with the Search Tree tool gained by a novice and an expert user. Then final considerations follow.

3.6.1. Novice user experience

Profile of the user

Three years experience of Prolog programming, knowledge of resource optimisation, discrete and continuous LP problems, but only with the traditional view and instruments of operational research. No previous experience with constraint programming.

Learning CP

The user reports that the constraint-programming paradigm was immediately clear, just after some small examples guided by an expert user. Then the user noticed, studying alone another problem, that the resulting representation was still driven by the classical formulation, while the model was still missing the CP-oriented shape.
The usage of the Search Tree tool allowed him an easier understanding of the constraint programming concepts and modelling, and made him conclude that

> *It is simpler to shape problems under the constraints view, since the representation results more natural, closer to spoken description of the problem, while the classical formulation is more mathematical, less comprehensible: in fact, the complexity is hidden within the solver. This is an immediate advantage for novice users, but fine-tuning the solution search seems to require more CP experience and a conspicuous knowledge of Chip insides.*

Experience with the tools

GUI evaluation

The idea of a graphical tool for visualising the search tree seems suitable to the novice user. He had never tried to follow the behaviour of a CP program without the Search Tree tool, but he does not think he will ever do so after using a visual debugger. The user reports that the graphical representation is especially useful for the following reasons:

- it allows representing parts of a computation (regardless of non-termination or of saturation conditions) like domain variable assignments, failures, etc.
- it lets the programmer move quickly in any point of the computation and fetch the correspondent state;
- it permits the instant information on constraints, domains, variable values and their evolution (e.g., *Variable update* view);
- it suggests a ready evaluation of the used strategy, just by inspecting the shape and the (early) branching of the tree, and the depth of the different paths.

The global constraint visualizers are apt in relation to the Search Tree tool, where partial solutions can be easily related to the search tree nodes.

Learning the tools

During the learning experience, the novice user reported that understanding how to decorate the code for Search Tree tool usage was not difficult. It was more difficult to order the variables (and their related domains) in the Search Tree tool views; the user suggests to add just a small demo in the Chip distribution to explain the use of this feature.
At the same time, the user was not able to name a variable so that it was shown with that label and not with the internal name in the Search Tree tool. The views that have a table form (e.g., *Propagation*) are not so useful when they do not allow to connect the internal representation index with the source code name: it would be better to have a tip that reports the code name or the piece of code where the definition is.
Initially, the fact that the variables (or domains) appear either horizontally or vertically according to the selected view disoriented the user. Obviously this happens because in some views the tree node depth is represented vertically, so the other dimension must be shown on the other axis.
Learning how to use the global constraint visualizers was very easy, as the wrapper mechanism is elementary. Some more attention had to be paid to understand the displayed information.

Novice user considerations

- The novice user reports a quite positive judgement on the look and usability of the interface of the Search Tree tool. He misses only the scroll bar "active" dragging action.
- Examining big problems (i.e., with many variables and constraints), when a deep node is selected the user must wait a lot of time for the corresponding view to be drawn.
  If this is generally acceptable, it is a little frustrating when exploring nodes that are nearby in the tree. The most frequent action of the user (at least a novice one) is to select adjacent nodes and switch between the two so to depict the involved variations, but this way the views are always recalculated. Even complying with Cosytec's decision of not retaining all the traces for memory consumption reasons, the tool could memorise all the information pertaining to visited nodes (which are usually only a small subset of the overall tree), and/or save some traces. Hence, the generation of adjacent node views would be accelerated by computing only the difference between the two states.
- The novice user misses a representation that allows the observation of the implemented strategy from a more abstract point of view[21]. This representation could draw out summary information on (parts of) the implemented strategy. As a matter of fact, with the current tool it is possible to restrict the attention to a subset of variables/constraints, and thus to observe the behavior of the application at a small detail, but the user is not supported by the tool in deciding which subset of the domain is to be inspected with more attention.
- Minor bugs should be fixed and some improvements could be added in the global constraint visualizers, like the *visualize_update* function (which aborts when windows are closed), the zoom feature (which does not act also on graph labels), and the task index selection problem (where corresponding *indexes* are highlighted on other windows, instead of corresponding *tasks*).
- Other visualizers for global constraints could be added to cover also the missing representations.

3.6.2. Expert user experience with the tools

Profile of the user

Five years experience in Prolog programming. Knowledge of the classical techniques in linear programming, for typical operational research problems, on discrete and continuous domains. Knowledge of local consistency methods for finite domain constraint problems, and their integration in logic programming. Five years experience on Chip programming.

Experience with the tools

GUI evaluation

A graphical interface that shows the CLP computation is one of the best things for a programmer, especially if he/she had just to imagine what was going on, basing on specific CP development system knowledge.

Even the simple tree shape is enough to support the programmer in forming a representation of the solving process as compared to what foreseen. Opposed to the traditional debugging (text trace, debugging port model), with these tools there is complete control over the computation, also with the opportunity of browsing back and forth, and comparing different nodes. The availability of statistical information is precious, like the number of constraints, variables, failures, visited nodes.

In the Search Tree tool there is some problem in comparing different views, as the presented information is not always coherent. For instance, in *Propagation Events* each variable is named by its internal index, while in the tree the intuitive (conventional) index is used, being also the only identifier in *Spy Variable*.

Real problems are so large that sometimes the navigation in the tool views is complex, as only a small part of the problem can be visualized. The tree collapse feature is especially useful, even if it can be fully exploited only after some familiarity with the tool.

Graphic visualisation of global constraints is important too, because their meaning is clarified: in fact, they have a computational behavior that is not always obvious. The types of available views stick to the usual definition of the corresponding constraints, as they were originally inserted in Chip: this does not exclude that particular uses of a constraint are not properly represented in the available views.

[21] Of course, it is another issue if this is only a missing point in the present tool (thus a possible extension) or if the design of a radically different instrument is required.

Learning the tools

At first sight, the basic use of the tool is very easy. Adorning the code is simple and the essential features can be immediately explored. It is more complex to control the tool in order to show only some parts of the search tree or some interesting variables, or to force a certain order. The tree navigation is comfortable, apart from the slow refresh experienced in big problems.

The most intuitive views are *Domain State* and *Domain Update*. It is easier to analyse the tree with these views, since a lot of information is included on the variable domain refinement on any assignment (node): for instance, in the *Domain State* view the user is given information not only on variable domains, but also on active constraints and the refinement type.

The *Incidence Matrix* is easy to understand too, and it is useful for an overall view of the implemented constraint model. This view would be improved with more direct (visible) links towards the source code.

Propagation and *Propagation Events* are more complex than the other views. The latter encompasses so many contents that a lot of experience is required to exploit it fully.

