Debugging Constraint Programs

Micha Meier

ECRC- ECRC-95-15

technical report ECRC- ECRC-95-15

Debugging Constraint Programs

Micha Meier

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : micha@ecrc.de

Abstract

Constraint programming (CP) is in its substance non-algorithmic programming, not last because it is often being applied to problems for which no efficient algorithms exist. A not immediately obvious consequence of this fact is that debugging CP programs is principally different from debugging algorithmic programs, including imperative, functional or Prolog programs. It is also more difficult. Moreover, it is frequently necessary to apply *performance debugging* to CP programs, which are correct but too slow to be feasible. The whole area of CP debugging is still lacking both methodology and tools to support users in improving their programs.

In this paper, we present a paradigm for tracing constraint programs and the design and implementation of **Grace**, a graphical environment for tracing CLP(FD) programs on top of **ECL^{i}PS^e**.

1 Introduction

Developing CLP applications is a difficult task. This is to a large extent due to the fact that CLP is usually being applied to combinatorial search problems for which no efficient algorithms are known. Instead of following some well-known rules, the users have to experiment with various models, approaches and strategies to solve the problem. Apart from handling the usual *correctness debugging* of their programs, users are also facing the problem of *performance debugging*, i.e. finding a way to execute CLP programs efficiently. To date, there exists no satisfactory methodology for debugging CLP programs. There are basically two ways to approach the problem: either try to apply all available methods exhaustively, last resort being simplification and downscaling of the problem, or to analyse the behaviour of the program and try to understand what is the reason for its poor performance. The current CLP systems unfortunately offer little support for this task and there are no widely available tools which would support tracing and (performance) debugging CLP programs.

Our goal is to contribute to the methodology of performance debugging of constraint programs and to develop an environment to support it. Although much of what we present here applies to the whole CP area, our primary target is performance debugging of CLP(FD) programs and more specifically, programs based on labeling finite domain variables and backtracking search.

In this paper, we first analyse the features of debugging constraint programs as opposed to algorithmic ones and draw the conclusions about the features of a debugging system. Next we present the functionality of the **Grace** system and show some examples of its use. We also discuss implementation issues and show how an advanced debugging environment can be implemented on top of **ECL^{i}PS^e**, using its control features. Finally we discuss the possible extensions of the environment both in the CLP(FD) area and in other areas of constraint programming.

2 Debugging Constraint Programs

To support debugging of constraint programs, we first have to analyse the main features of CP as opposed to more conventional, algorithmic programs.

2.1 Features of Constraint Programs

The constraint programming paradigm is inherently different from imperative, functional or Prolog-type logic programming, since it is *non-algorithmic*. Even in Prolog (without delays), and more so in conventional languages, the

program execution follows a fixed scheme which implements a particular algorithm, no matter if the program is declarative or not. CP however, only states constraints and then looks for a solution that satisfies them. The search is rather data-driven than program-driven.

A consequence of these facts is that debugging CP programs is inherently different from debugging algorithmic programs:

- Debugging algorithmic programs is itself also algorithmic, it can follow a particular debugging algorithm, which guarantees success. Even the debugging itself can be automated.
- It is *local*, each program piece, e.g. a function, can be considered and debugged separately and without the execution history.
- It is possible to decide at any execution point if the current state is correct or not.
- Source-level (e.g. for C) or invocation-based (Byrd box for Prolog) debuggers are well suitable for this kind of debugging.

On the other hand:

- CP debugging, and especially performance CP debugging is not algorithmic and it can hardly be automated.
- It is mostly *global*, it is necessary to consider the whole program and also the execution history.
- Especially for performance debugging it is not possible to decide if a particular execution state is correct or not. The execution state is a point in the search space and we cannot decide if the solution will be quickly found or not.
- Similarly to the search for a solution which is performed by the program, the debugging itself is also a search problem. The debugger to support it must therefore be highly interactive and open. Debugging paradigms for algorithmic languages are not suitable for constraint debugging, it makes little sense to extend source-level or Byrd-box debuggers for constraints.

2.2 Approaches to Debugging

We can divide the debugging approaches into *experimental* and *analytic* ones. A debugging environment must support both of them in a satisfactory way.

2.2.1 Experimental Debugging

Experimental debugging does not necessarily assume a deep knowledge of the methods used. All that is needed is a large repository of available methods with which the user can experiment. This approach is quite appropriate for non-expert users or for the first estimation of a given problem and it is of course applicable only to performance debugging.

When we consider the particular area of CLP(FD), we can divide a program into three main parts:

1. The *model*, which, among others, specifies all variables, their domains and their interpretation. It also specifies the conceptual constraints imposed on the variables.
2. The actual *constraints* which are used to express the properties of the solution in the particular model and system.
3. The *labeling strategy* which is used to search for the solution(s). This includes both selection of variables to label and value ordering in the selected variable domains.