The possibility of using *search_node*/3 in a sharp way (to represent only a portion of the search tree) has been only partially evaluated, because this involves the redefinition of the labeling. This is not always justified, since it perturbs the heuristic.

The global constraint visualizers are easily understandable, even if they presume a good knowledge of the constraint semantics and of the typical problems they are aimed at. However, the standalone utilization of the visualizers (i.e., without the Search Tree tool) is more difficult, since the user must add some machinery to slow down the graphic update during the search in order to follow the evolution.

Expert user considerations

- The Search Tree tool is powerful, especially for the number of accessible details. Even a skilled user can have a direct image of the computation only with the tools.
- The most difficult part of the solving process, hardly analysed with the usual trace debugging, is the constraint awakening that follows an assignment. The data available especially in *Propagation* and *Propagation Events* are quite clarifying, but they are not banal to interpret.
- It is expensive to adorn the code to obtain some particular effect (e.g., a different order, a subset of variables, meaningful labels).
- The initial constraint propagation performed before the labeling is not shown. Sometimes it is useful to know the exact behaviour of Chip in the initial propagation phase.

Generally, the expert user judgement on the tools is extremely positive. There are two main points: the tools allow instant information on the solution search process without requiring a particular experience; the Search Tree tool gives different levels of possible use, where the user can explore and apply new features as experience increases.

4. Evaluation of the PrologIV debugging tools by PrologIA

4.1. Introduction

During this project, PrologIA has mainly developed two relatively independent tools: an assertion tool and a visualization tool. These tools are both available in the last release of the software.

Several applications are certainly in development with Prolog IV, but the difficulty is that most of the times, we don't know these applications: customers don't inform us about what they do with our products.

During the assessment phase, we have tested these tools independently on applications written inside PrologIA. The assertion tool has been tested on the program developed to introduce the assertions in the software (a kind of auto-bootstrap). The visualization tool has been tested on an application developed for the Air Liquide company.

4.2. Assertion tool

4.2.1. Problem description

The goal is here to introduce assertions in the code written to implement assertions themselves into PrologIV. For each compiled rule, this program is called by the PrologIV compiler to try to prove the rule before adding it in the current code. If the rule is not proved, some dynamic checks are added inside the rule. But the important and sensible part of the program called by the compiler is the static prover. This program works locally to each rule (there is no global analysis) like a meta-interpreter checking and adding conditions. For each rule, the following is performed:

1) the store is initialized to empty

2) the preconditions of the rule are added to the store

3) for each call which occurs in the body of the rule, if it is a predefined constraint, it is added to the store, otherwise:

   a) the preconditions of the called predicate are checked (that is to say they are implied by the current state)

   b) the postconditions of the called predicate are added to the store (that is to say we assume the predicate definition is correct)

4) the postconditions of the rule are checked (to be implied).

So, it is this program which is the heart of the prover that we want to consolidate by introducing assertions.

4.2.2. Usage of the assertion tool

Thus, we have done what we should have done before the development of the program (if the tool had existed): we added assertions for the rules involved in a part of the static prover.

Debugging the actual prover

The first step was to create empty assertions for the rules we wanted to prove in a second step. That means to execute the prover on the program without additional conditions. Thus, the rules are checked according to the predefined predicates they use (here, only assertions on predefined predicates are taken into account).
At this step, we had a strange behavior: a rule was found "inconsistent" by the prover. Normally, no difference should have been found with empty assertions. Thus, this behavior denoted a problem in the prover. After classical debugging sessions, it was found that an internal constraint of the prover was executed while it should not be. So, at this stage the tool allowed us to find a bug in the actual prover.

Introducing conditions

Assertions can contain three kinds of conditions:

- meta-properties that now essentially contain some non-basic types, not implemented in the predefined constraints of the software ;

- predefined constraints on types of arguments ;

- other predefined constraints.

In our program, we used these three types of conditions.

Conditions on types

Predefined constraints on types of arguments have been introduced in assertions. These constraints are easy to specify. One reason is that they involve only one argument at a time. Another reason is that it is natural for the programmer to think about types expected (or returned) by its program.

For privileged subdomains of PrologIV (identifiers, real numbers, ...), the proofs are completely performed. For the other domains (e.g. integers), the proof can fail and raises unuseful warnings.

Example:

```
:- rule_prove(Name, Arity) : (identifier(Name), int(Arity)).
:- rule_check(Name, Arity) : (identifier(Name), int(Arity)).

rule_prove(Name, Arity) :-
        rule_check(Name, Arity),
        ....
```

The result of the compilation of this rule is:
"Warning: Precondition N° 2 (int(_233)) of rule 'rule_check/2'
fails while proving 'rule_prove/2' rule number 1.

This behaviour results from a success of the following list of goals:

```
?- int(X), nint(X).
x ~ real
```

while with a privileged subdomain like:

```
?- identifier(X), nidentifier(X).
```

we obtain a failure, thus a success of the proof.

This test can suggest one improvement of the language: creating a privileged subdomain for each different type of argument. This improvement will avoid some unexpected results like mentioned above.

Conditions on meta-properties

At this time, the set of meta-properties is not very rich and has to be improved. Essentially, we can specify groundness property, boundness property (not to be a variable), and the property to be a list of numbers (eventually in an interval). These properties are easy to specify, and are well controlled by the prover. By opposition to constraints on types, this category of conditions would not be obvious to write in the rules (instead of in the assertions) by the programmer in order to be checked while executing the program.
Example:

```
:- rule_verify(NameHead, Arity, Numrule, Rule, Result) : (identifier(NameHead), real(Arity),
        real(Numrule), bound(Rule))    =>   (ground(Result)).
```

In this rule, the argument 'Rule' must be bound (it can contain some variables) when the rule is called while the argument 'Result' must be fully known (ground) after the execution of the rule.

Being independent of the solvers, they are the conditions having the greater part of reliability in the proofs. We can just encourage the idea to increase the set of this kind of constraints (freeness, list of lists, ...).

Constraints as conditions

This is the most difficult part to specify in the assertions: the usage of some predefined constraints. They can obviously involve several linked arguments. Most of the constraints could have been checked directly in the program. But here an advantage of a prover can appear: it is surely better to prove statically that a condition is always true rather than to verify it for each execution. The performances will obviously be increased and a condition which is not always verified will be detected at compile-time, perhaps underlining a bug in this way.

We used some general constraints in the assertions of our program. We principally used interval constraints to specify to a variable to be in a given interval. Usually, the proof can completely be performed when the bounds of the interval are known. Otherwise, we can have a problem with the incompleteness of the interval solver and the proof can raise a (unuseful) warning. This allows to say it is recommended to use these kinds of conditions into the limits of the internal solvers.
Example:

```
:- exec_tail(DeclTail,Name,Arity) : (identifier(Name), Arity ge 0).
```

In this assertion, the argument 'Arity' is constrained to be a (real) number greater than or equal to 0. If the corresponding rule can be proved, that means it is not necessary to check this inequality in the code. If the rule is not proved, the corresponding verification will be automatically added in the rule.

Dynamic checking

This functionality allows to test dynamically conditions that have not been statically proved. We can say that it is a good feature that the generation of dynamic checking is only optional. Some warnings are not useful (because the incompleteness of the solvers), but it is not very boring to see them. It would not be a good thing to have dynamic checks generated in the rules without having the feature to unbranch them (because a lack of performances). If all warnings are pertinent, we can let the dynamic generator doing its work. Otherwise, it is important to do the necessary to suppress not useful warnings or to unbranch the dynamic generator.

Time considerations

Compilation time has been measured at different stages of the introduction of the assertions. According to this stage, the time can increase from five to ten times the compilation time without any assertion. The time spent does not follow a linear chart, and principally depends on:

- the number of constraints involved in the proof,
- the types of constraints used, and essentially the internal solver used to perform the proof. For example, a proof of a rule using internally an interval reduction algorithm can spend a long time for a relatively small number of constraints, while other rules involving several linear constraints can quickly be proved.

Anyway, if assertions are used for the main part of a program (the most sensible part), the factor of compilation time we had observed is not incompatible with a debugging session time and also not incompatible with a final compilation like for building a new version of an application. This factor could be more difficult to accept for a big program where all the rules have assertions.

As mentioned above, we obtain a success with the list of goals:
?- int(X), nint(X).
but this success is obtained after a relatively long time. It is the same for some derived lists of goals like:
?- size(N, X), nint(N).

We obtained this configuration in a proof ('size/2' was used in a rule and the type property 'int' had to be proved for the corresponding argument) : the compilation time was consequent.

This is to confirm that a privileged subdomain has to be created for each different type of argument.

4.2.3. Evaluation

Advantages

The principal advantage in using such a tool during a development is to force the programmer to give a (good) specification of his program and to maintain it. By giving conditions on a predicate, the programmer gives a part of the specifications. When the program is modified, or of course if it is not well written (some errors occur), the rules which do not match any longer with the conditions are signaled. This behavior can appear when the functionality of a predicate is increased while developing a program. For example, some rules are added to handle some cases which were not specified at a first stage of the development.
In this way, non regression tests are automatically performed.

Disadvantages

There are two kinds of disadvantages in assertions:

- disadvantages due to limitations of the tool,
- disadvantages due to the effort asked by the usage of the tool.

The first category could be improved as mentioned in the next section.
The second one can be reduced by the addition of a generator of assertions, as the tool developed by UPM. But there is a minimal set which must be written by the programmer, and this can take a long time. This effort can be compared to the documentation of a program (comments, technical documentation): it is always the work we will do "when it will be possible". It is possible we want to specify assertions only when bugs appear at execution-time. Another alternative is to use other diagnosis techniques such as declarative debugging.

Improvements

We can now suggest some improvements for the prover.
The fist one would be to allow several assertions for a rule (a kind of "or" between assertions). We think it is a real research problem because it is not clear to define the behavior on an error, and principally how to determine the localization of the problem. This determination would certainly involve a big set of tests (several combinations) and a lot of execution time.

The second improvement would consist in limiting warnings (and thus dynamic checking generation) due to the co-usage of constraints handled by different solvers (linear and interval solvers in our case) or the usage of constraints on non privileged types. This means to limit (to forbid ?) the usage of these features in assertions. For the first category (co-usage of solvers), that implies we should use interval constraints in assertions for a rule using interval constraints and vice-versa. Rules using the two solvers remain a problem to be proved.
For non privileged types, perhaps it would be better to consider them as abstract properties to obtain a better behavior on proofs.

The last improvement would be to give the feature of splitting an argument into an assertion to be able to constrain a substructure of that argument. For example, we need to write the assertion to propagate the result that if the predicate *=../2* is used, the first argument of the list will be an identifier:

```
:- =..(X,Y) => Y = [Name | Args], identifier(Name).
```

But this feature will imply the introduction of existential variables in assertions, and these variables must clearly be identified (syntactically) to have different behaviors while used in the proofs.

4.2.4. Conclusion

In conclusion, we can say that the assertions represent a very useful tool that requires a real effort from the programmer. This effort consists in writing the assertions at the beginning of the development and updating them with the modifications of the program. The programmer is not obliged to write assertions for all the rules of the program: for simple parts, it would not be a benefice operation. But if, and only if, this effort is made, especially for complex parts of a program, the average time spent in locating errors can highly be shorter, overall for evaluative maintenance of the program.

4.3. Visualization tool

4.3.1. Testing conditions and goals

The goal of this document is to present the results obtained while testing the visual debugging tools of Prolog IV developed during the DiSCiPl project. These results are presented in two sections:

1. The tests done, the results and benefits obtained, some remarks on the tool used (section " Description of the experience ").
2. The conclusion we can draw from these tests and some recommendations we can formulate on the tool and methodology proposed.

These tests have been done on a particularly well adapted application: the Air Liquide one. This application, or, at least, the part of it we are interested in here, consists in modelling Air gas production plants. The application provides several means to get information from these models, like solution finding (within conditions determined by the user), minimizing variables, reducing variables domains, etc.
The models built for the production plants are represented by sets of numerical constraints, mixing Boolean, Integer and real numbers. Of course, all the functionalities mentioned above are implemented by means of enumeration techniques. In some cases, the built-in Prolog IV enumeration techniques are capable of finding solution quite easily. In some other cases, it is more difficult to get the requested results and some specific search technique has to be implemented.
We have tested the visualization tool in this context, trying to understand the reasons why such or such enumeration technique does not work on a specific part of the problem, and trying to find efficient alternative solution.

The next section gives an overview of the application developed for Air Liquide. It can be remarked that the application is still being developed, even though some parts of it are already operational.

4.3.2. Presentation of the Air Liquide application: a continuous production optimization problem

The company
Air Liquide is the most important industrial gas producer in the World. We focus here in the Air gas Northern Europe production network developed in Belgium, France and The Netherlands. This process consists of separating air in its three main components, Oxygen, Nitrogen and Argon by means of cryogenic distillation. In addition to this, all five plants dispose of resources to liquefy these gases. Since Argon is always produced as a liquid, the result of these processes is a product mix, consisting in:

1. Gaseous Oxygen.
2. Liquid Oxygen.
3. Gaseous Nitrogen.
4. Liquid Nitrogen.

5. Liquid Argon.

The liquid products are stored in tanks and delivered to customers by trucks. The gaseous products are compressed and transported by means of a piping network, connecting the five plants of Dunkerque (F), Mons (B), Antwerpen (B), Charleroi (B) and Seraing (B) and which covers Belgium, the Northwest of France as well as the South of The Netherlands .