Experiments on each of these items should be well supported by the debugging environment:

1. While a debugging environment cannot directly support the user in modelling the problem and switching from one model to another, it should at least be able to compare different models, especially to compare two programs running concurrently with different models.
2. The debugging environment must be able to show the effect of using particular constraints, the amount of propagation achieved, and it also has to allow dynamic adding of new or redundant constraints.
3. The environment should have a large repository of available strategies and heuristics which are easy to use and to combine. Interestingly, CLP(FD) can be used as a unifying framework for various other CP approaches based on finite domains, e.g. local repair or statistical methods and an ideal debugging environment would allow to experiment with various such methods to look for the most suitable combination for the particular application.

2.2.2 Analytic Debugging

Analytic debugging assumes a more thorough knowledge of the mechanisms that are involved in the constrained search and it is the only way to perform

correctness debugging. An appropriate debugging environment allows to filter and structure the tracing data at various conceptual levels, so that the user is able to infer new information and use it to improve the performance. Some typical analytic debugging approaches are:

- **Looking for redundant constraints.** In most CLP(FD) systems each constraint is considered separately, the only constraint which is used in combination with others is the variable domain itself. It may often happen that some information is not encoded in the constraints because it seems obvious. For instance, from

  ```
  X + Y = 10
  X + Y + Z = 12
  ```

  a CLP(FD) system is not able to deduce that Z = 2. A debugging environment must make it as easy as possible to spot similar cases.

- **Developing new labeling strategies.** When the search path is being displayed in a suitable way, the user may be able to see the reasons for not finding the solution quickly or for not finding a good solution first.

- **Finding the appropriate propagation amount.** A modification of one variable's domain triggers some of the constraints that are attached to this variable. Finding exactly the right amount of constraints to trigger is an important part of performance debugging. If some of the woken constraints make no propagation or only an insignificant one, they only slow down the execution. On the other hand, if the domain update does not trigger enough constraints, the pruning will be insufficient and the search space remains too large.

- **Finding the reason for a failure.** This is important both for correctness and performance debugging. For correctness debugging it helps to find the reason for not finding the correct solution. For performance debugging it helps to find out how some branches in the search space are pruned away. As a result, the user may realise that the reason for pruning is different than expected and that e.g. some constraints which were expected to cause the pruning were not triggered.

- **Finding the reason for a wrong solution.** If the program is about to consider a wrong solution, some of the constraints are wrong or missing. A debugger must assist in fixing this problem.

2.3 Support from the Debugging Environment

The support expected from a CLP(FD) debugging environment can be seen from several different angles.

2.3.1 Display

The displayed data is obviously very important for analytic debugging. The debugger must display all necessary information, but it must also apply *display economy*: the information to display is potentially very complex and large and thus the display must contain only the very necessary data so that more information can fit on the screen. In contrast, see the output of the **ECL^{i}PS^e** debugger in a CLP(FD) program:

S (61396) 20 RESUME qeq(0, 2575, T_g71094{[0..2575]}) (dbg)?- creep
S (61396) 20 EXIT qeq(0, 2575, 0) (dbg)?- creep
(65874) 20 RESUME 3000 - C_g517524{[19..74]} - C_g518588{[0..560]} -
C_g519550{[20..36]}-0 - C_g523442{[22..35]} - C_g524476{[15..33]} -
C_g525540{[9..39]} - C_g526604{[0..510]} - C_g527566{[0..650]} -
C_g528528{[15..70]} - C_g529592{[6..45]} - C_g530634{[8..49]} - C_g531698{[10..61]}
- C_g532762{[0..680]} - ... #>=0 (dbg)?- creep
(65878) 20 DELAY 3000 - C_g517524{[19..74]} - C_g518588{[0..560]} -
C_g519550{[20..36]} - C_g523442{[22..35]} - C_g524476{[15..33]} -
C_g525540{[9..39]} - C_g526604{[0..510]} - C_g527566{[0..650]} -
C_g528528{[15..70]} - C_g529592{[6..45]} - C_g530634{[8..49]} - C_g531698{[10..61]}
- C_g532762{[0..680]} - ... #>=0
(65874) 20 EXIT 3000 - C_g517524{[19..74]} - C_g518588{[0..560]} -
C_g519550{[20..36]} - C_g523442{[22..35]} - C_g524476{[15..33]} -
C_g525540{[9..39]} - C_g526604{[0..510]} - C_g527566{[0..650]} -
C_g528528{[15..70]} - C_g529592{[6..45]} - C_g530634{[8..49]} - C_g531698{[10..61]}
- C_g532762{[0..680]} - ... #>=0 (dbg)?- creep
(65877) 19 EXIT indomain(0) (dbg)?- creep
(65879) 19 CALL instantiate([D_g5234{[0..10]}, D_g1544{[0..10]}, D_g7284{[0..10]},
D_g4824{[0..10]}, D_g7694{[0..10]}, D_g8514{[0..10]}, D_g8924{[0..10]},
D_g9334{[0..10]}, D_g9744{[0, 1]}, D_g10154{[0, 1]}]) (dbg) ?-