For each plant and at each moment, three major decisions have to be taken:

1. The total air flow to be treated.
2. The total amount of gas to be liquefied.
3. The proportion of liquid to be produced as Oxygen, Nitrogen, and Argon.

The needs

The basic need of Air Liquide is very simple. The energy cost over this network represents about 40 MECUs for this relatively (to the company size) small network. The global turn-over of the company being more that 5,000 MECUs, and knowing that the energy costs represents the most important budget, it is easy to understand that this is a key factor for the company. The application goal can be simply defined as building optimal production plans for the entire considered network.

This task is made difficult by the following points:

- The description of a plant is a set of **complex equations** (generally non-linear, eventually discrete) describing the work of all the facilities (compression, liquefaction, etc.)
- The pipeline gas transportation is described through difficult **differential equations**.
- The **energy cost varies** dramatically from one place to another, from one period of the year to another one, from day to night, from working days to week-ends, etc.
- A deep understanding of the work of such production systems is completely impossible for non-specialists, thus implying a very **opened system where the model can be directly accessed by the user's experts**. Only Constraint Programming techniques enable, by definition, this key property.

Technical characterization

The model is completely heterogeneous, non-linear, non-continuous (0-1 values, choices, etc.). It is composed of about 500 constraints to model the entire network, and a short extract from the constraint set is presented here to show its syntax and complexity:

```
PSNG_BP = 1.2309 + 0.0000014719 * FLNG_RES - 0.04016 * MRNG_S2S3
FLNG_RES = FLNG_CYCRE + FLNL_VAP + FLNG_C50 +FLNG_S1 +FLNG_S2S3;
FLNG_CYCRE = IF (MRNG_C30 = 0,0, FLNG_RES- (FLNL_VAP + FLNG_C50 +FLNG_S1
+FLNG_S2S3))
FLNG_S2S3 = IF(MRNG_S2S3 = 2, 7300,IF(MRNG_S2S3 = 1,3650,0))
KWNG_S1 = IF (MRNG_S1 = 1,500,0)
FLNG_S1 = IF (MRNG_S1 = 1, 4800,0)
FLNG_C50 = IF(MRNG_C50 = 1, 9981.2 * PSNG_BP - 46.4577 *(TE00_EXT +273.15) +
10954,0)
KWNG_S2S3 = IF (MRNG_S2S3 = 2 , 1620, IF (MRNG_S2S3 = 1,810 , 0))
CHKWNG = KWNG_C50 + KWNG_S1 + KWNG_S2S3 + KWNR
KWNR = IF (FLNR > 10000,2700,1000)

Q = FLNG_C50 ; Pref = PSNG_RES+2 ; Pasp = PSNG_BP ; T=TE00_EXT + 273.15
Ki = 0.0001031374 * Q * T *log (Pref /Pasp)
Ri = 0.0013694 * T +0.000029198 *Q -0.148
KWNG_C50 = Ki/Ri
```

The application developed

The application developed provides the following features:

1. Open semi-graphic model description based on a graphical box model, and on Excel-like constraints to define the equations. The constraints are here automatically transformed into Prolog IV syntax, so that user can directly handle a more familiar syntax (the Excel one). Several Interactive Graphic User Interface functionalities help in defining this model, including automatic graphical relation views.
2. Testing Features enable to use the model in analysis mode (setting command values and observing results), as well as in synthesis mode (asking to optimize a value: energy consumption, production, cost, or finding a solution in a particular situation).

3. Dynamic Simulation enables to set the network pipelines and liquid consumption among several defined periods, and ask the system to optimize the production plans for each plant, under stock constraints. It is also possible to tell the system that some specific equipment is not available in certain periods, to fix long-term stock policies, etc.

In this visualization tool testing procedure, we have focused our attention on the second point, the " testing facilities ", involving many enumeration techniques.

4.3.3. Description of the experience

The experience done consists in improving a difficult optimization procedure over one of the production plants models. We can first remark that this optimization problem is similar to a solution finding problem. In fact, knowing some of the problem properties and the fact that we are dealing with an industrial problem, in which finding an approximate optimal solution is highly satisfactory, we can be sure that if we can find a solution under any specific domain restrictions, we will then be able to find the requested optimal one.

We will here distinguish two ways for exploiting the visualization tool:

1. The micro-analysis. It consists in observing the execution tree, node by node, basically using the " Name " option on, to see exactly what happens during the execution. In this context, to our opinion, the main use of the visualization tool is to debug abnormal program behaviors, and it can be used to debug Prolog programs with or without constraints, with or without fundamental backtracking. I have used this aspect of the tool when the answer given by Prolog IV was not the one expected, while trying new search strategies. This usage of the tool complements the classical box model debugger.
2. The macro-analysis. Contrary to the first impression we get using the tool, the most important and interesting part of it can be the analysis of the search tree shape, without looking into the details of it, the name of the predicates, etc. It gives the user the possibility to have an overview of the execution tree shape. Is it balanced ? Is it deep or wide ? Is it homogeneous ? etc. We can remark that the search tree we want to study is a little bit hidden by all the Prolog things not interesting us in this exercise (we focus here on the search tree: we work with CP more than CLP or LP). Nevertheless, this problem can be solved by the possibility of colouring some particular nodes of the tree. Unfortunately, this functionality is not available during the execution of the program, but only after.

We have made our experience on a difficult, large-scale problem, for which it is difficult to find solutions (one of the most complex production plants considered in the Air Liquide application). The targeted constraint system has several hundreds of constraints and about one hundred variables.