Due to the size of problems we are aiming at, it might not be feasible to display the whole constraint network like e.g. [11]; even displaying parts of the network may not necessarily help the user. On the other hand, displaying all of the problem variables is necessary. In case that some data is not or can not be displayed by default, it must be possible to display it on demand.

The debugging environment needs both fixed and user-definable displays. The fixed ones give a structure to the display and to the debugging process itself, the user-definable displays are added to support a particular user and a particular application.

The analytic debugging needs data from different conceptual levels so that more global or more detailed information about the application becomes visible:

- **Solution.** Displays only the solution, its cost, search time, number of backtracks etc. This is also suitable for methods which are not based on backtracking search or which are available only as a black box.

- **Search path.** Displays the path in the search space that leads to the solution. For CLP(FD) this is the sequence of labelled variables and their values.

- **Selected data.** At this level the value of selected variables, expressions and terms is displayed on each step. The expressions must be user-definable, e.g. the current cost estimate, the current value of some particular constraints, number of constraints which are already satisfied and those that are not etc.

- **All data.** Displays the current value of all involved variables.

- **Domain updates.** The sequence of domain updates which are caused by the last labeling step is displayed, either step by step or together as a sequence of events.

- **Constraint propagation.** At this most detailed level the sequence of constraints propagation becomes visible. The environment shows which constraints were triggered and re-evaluated by the most recent labeling step and what was the result of their re-evaluation.

2.3.2 Interaction

The debugging environment must be highly flexible and interactive. Most of the user actions should be mouse-oriented, choices supported by buttons and menus. Usual debugger commands like single-stepping through labeling steps, skipping, jumping, retrying, setting breakpoints etc. would be the main vehicle for analytic debugging, choices and comparisons of various strategies would be used for the experimental one. Changing to other conceptual display levels and dynamically modifying the displayed information has to be made easy.

2.3.3 Integration

Ideally, the debugging environment would be another process which would control the user application, run and stop it and extract data from it. This is however possible only with a very low-level programming approach and it is completely machine-dependent. On the other hand, the debugging environment which we expect to be most useful has different properties:

- It is *open*, the user can extend it, define new primitives and use new debugging approaches.

- It is easily maintainable and programmed in a high-level language, possibly in CLP(FD) itself.

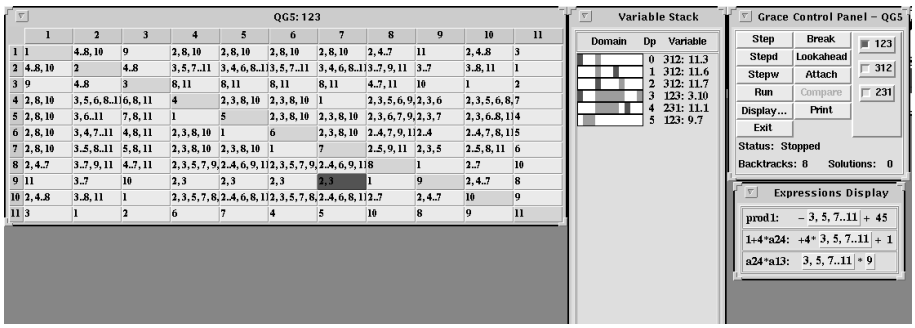
- It consists of two basic layers: the environment itself is written on top of basic building blocks which allow to access, display and modify data and which are available to the user for extensions.

- Since performance debugging usually handles long programs, the execution overhead caused by debugging must be minimal or, because displays slow down the execution in any case, at least it should depend only on the amount of data displayed.

  Particularly, it must be possible to place breakpoints on variable changes. This is feasible only when the debugger is tightly connected with the object program. Using another process (like in *dbx*) makes efficient variable breakpoints very difficult.

3 Grace Features

The above considerations have led us to the implementation of **Grace**, a graphical constraint tracing environment on top of the **ECL^{i}PS^e** system [5]. The special control primitives available in **ECL^{i}PS^e** [9] make it possible to implement the whole environment in **ECL^{i}PS^e** itself and still to meet most of the requirements stated above. The graphical part was developed using the **ProTcl** [6] interface to the Tcl/Tk toolkit [10]:

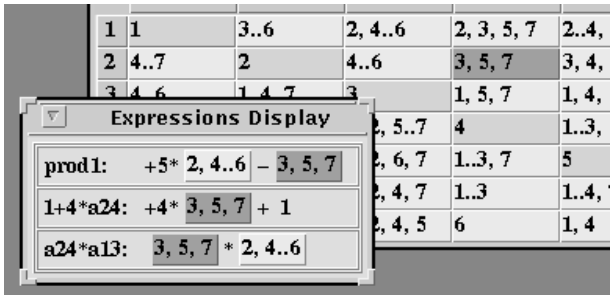


3.1 FD Variable Display

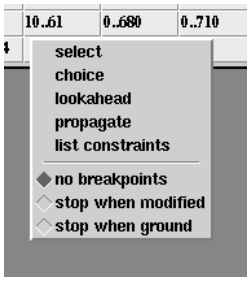
The FD variables in **Grace** are displayed as active fixed-size buttons with the variable domain as button text:

To be able to spot different displays of the same variable on the screen, each variable is a *hyperlink*: when the cursor enters the variable display, the button is highlighted and, if there are other displays of the same variable on the

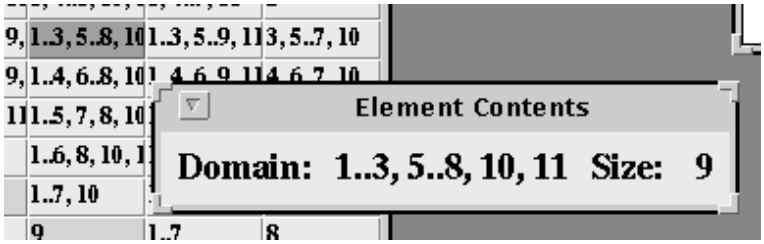
screen, they are highlighted as well:



When pressing the first mouse button in the variable display, a menu pops up with various options depending on the window type in which the variable is located:



The middle mouse button causes a separate display of the whole domain and its size, which is useful if the domain is too long to fit into the button:



Each variable has an attached *print daemon* which updates the display if the domain of the variable changes and undoes this change on backtracking:

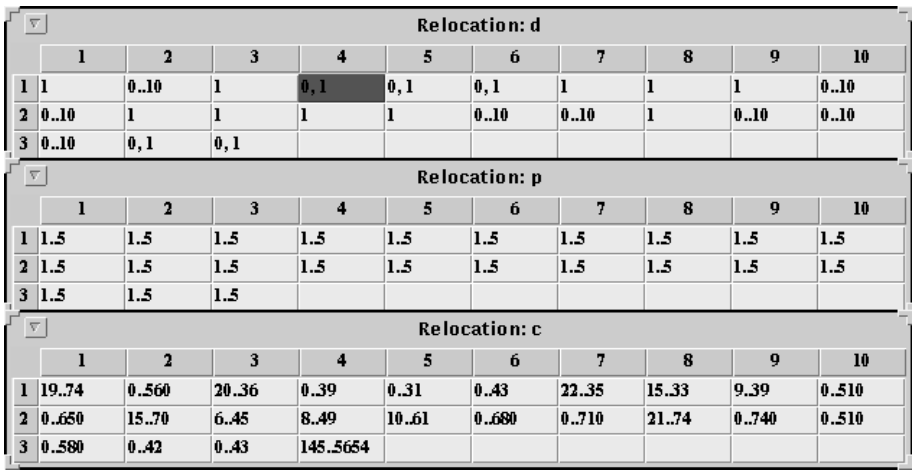
```
% display_handler(Var, OldDomain, VarID)
display_handler(Var, _, ID) :-
    var_domain(Var, D),
    <Display in Tk the domain D at the location ID>
    (var(Var) ->
        % wake when domain of Var changes
        delay(display_handler(Var, D, ID))
    ;
        true
    ).
display_handler(_, Old, ID) :-
    <Display in Tk the domain Old at the location ID>
```

```
fail.
```

In this way, the display can be incrementally updated on both forward and backward execution. This is a well-known CLP(FD) trick to keep the variable display valid, available in languages which can attach a daemon to a FD variable which is triggered on variable domain updates.

3.2 Variable Matrices

The FD variables represent the actual state of the computation and it is therefore important to display them all. **Grace** displays variables in two-dimensional *variable matrices* in a spreadsheet format, because this allows to display a large number of variables in a dense and structured way:



Relocation: d

	1	2	3	4	5	6	7	8	9	10
1	1	0..10	1	0,1	0,1	0,1	1	1	1	0..10
2	0..10	1	1	1	1	0..10	0..10	1	0..10	0..10
3	0..10	0,1	0,1							

Relocation: p

	1	2	3	4	5	6	7	8	9	10
1	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
2	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
3	1.5	1.5	1.5							

Relocation: c

	1	2	3	4	5	6	7	8	9	10
1	19..74	0.560	20..36	0..39	0.31	0..43	22..35	15..33	9..39	0.510
2	0..650	15..70	6..45	8..49	10..61	0..680	0..710	21..74	0..740	0.510
3	0.580	0..42	0..43	145.5654						

The user can put all variables into one matrix or split them into several matrices. Each matrix can be displayed or hidden independently and when it is resized, the appropriate font is selected.