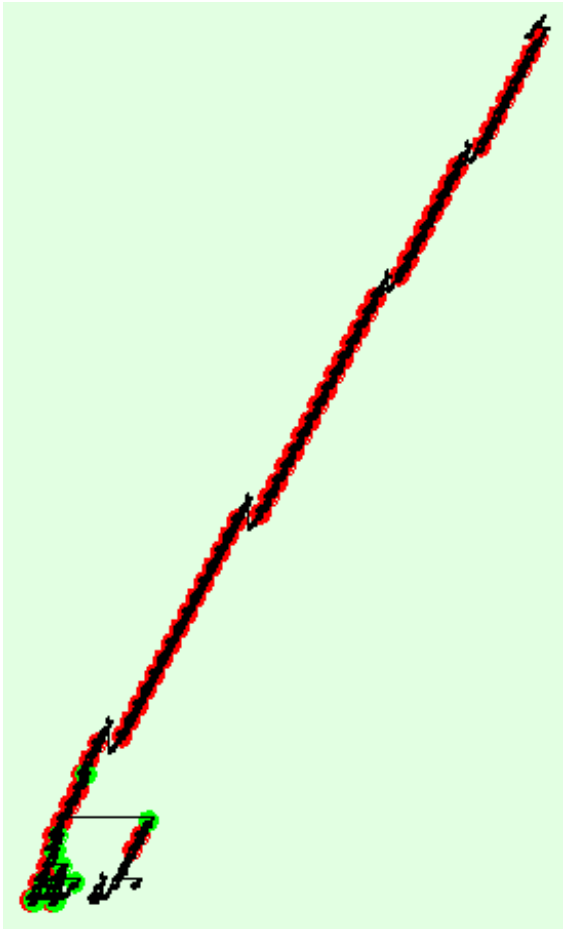
Let's enumerate the problem variables with a classical LIST_ORDER search method (splitting the variables as indicated in the splitting list). This trivial enumeration method code is as follows:

```
mon_enum([]).
mon_enum([X|L]):-
   split_var(X),
   mon_enum(L).

split_var(X):-
   float_size(X,le(100000)),true,!.
split_var(X):-
   milieu(X,M),
   decouper(X,M),
   split_var(X).

decouper(X,M):-
   le(X,M).
decouper(X,M):-
   gt(X,M).
```

```
milieu(X,M):-
   bounds(X,Min,Max),
   float_rank(Min,RMin),
   float_rank(Max,RMax),
   float_rank(M,floor((RMin.+.RMax)./.2)).
```

It basically consists in splitting the variable domains, respecting the list order, as long as their domain size, expressed in number of atomic intervals, is greater than 100,000, an already quite good precision. The domains are split in such a way that the left part and the right part have the same number of atomic intervals. This is the best theoretical choice since it gives the best algorithmic execution bound.

We get then the following search tree (incomplete):



(Failure at level 57)

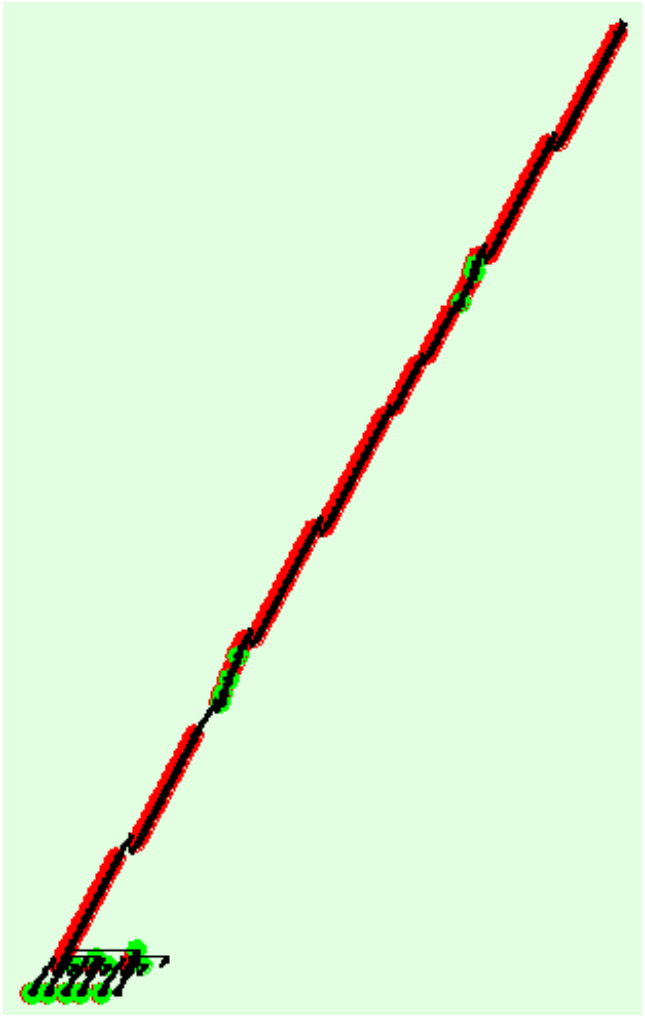
The red points indicate that the left part of the interval is selected, and the green ones indicate that the right part has been selected. " Failure at level 57 " means that the failure points - that are all at the same level - are at the level 57. If the whole tree has the same deepness, it means that there would have been 2^{57} nodes to explore it.

As you can see, it is completely impossible to see, at this zoom level, the details of the tree. Nevertheless, we can deduce from it some interesting information:

As we can see at the beginning of search tree (the search has been interrupted without finding a solution), it is very deep and very regular. The first thing that can be deduced (pragmatic deductions and not theoretical ones, as in all this document) is that the search tree is as large that it is not possible to explore it entirely (deepness).

Another thing, is that the tree is very regular with all failure nodes at the same level. The bad new is that it can let us suppose that the tree will go on being regular, and as deep as in this first phase, thus confirming that it is very big. The second information we get from it, is that the failure is always obtained splitting the same variable. One idea can be then to use this variable earlier in the enumeration process.

If we try to enumerate first the failure variables of our first example, we get the following search tree:



(Failure at level 101)

As it can be seen easily, the obtained result is not better than the previous one (even worse) in terms of deepness, but the early green points show that some failures have arisen earlier in the tree. Nevertheless, studying the variables enumerated in this search tree, we can see that the system needs to enumerate in fact the same variables.
It is impossible to conclude saying that one method is better than another one since one tree is deeper, but the other one has early failures. In both cases, it seems not probable that we will get a solution in the reasonable time.

The analysis of these first trials, including the information given by the graphic tree visualisation tool, but also including other information, indicates that there are two major possible causes explaining the difficulties we have in finding a solution to this problem:

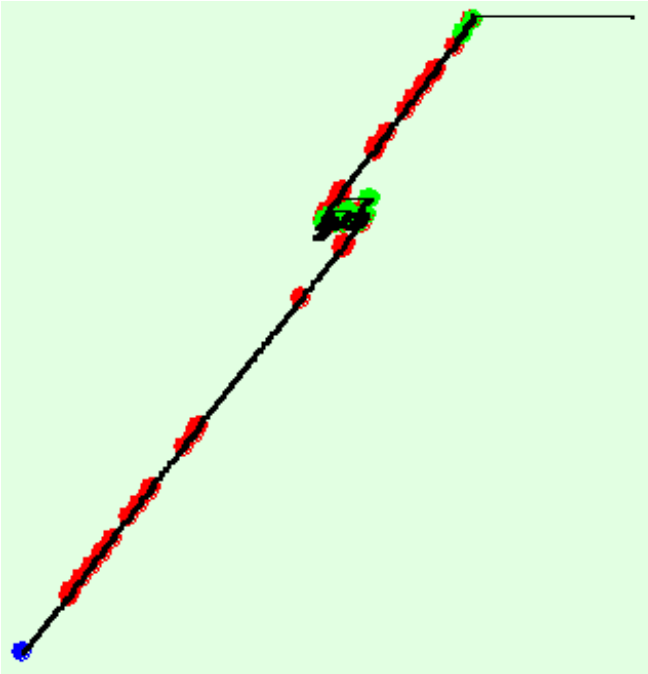
1. The failure level is too deep. The conclusion we can draw from that point is that the reduction power on each node of the search tree is too weak. We can also expect that, as demonstrated in the previous trial, the order of enumeration of the problem variable is not the right one. Among other things, we can remark that the Boolean variable of the problem are never enumerated.

2. The variable domain splitting does not seem to be adapted to the problem. In fact, splitting the domains in two sub-domains having the same number of atomic intervals can lead to enumeration problems when some of the variable, that can have great values and are in fact enumerated close to zero. This is due to the structure of the IEEE 754 floating point numbers that are used inside the solver, and that are highly dense close to zero.

Let's make another trial, trying to cope with the previously enumerated problems:

1. In order to cope with the problem of reduction power, we will add, at each step of the enumeration, *i.e.* on each node of the enumeration tree, a complementary algorithm of bound reduction. The kind of algorithm is described for instance in [Consistency techniques for numeric CSP's, O. Lhomme. proceedings of IJCAI, 1993 ; or Problèmes continus en PLC avec les intervalles, S. N'Dong, M. van Caneghem, IV^{èmes} Journées Francophones de Programmation en Logique (JFPLC'95), Teknea, 1995.]. The cost of such an algorithm is obviously greater than the simple fix point one, but it can bring additional power to the reduction algorithm.
2. We will now place first in the enumeration list the Boolean variables.
3. In order to cope with the problem of the precision and domain splitting, we will now cut the variable by their arithmetical centre, and we will select a precision adapted to each variable of the problem, *i.e.* a significant precision compared to the variable meaning and possible values.

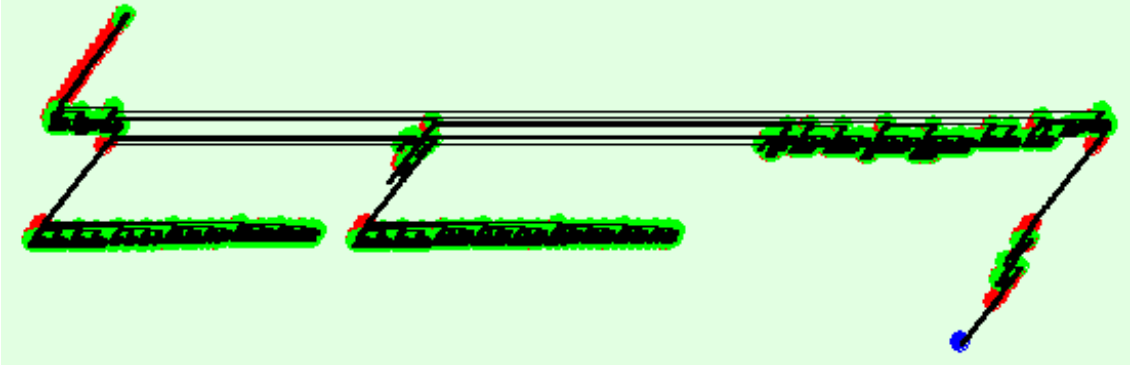
Applying these modifications to our program, we get the following search tree:



(1,388 nodes in the execution tree)

As it can be seen on the execution tree (the blue point corresponds to a solution), a solution is now found within a reasonable time, and, more important, with a search tree having a reasonable size. This program now can find contradictions quite early, and converges to a solution in a quite straightforward manner.

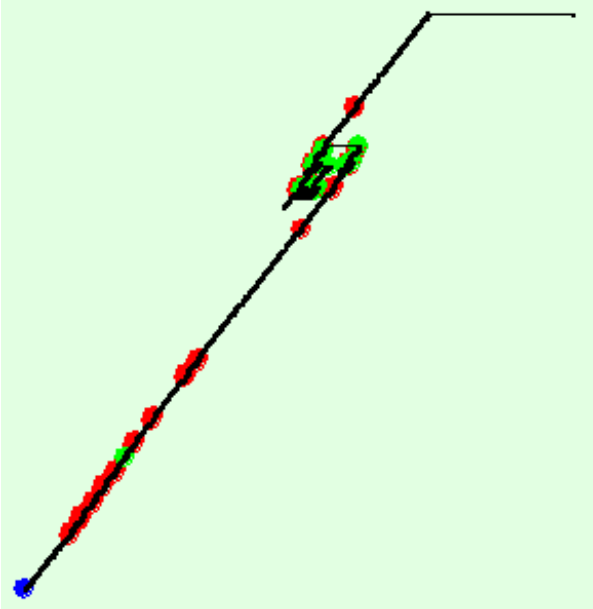
If we try to get more precise solutions (the previous tree corresponds to a quite imprecise solving process), we get the following search tree:



(more than 16,305 nodes in the execution tree)

As it can be seen here, it is quite difficult to obtain a more precise solution (The tree is more than ten times bigger than the previous one). The form of the execution tree suggests, one more time, that the order of the enumeration variable list can be refined, and that we can obtain even better results.

But we can also use the previously obtained rough results as a start point for the refined search. In this case, we get the following execution tree:



(1,270 nodes in the execution tree)

We can see here that we are able to get a refined solution quite easily, compared to the previous attempt, even if we have to bear in mind that we have made two operations to get there. The sum of the nodes in these two operations is 2,658, compared to the 16,305 nodes that were needed to get directly the refined result. We can remark that this enumeration procedures chaining, with a more and more important precision, is an indirect way to implement a partial GREATEST_DOMAIN search method (enumerating first the less constrained variables, *i.e.* the variables having the largest domains). We can conclude from this experience that an efficient enumeration strategy would probably consists in melting the LIST_ORDER approach to the GREATEST_DOMAIN approach.

We will stop here the description of our experience, but we think that they have shown that the search tree visualisation can be one of the tools helping in the definition and refinement of the enumeration strategies necessary to get the best of domain reduction based solving systems.

4.3.4. Conclusion and recommendations

While using solving algorithms based on domain reduction techniques (Finite domains, intervals, etc.), one of the main problems we have to face is the enumeration procedure.
Basically, this kind of solving methodology is not complete, and we have to apply a divide-and-conquer strategy to refine the proposed results.

Rationally implementing this kind of technique on various problems is a difficult job, often based on the experience. Some classical techniques are known to be efficient on specific problem classes, but there exists no global methodology to find the right technique for a given set of constraints.

The execution tree visualisation tool, implemented in the new Prolog IV version, aims at helping the programmer in this difficult task. The basic idea of the tool is to give the user a graphic view of the execution tree. Having a view of the search tree is, in some sense, the dream of any enumeration technique programmer.