3.3 Variable Identification

When the user wants to query or modify data attached to a variable, two cases may occur:

- If the data is directly attached to one displayed variable, then the operation can be performed with the mouse. This concerns e.g. modifying the domain of a variable, selecting for the next labeling step, listing constraints attached to a variable etc.

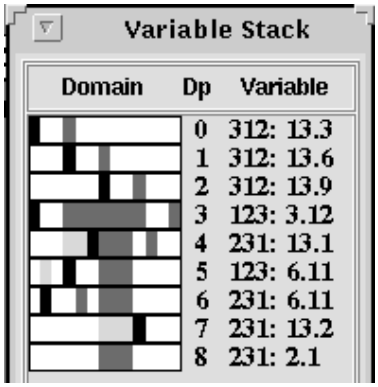
◯ If the data concerns more than one variable or if it involves typing in additional information, a textual identification of the variable(s) is used [1].

Variables cannot be usually identified by their name because in most CLP(FD) applications they are created dynamically with the same name, e.g. in a recursive predicate. **Grace** therefore uses a different naming scheme, which is shared by the control and the graphics part. Each variable is identified by the name of its matrix and its position, e.g. the highlighted variable in the picture above is a string "**d.1.4**". Then, for example, the expression $5 * D_{1,4} + 3 * P_{1,4}$ would be typed in as

```
5 * "d.1.4" + 3 * "p.1.4"
```

3.4 Search Path Display

Grace uses the *variable stack* to display the current position in the search space and the previous and remaining choices:



Each row in the variable stack represents one variable that has been already labelled. The bar to the left represents the variable domain, to the right there is the depth in the search space and the variable position ('312' etc. happen to be the matrix names). The colours in the domain bar have the following meaning (increasing lightness in black and white display):

red - current variable value,

blue - values in the domain to be still tried,

gray - values that have been already tried (and failed),

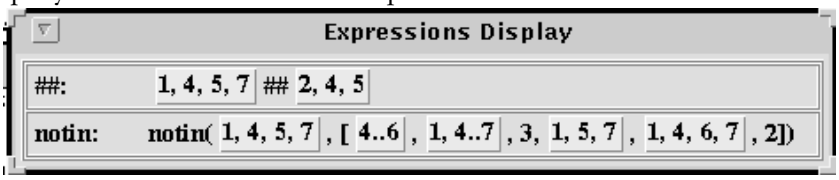
[1] A more mouse-oriented approach would be to type in an expression without variables and when it is displayed, to use a drag and drop operation to insert the appropriate variables. This requires more user actions than typing in, moreover a fully programmable environment needs a textual representation of variables in any case, so that displays can be pre-programmed.

white - values already removed from the domain before the variable was selected.

Each row has a popup menu which allows to retry the labeling at this depth, skip to the next value of this variable and fail to the next value of the variable (e.g. to speed up the execution if there is no solution for this label).

3.5 Terms and Expressions Display

Grace displays the value of selected terms and expressions in a separate window. *Terms* are arbitrary Prolog terms which are displayed in their usual form except that the variables in them are displayed as hyperlinks and their display is updated when the variable domain changes. This form can be used e.g. to display the current state of a specified constraint[2]:



Constraints attached to a particular variable can be listed from its menu and with a mouse click they are moved to the expression display.

Expressions, on the other hand, are being re-evaluated each time one of their variables becomes instantiated and the display is updated accordingly. For example, the following picture shows a sequence of updates to a cost expression:

3.6 Execution and Display Control

Grace is being controlled from the *Control Panel* window. It contains a number of buttons to control the execution, checkbuttons that control display

[2] Note that it does not matter how the constraint is actually implemented. Even if it is compiled into a sequence of machine or abstract instructions, or decomposed into a number of primitive constraints, we only need its structure.

of variable matrices, and several status items. There are two basic modes for the execution, *step* and *run*. In the *step* mode, the execution stops before every labeling step and the whole display is updated. In the *run* mode, the execution stops only on breakpoints or when a solution is found or when the user types Ctrl-C. The display updates in the *run* mode are controlled by Display options in the control panel:

none - the display is not updated at all, the program is running with maximal speed. With this option, the overhead of the debugging environment is minimal (less than 3 %) but whenever the execution stops, the complete execution state can be displayed. This feature alone is extremely helpful for tracing CLP programs.

variable stack - only the display of the variable stack is updated every time a new variable is labeled or when the program backtracks. This is relatively fast and it gives the user an animated view of the execution.

expressions - in addition to the variable stack, the expression display is updated as well. The user has thus the possibility to see the animation of certain variables or expressions which is still faster than to display all variables.

all - on each labeling step, the whole display is updated, including all visible variable matrices. This mode is provided mainly for demonstrations.