In order to test and to evaluate this new feature, we have tried to solve a real-life, hard, and still unsolved (by us!) problem. We have concatenated the information given by this new tool to the information we can have from other tools or techniques. Finally, we have found a way to get good solution from this difficult problem, in a reasonable time, and with a quite small search tree. Therefore, we can conclude that this kind of tool can be very useful.

Another important issue of this study is that the use of such a tool can be divided in two classes:

1. Using it on small example, taking all the tree details into account. In this context, the tool seems to be easy to approach and use, and will mainly be helpful to debug programs or as a pedagogic tool.
2. Using it to help improving search strategies on big problems, looking at the global behaviour of the tested enumeration strategies. In this context, it seems to us that the tool needs a deep understanding of Prolog, and of the Constraint Programming technology to be used successfully.

Concerning the tool functionalities, the nodes colouring is very interesting and useful. It can be associated to the possibility that is given to have at the same time:

- code compiled in debug mode thus providing information in the debugger and in the execution visualisation tool
- code compiled in no-debug mode.

Combining the two, it is quite easily to show the necessary information and to hide the uninteresting processes.

The 3D-representation of the execution tree is quite surprising at the beginning, but is easy to adapt to. It makes the trees as readable as possible (we know this is not a trivial problem). The main difficulty concerning the tree readability is that two nodes at the same visual level can have in fact very different logic levels, and it can be disturbing while analysing the tree morphology.

Some details would have been appreciated, like the capability to use the pre-defined enumeration strategies and to see the resulting execution trees. In the current version, this is not possible because these enumeration predicates are pre-compiled in a specific mode that is not compatible with the debug mode. Another interesting thing would be to be able to see the variables and their states on the tree visualisation, in order to be able to analyse what are the choices that are made during the enumeration process.

Finally, we can regret that the speed of the whole system is not sufficient to run bug search instances. This is compensated by the great robustness of the system, which can handle execution trees with more than 20,000 nodes without difficulty.

Finally, we will conclude saying that, to our opinion, this kind of tool is definitively useful for Constraint Programming technology users. This very first version of such a generic tool is very encouraging, and will become even more useful and powerful when some additional features will be added to it.

5. Final Considerations

5.1. End users

Novice users and expert users have tested the tools. The novice users have a good Prolog background and experience in modeling combinatorial problems. The expert users have several years of CP programming experience in addition.
No people have been involved for whom "combinatorial" programming was completely new. However the expert users noticed that giving demonstrations and/or introductions of CP programming to completely novice users, becomes easier using the graphical debugging tools.

5.2. Applications

All end users started using the debugging tools on academic examples and course exercises. Thereafter the end users experimented with existing and new industrial applications. A short overview of the applications and tools used can be found in the first section of every evaluation chapter, a detailed description in the subsequent sections. There is no overlap between the applications treated by the different end users. Also, the different end users focused on different parts of the debugging tools: ICON concentrates more on the CHIP graphical propagation views, OM Partners concentrates more on the CHIP global constraint visualizers. PrologIA tested the PrologIV debuggers.

5.3. Need for a debugging methodology

For the tool developers the results obtained with industrial applications are certainly most interesting. However the end users did include in the report the experiments with academic examples, since after all, a lot of time has been spent learning how to interpret the information shown by the visualization tools. Although it is rather easy to get the graphical debugging tools working, the end users strongly recommend the development of a "graphical debugging methodology". This debugging methodology could help the end user to exploit the visual information in the best way: which conclusions can be deduced immediately from the search tree geometry and/or domain behavior? What views are most interesting when going into more detail? Which functionality still has to be discovered?

5.4. Program amelioration due to the use of debugging tools

Before starting the assessment phase, it has been suggested to evaluate the debugging tools by using statistics on the number of errors, the developing time, etc. when programming with/without debugging tools. For several reasons this was not feasible:
First, given the time and resource limits of the project, industrial partners could not afford to train people or to develop new applications without using the best and most advanced tools.
Next, since the graphical debugging tools only can be used when a program already runs, the debugging support for a large part of the program development has not changed. We still have:

- The typical syntactic and semantic errors.
  In general, they can be found using classical debugging techniques like text tracing. Here, the assertion tools could give assistance.
- The errors due to data inconsistency.
  These inconsistencies do often provoke the violation of constraints and the abrupt stop of the CP program (the famous 'no' answer). The detection of these data inconsistencies is very time consuming and can in general not be done by simple text tracing.

Consequently, objective before/after comparisons were only possible when using the debugging tools in existing applications and looking to areas where the graphical debugging tools provoke assistance, for example:

- the behavior and efficiency of the labeling/optimization routine,
- the efficiency of the constraint propagation,
- the effect of adding redundant constraints to the model.

Well, the end users have "to admit" that all existing industrial applications for which the debugging tools have been used, have been improved (see detailed application descriptions of OM Partners, ICON and PrologIA). In most cases the labeling routine and/or propagation have been improved and result in a performance amelioration and/or better optimization. However, the amount of treated applications is not sufficient for statistics.

5.5. General conclusions on the CHIP graphical debugging tools

The graphical representation of CLP problems through the search tree tool and global visualizers is what a user needs to understand and explore different search strategies. They encourage the user to optimize the search strategy and to go farther than the first feasible solution.

Although most real-life problems still have to be scaled when using the debugging tools, the number of variables and constraints that can be used is big enough to test valuable industrial cases. As a result, new applications are written in such a way that the program can be started with/without debugging tools. The expert users are convinced using the debugging tools will make the maintenance of industrial applications easier and less time consuming.

5.5.1. Search tree tool

The overall view (shape of the search tree) in the search tree tool immediately gives an intuitive picture of how well the search strategy works (what is the degree of propagation). Afterwards, the user can inspect the execution in more detail, by clicking on the nodes of interest (comparing different nodes) and investigating the different views. The search tree tool allows complete navigation through the tree, not just between adjacent nodes. Novice users reported that the constraint paradigm became immediately clear, expert users realized that a lot of difficult text tracing work, trying to understand what was happening, will be avoided in the future.

For all users, the domain state and update views in the search tree tool are quite easy to understand but working out the propagation view requires some more effort. The propagation event view is even less accessible for an end-user, but may no doubt provide essential information to the implementers of the system.

Some suggestions for an easier and better search node annotation are given below. Keeping track of the search node number requires minimal effort when using CHIP objects (in that case the number is just an extra field in the object structure); it's not necessary to extend or build data structures to link the numbers with the variables. However, especially when dealing with large problems, the user should have the possibility to define an application specific node labeling.

5.5.2. Global constraint visualizers

The global constraint visualizers offer graphical representations that are tailored to the specific problem at hand (to a specific use of the constraint). The fact that they can be used in combination with the search tree tool results in a powerful tool set.