The execution can also be traced in fine-grained steps:

- Stepping through domain updates. The execution stops each time a variable is updated, this variable is highlighted and the old domain is printed in the status display. When the propagation fails, the system warns the user and then it waits for the next user command. In this way, the variable configuration directly before failure can be inspected.
- Stepping through woken constraints. All constraints which are triggered and re-evaluated up to the next labeling step are displayed in a separate window in a way which is similar to Prolog debuggers, with the CALL and EXIT or FAIL ports. This display is user-defined, the system may stop at every constraint invocation and update the display, or stop on failure, etc.

At any time, all matrices can also be printed on the printer for a more thorough or off-line analysis of their state.

3.7 Running Two Programs Concurrently

With **Grace** it is possible to compare two different strategies applied to the same problem. Both applications are started with **Grace** and one of them, the

slave, is then attached to the other one, the *master*, by clicking on the *Attach* button. From now on, most of the control commands executed in the master will be communicated to the slave so that both processes are synchronised. Moreover, the variable selected for labeling in the master will also be selected in the slave. At each labeling step it is then possible to see the comparison of the current values in variable matrices: identical domains are unchanged, smaller domains are displayed in lighter colour whereas larger domains use darker colour. Variables whose domains are not comparable are displayed in a different colour. This mechanism gives the user rudimentary facilities for comparison of two different approaches to solve the problem.

3.8 Interface

The interface between **Grace** and the user CLP(FD) program consists of several predicate calls that have to be inserted into the user program at appropriate places:

- **grace_start(Title, MatrixList)** - start **Grace** with a given title and a list of variable matrices.
- **grace_label(Var, Rest, NewVar, NewRest)** is inserted into the labeling predicate just before labeling the variable **Var**. Since the selected variable can be modified by the user, this predicate returns the newly selected variable and the rest as separate arguments. The labeling predicate then looks for instance as follows:

```
labeling([]).
labeling(Vars) :-
    deleteff(Var, Vars, Rest),
    grace_label(Var, Rest, NewVar, NewRest),
    indomain(NewVar),
    labeling(NewRest).
```

- **grace_display_term(Term, Name)** displays the given term in the Expression display.
- **grace_solution** is called when a solution has been found. It stops and displays the execution state.

4 Implementation Issues

The implementation of an efficient and flexible debugging environment is a challenging task. The more functionality is required from the system, the more of complex and low-level coding is necessary to implement it. Low-level

coding, however, is not compatible with flexibility, maintainability and extensibility. High-level coding, on the other hand, usually causes an unacceptable execution overhead. For example, the otherwise excellent OPIUM system [2] gives the user a wide spectrum of ways to debug Prolog programs, however it can hardly be used for real-time debugging, due to the time and space overhead.

The $\mathbf{ECL^iPS^e}$ system [4, 3] based on Sepia [8] is a logic programming system which has been designed to support a wide range of extensions. To implement the extensions in an efficient way, the system provides high-level interfaces to low-level kernel primitives so that the whole extension code can be written in $\mathbf{ECL^iPS^e}$ itself. $\mathbf{ECL^iPS^e}$ also provides several CLP libraries for various domains.

Grace has been fully implemented in $\mathbf{ECL^iPS^e}$ and in Tcl/Tk and its implementation has benefited to a large extent from the special control and extension primitives available in $\mathbf{ECL^iPS^e}$. In this section, we will list the most interesting features of the implementation and the $\mathbf{ECL^iPS^e}$ primitives which were exploited. We will at the same time refrain from describing the complex issues of display and events handling which were tedious to implement but are not of particular interest for the CP community.

4.1 Logical vs. Extralogical Primitives

Although a CLP program is declarative, a CLP debugger cannot be completely declarative, otherwise all user actions would be undone on backtracking and some features would be difficult or impossible to implement. On the other hand, some user actions, like setting a breakpoint on a variable update, can be undone on backtracking without much harm. We have therefore decided to implement **Grace** as declaratively as possible, without defining special primitives to support imperative actions of the debugger.

4.2 Global Data

$\mathbf{ECL^iPS^e}$ provides two kinds of global data: *global variables* which are in fact destructively updatable arrays and which can be used to store ground terms, and *global references* which can also be updated but their value is restored on backtracking and they can contain terms with variables. All permanent debugger data have been implemented with the former type, for instance the current execution mode, current execution priority and breakpoints not related to variables (e.g. search depth or goal number).

The backtrackable global references have been used to store all variable matrices. In this way, the matrices are accessible everywhere in the program without having to pass them as arguments to all predicates. The current search

depth is also implemented using this mechanism, because it has to be restored on backtracking.

4.3 Variable Attributes

The attributed variables available in **ECL^{i}PS^e** [5] make it possible to associate transparently attributes to variables. **Grace** uses this mechanism for several purposes:

- to associate the variable identification (i.e. matrix name and position) with the variable,
- to remember the initial variable domain which is necessary for the display in the variable stack
- implicitly, to associate daemons with variables; this was done using the coroutining primitives which are themselves built on top of the attribute scheme. The system uses various daemons: to update the display of variables, to recompute and redisplay expressions, to set a breakpoint on a variable and to step through domain updates.

4.4 Execution Priority

A major issue in implementing **Grace** was the question of minimal execution overhead. With our scheme of associating print daemons with each variable it is not straightforward to suppress their execution in case we want to run the program with maximum possible speed. Fortunately, the new waking scheme in **ECL^{i}PS^e** which is based on suspension priorities [9] could be used for this purpose. Every suspended goal in **ECL^{i}PS^e** (suspension) has an associated priority. When the goal is woken, e.g. by updating the domain of a variable, it is not immediately executed, but it is passed to the *waking scheduler* instead. The scheduler takes care that all woken suspensions are executed in their priority order and that a suspension with lower priority does not interrupt the execution of predicates with higher priority.

This scheme could be directly exploited in **Grace**: the various daemons have different priorities and the program is also assigned a given priority depending on the amount of display updates which are required. When no updates and maximum speed is required, the program priority is set to a value which is higher than that of any display daemon. Whenever a domain variable is updated, its associated print daemon is woken and passed to the waking scheduler, but since its priority is not high enough, it is never actually executed, the display is not updated and also no new daemon is placed on the variable from its body. The overhead is therefore reduced to the first waking of the

daemon of each variable, which is a simple operation. By setting the program priority to different values the amount of display updates can be controlled.

4.5 Stepping through Domain Updates

Showing the successive changes of variable domains is also implemented using the waking priority scheme. When this execution mode is first entered, the system places another display daemon with a very high priority on each variable. When a domain variable is updated, this daemon will be the first attached suspension to be woken (before all the constraints). It will update the display and then wait for user action, i.e. clicking on a button. Then it places a new daemon on the variable and exits. In this way, all domain updates can be successively traced no matter how many constraints are invoked inbetween.

4.6 Breakpoints on Variables

The **ECL^{i}PS^e** finite domain constraint solver [5] allows to set up suspensions which are woken when a variable is constrained in various ways:

- the minimum or maximum of its domain changes,
- any element is removed from the domain
- variable is instantiated
- variable is bound to another constrained variable

This mechanism is used to set breakpoints on these events for a particular variable: when the suspension is woken (its priority is higher than that of the *run* mode), it changes the mode to *step* and prints information about the break into the status line. Note that this approach imposes no execution overhead.

4.7 Interrupting the Execution

When the program executes in the *run* mode, no X events are served (because of efficiency) and thus it cannot be stopped by clicking on a button. Instead, the handler for the keyboard interrupt signal is modified to set the *step* mode so that the execution stops on the next labeling step.

4.8 Stepping through Woken Constraints

Constraints are implemented in **ECL^{i}PS^e** as suspensions which wait for a modification of one of their variables. When the variable is modified, the

suspension is woken and the constraint is thus re-evaluated. As we said before, the waking does not happen directly, but through the waking scheduler. When stepping through woken constraints is required, **Grace** redefines the built-in waking scheduler with a predicate which is equivalent, but enhanced with printing the debugging information before and after calling the woken suspension. When this stepping is no longer required, the built-in waking scheduler is restored again.

The waking is thus temporarily enhanced and slowed down, which is exactly what is needed. Moreover, the user can define her own way to trace the woken constraints.

4.9 Listing Constraints Attached to a Variable

The constraints mechanism in **ECL^{i}PS^e** is fully accessible to the user and it is thus possible to obtain the list of constraints attached to a particular domain variable using available primitives.