Adapting the program to use the global visualizers was straightforward, as it only involves adding a simple wrapper around the global constraints. Combining the global constraint visualizers with the

search tree tool has the advantage that the visualizers are automatically updated when navigating through the search tree. However, it also requires that *all* variables determining items (e.g. rectangles) in the visualizers are included as search tree nodes. These extra search nodes complicate the views in the search tool, mixing up the relevant information with irrelevant details.
Standalone use of the visualizers may provide more information than in combination with the search tree tool, e.g. if the predicate *min_max* is involved. The reason is that the search tree tool does not remember constraints introduced during the search.

The GUI's of the global visualizers often come very close to dedicated application GUI's.
Although it is not the intention to compete with specialized GUI's, it should be possible to extract more user defined information from the global visualizers. Suggestions for amelioration can be found below.

5.5.3. Suggestions for extensions to the CHIP debugging tools

Beside the correction of some bugs, the following extensions are suggested.

User definable annotation

In the current tools, annotation of the search nodes and items in the visualizers is predefined. Especially when dealing with large problems, this information is quite cryptic for the user. It is difficult to relate the annotation with the problem concepts (e.g. tasks to be scheduled). Moreover, there is no uniform notation (numbering) of variables/objects over the different visualizers and search tree views. Hence, clicking on some item (rectangle) in a particular visualizer highlights items with corresponding numbers in other visualizers, but these do not necessarily correspond with the same task.

A possible extension is to allow user-defined annotation. In this way, the user can specify information that is relevant for the specific problem at hand:

- In the search tree tool, the *search_node* predicate could be extended such that the user can specify the node label, e.g. *search_node(X,N,'task3-product1',indomain(X))*. An extra option "Show User Label" in the search tool could then annotate the search node with this label.
- The *visualize* wrappers could also be extended to allow user-defined annotation of the items in the visualizers. E.g. an extra argument could be the list of labels corresponding to the list of rectangles in the diffn, or the list of variables (starts, durations, ends,..) in the the cumulative.

A further extension would be user-defined coloring of search nodes or visualizer items based on some characteristic of the underlying problem concept (e.g. based on task type).

Based on these annotations the user could identify the search tree nodes that are interesting according to a certain criterion, and consequently decide which part of the tree to observe.

Tree information

- It would be interesting if the user could easily identify on the tree some areas sharing common characteristics, for instance big domain reductions, isomorphic subtrees, unaccounted problem symmetries, etc. The availability of statistics could be quite helpful to let the user compare the quality of different strategies.
- Another possible improvement is to highlight/color the nodes that (do not) satisfy specific conditions. For instance, during performance debugging, the user should be able to quickly identify *min_max* failure nodes (i.e., solutions worse than a previous one) in order to refine the heuristic. With the present tool, the user must select any failure node and find out if the cause of failure is interesting.
- No information is provided about shallow backtracking, i.e. backtracking within a search tree node. Such information is also relevant when comparing different labeling strategies and their degree of propagation. A proposal is to display the number of backtracks within each search tree node.
- Constraints introduced during the search are not remembered by the search tree tool (e.g. the extra constraint introduced by *min_max* after a first solution has been found). This results in a wrong image in the domain state view (the domain may actually be smaller than shown). The correct domain can only be seen in the indication of the domains in the search nodes.

View manipulation utilities

The user would be helped during both correctness and performance debugging if the tool gave some assistance in the search tree exploration. Some possible improvements could be the following ones:

- Collapsing the nodes that represent variables becoming ground because of propagation.
  Actually, a typical event is the reduction of variable domain to a single value, due only to constraint propagation following some assignment. This fact is translated on the search tree into node chains where there is no additional domain reduction. The graphical representation of this property would avoid that the user analyses these nodes, which do not carry any extra information.
- Defining variable order in search tree views.
  It would be helpful to let the user change the order of variables in views, in order to have a better visual representation for certain problems without tweaking with the *search_number* adornment. More generally, it would be advisable to simplify the search tree tool wrapper, in order to avoid the reordering of traced variables inside the code (see above). Another solution could be to allow the user to dynamically change the visualization and ordering of the elements inside the tool with mouse actions, for instance by means of drag-and-drop of rows and columns, or with pop-up menus.

New views

- User definable representations (based on existing view dimensions)
  The 2D representation of domains is suitable. As a possible extension, we suggest a more general framework where the user defines his/her own representations.
  The idea is that if the tool provides some "dimensions" (e.g., list of the variables, list of the domains, constraints, propagation, etc.), the user could select as horizontal or vertical axis the preferred items, possibly selecting subsets or organizing them in parallel layers, in order to represent/visualize also multidimensional problems.
- A utility to compare graphical views pertaining to different nodes.
  Very often during the analysis it is important to understand the ongoing domain reduction. In the current tools, the user selects in turn two different nodes in order to depict the changes. It would be better to have a "diff" utility that represents graphically the differences in a view between two nodes (e.g., a *Diff Domain State* view).
- A utility to give information about the change in variable ordering in different branches.
  The user had problems to find out the changes in variable choice in the different branches of the tree. The reorganization of the search can be very important in case of optimization. A new view (the phase line display) covering this aspect, is described in a recent Cosytec paper.[22]

Extension of the global constraint visualizers

- Standalone visualizers: breakpoint with user interaction
  During standalone use of the visualizers, the user should have the possibility to insert a breakpoint upon which some action can be taken. E.g. it would be nice if the user could ask for additional information, such as the domain of a variable, etc. Currently, one has to decide in advance which information has to be displayed. No interaction / extension is possible once program execution has started.
- 3D diffn_visualizer.
  A visualizer of 3D problems could help in case of bin packing problems.
- Better scrolling and zooming functionality.

5.6. General conclusions on the PrologIA debugging tools

Since the PrologIV debugging tools have only been tested by the PrologIA users, the conclusions of the appropriate chapter are not repeated here.

[22] Ship loading – An example of Contradictions, by H.Simonis (Cosytec SA).

The conclusions on the prologIV assertion debugger are globally less positive than on the graphical tools. However the principal advantage mentioned, i.e. the fact that the programmer is forced to give a good specification of the program, can not be neglected, and make the CHIP users eager to use CHIPRE.
The conclusions on the PrologIV graphical debugger are comparable to the conclusions on the CHIP search tree tool. Certainly some of the suggestions for extension of the CHIP tools are also valuable for the PrologIV tool developers.

References

[1] The CHIPRE System: A User's Manual, Facultad de Informatica, UPM, 1999, May, CLIP1/99.0, Technical Report

[2] The Logic Programming Paradigm: a 25 Year Perspective, Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging, written by M.Hermenegildo, G.Puebla, F.Bueno, edited by KR.Apt, V.Marek, M.Truszczynski and D.S.Warren, Springer-Verlag, 1999, LNAI, pages 161..192, In press.