4.10 Restoring a Previous State

For the *retry* command it is necessary to restore a previous execution state and restart it. **Grace** creates an additional choice point on each labeling step, which is used to count the number of backtracks. This choice point can also be exploited to restore a previous state, if the execution fails up to this point. This failure, however, is not always easy to enforce: **Grace** can of course force the current labeling step to fail and to fail each time it obtains control until the target depth is reached, however the user program obtains control after each labeling step and after failure it retries another value in the domain of the labeled variable (see the **labeling/1** predicate on p. 13). This means that when executed declaratively, the program would perform a potentially large number of backtracks before it fails to the right choice point. Experiments have shown that this overhead is not acceptable and we thus had to use an impure primitive, *nonlocal cut* to handle this case. **ECL^{i}PS^e** offers two primitives, **get_cut/1** and **cut_to/1** which implement a nonlocal cut. **get_cut/1** marks the current execution state and returns this mark, whereas **cut_to/1** removes all choice points up to a specified mark. This mechanism is used to remove all user choice points in each labeling step so that retrying is an operation which is only proportional to the depth in the search space, but not to the size of involved domains.

4.11 Comparing Two **Grace** Processes

The Tk toolkit has a *send* command which allows one Tk application to communicate with another Tk process. This mechanism has been used to attach one **Grace** process to another and to synchronise their labeling.

4.12 Handling Window Events

Most of the GUI events are handled in $\mathbf{ECL^iPS^e}$. When the execution stops, the system blocks and waits for an X event. When it arrives, it is served by the $\mathbf{ECL^iPS^e}$ code and, depending on its type, the execution either continues, possibly in new mode, or it waits for another event.

The events which have no influence on the CLP data or execution (e.g. highlighting a variable when the cursor enters it) are handled by the Tk toolkit itself, so that the Prolog and Tcl code is cleanly separated.

5 Conclusions and Future Work

We have presented some basic principles for debugging CLP programs and from them we have concluded the properties of a system to support them. The design and implementation of **Grace** was surprisingly easy, taken into account the expected functionality and complexity. On preliminary tests [7] it showed to be quite useful and helpful for CLP tracing and debugging. As soon as the system is stable and foolproof enough, it will be released for public use.

There are many possible ways to enhance the current version of **Grace**, ranging from cosmetic ones to significant extensions. The most interesting ones are:

- Support for conditional breakpoints and user-definable breakpoints.
- Save/restore facility.
- Display the propagation steps in a graph format, similar to the Causality Graphs of [1]. It might also be interesting to explore the possibility of displaying selected parts of the constraint network as a graph. Since the actual constraints used to implement the application may be at a too low conceptual level, we see a need for more global and generalised representation of the constraint network which would give the user a good overview at the required conceptual level.
- Create a sophisticated repository of evaluation and labeling strategies and integrate them seamlessly into the **Grace** tracing paradigm.

- Support for parallel execution. The **ECL^{i}PS^e** system is able to perform OR-parallel search on shared-memory multiprocessors and in its next version it will be able to perform parallel search also on a network of workstations. The search paradigm in this context is slightly different from the sequential one - variables no longer have one value, because different processes explore different search paths in parallel and give the variables different values. On the other hand, the variable stack could be enhanced to visualise the search in all processes concurrently.

- With support for parallel execution it becomes very interesting to run several different strategies in parallel, possibly with some limited communication among the parallel processes. Rapid prototyping or experimental debugging in this context is an important area to explore.

- Inclusion of other methods than labeling based on backtracking search. Ideally, the debugging environment would also provide a number of other methods, e.g. local repair or statistical methods and it would allow to combine them into new and possibly very powerful strategies.

Bibliography

[1] Michael Dahmen. A Debugger for Constraints in Prolog. Technical Report ECRC-91-11, ECRC, 1991.

[2] M. Ducassé. Opium+, a meta-debugger for Prolog. In *Proceedings of the European Conference on Artificial Intelligence*, Munich, August 1988.

[3] ECLiPSe.
URL http://www.ecrc.de/eclipse/eclipse.html, 1995.

[4] *ECLiPSe 3.5 User Manual*, 1995.
URL http://www.ecrc.de/eclipse/html/umsroot/umsroot.html.

[5] *ECLiPSe 3.5 Extensions User Manual*, 1995.
URL http://www.ecrc.de/eclipse/html/extroot/extroot.html.

[6] Micha Meier. ProTcl, the Prolog interface to the Tcl/Tk toolkit.
URL http://www.ecrc.de/eclipse/html/protcl.html.

[7] Micha Meier. Visualizing and solving finite algebra problems. In *Workshop on Finite Algebras*, ECRC, Munich, March 1994.

[8] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[9] Micha Meier and Joachim Schimpf. Control in ECLiPSe. Technical Report ECRC-95-07, ECRC, February 1995.
URL http://www.ecrc.de/eclipse/html/reports.html.

[10] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[11] Michael Sanella. Analyzing and debugging hierarchies of multi-way local propagation constraints. In *Proceedings of the 1994 Workshop on Principles and Practice of Constraint Programming*, 1994.