facultad de informática

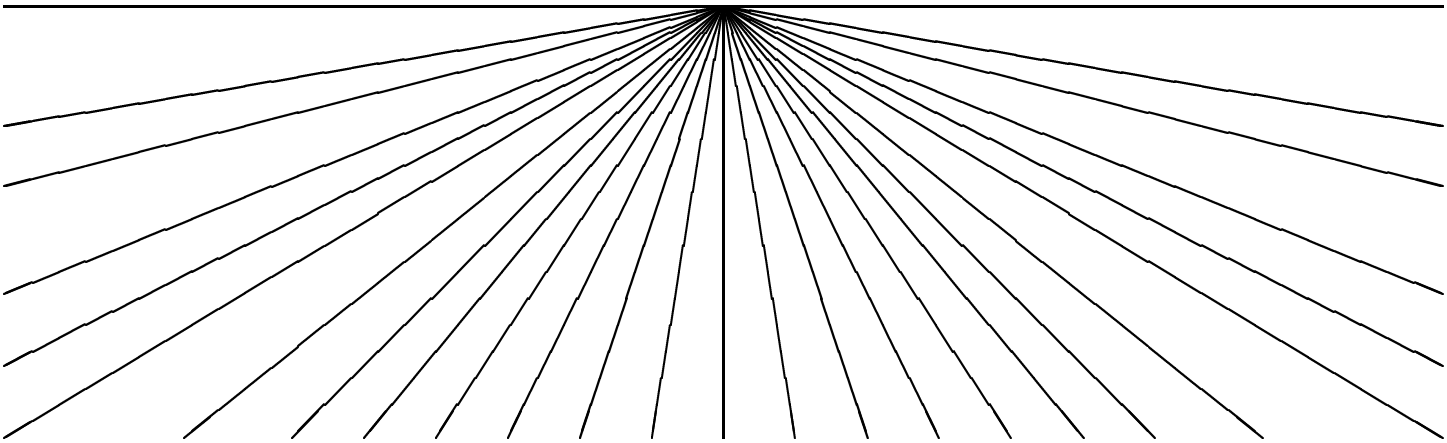
universidad politécnica de madrid

Some Design Issues in Constraint Program Visualization and Abstraction

Manuel Carro
mcarro@fi.upm.es

Manuel Hermenegildo
herme@fi.upm.es

TR Number CLIP1/97.1



Some Design Issues in Constraint Program Visualization and Abstraction

Technical Report Number: CLIP1/97.1

Authors

M. Carro
M. Hermenegildo
{mcarro, herme}@fi.upm.es
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid — Spain

Keywords

Constraint Logic Programming; Visualization; Debugging.

Acknowledgements

We wish to thank all the partners in the DISCIPL project, especially Abder Aggoun and Helmut Simonis from COSYTEC, and Pascal Bouvier and Claude Lai from INRIA, and also to the members of the CLIP group (http://www.clip.dia.fi.upm.es), for discussions regarding program visualization and debugging.

This work has been partially supported by the European ESPRIT LTR project DISCIPL # 22532 and Spanish CICYT Projects E96-1015-265 and TIC96-1012-C02-01.

Abstract

Visualization of program executions has been proposed previously, for purposes which include education and debugging. However, traditional visualization paradigms often fall short of expectations or are altogether inadequate for new programming paradigms, such as CLP, whose declarative and operational semantics differ in some crucial ways from those of other paradigms. In particular, traditional ideas regarding flow control and the behavior of data cannot often be straightforwardly lifted to (C)LP from other families of programming languages. In this paper we discuss the issue of how to visualize program execution and data evolution in constraint logic programming. We discuss a number of previously proposed visualization paradigms, and also propose a number of (to our knowledge) novel ones. The proposed graphical representations have been chosen based on the perceived needs of a programmer trying to analyze the behavior of large execution. In particular, we concentrate on the representation of the program execution behavior (control), the runtime values of the variables, and the runtime constraints. Given our interest in visualizing large executions, we pay special attention to abstraction techniques, i.e., techniques which are intended to help in reducing the complexity of the visual information.

Resumen

La visualización de la ejecución de programas ya había sido previamente propuesta para propósitos que incluyen la enseñanza y la depuración. Sin embargo, los paradigmas tradicionales de visualización a menudo no son adecuados (o incluso completamente inadecuados) para nuevos paradigmas de programación, como CLP, cuyas semánticas declarativa y operacional difieren en aspectos cruciales de las de otros paradigmas. En particular, las ideas tradicionales relacionadas con el control de flujo y el comportamiento de los datos no pueden ser transportadas directamente a (C)LP desde otras familias de lenguajes de programación. En esta artículo se examina el problema de la visualización de la ejecución de programas y la evolución de los datos en programas lógicos con restricciones. Se comentan algunos paradigmas de visualización ya existentes, y se proponen otros (según nuestro conocimiento) novedosos. La representaciones gráficas han sido escogidas basándose en las necesidades del programador a la hora de analizar el comportamiento de una ejecución no trivial. En particular, nos concentraremos en el comportamiento del programa (control), los valores de las variables en tiempo de ejecución, y las restricciones en tiempo de ejecución. Dado nuestro interés en la visualización de ejecuciones de tamaño considerable, prestaremos especial atención a las técnicas de abstracción, es decir, técnicas cuyo propósito es ayudar en la reducción de la complejidad de la información visual.

Contents

1 Introduction 1

2 Visualizing Control 1
- 2.1 Looking at Program Execution . . . 2
- 2.2 Looking at the Enumeration Process . . . 3
- 2.3 Visualization and Verification . . . 4

3 Displaying Variables 5
- 3.1 Herbrand Terms . . . 5
- 3.2 Finite Domains . . . 6
- 3.3 Intervals . . . 7
- 3.4 Linear Constraints . . . 7

4 Representing Constraints 8

5 Abstraction 10

6 Abstracting Control 10

7 Abstracting Values 11

8 Abstracting Constraints 13

9 Conclusions 14

1 Introduction

Visual depiction of program executions has a long history in Computer Science. Visualization has classically focused on the program flow (using flowcharts and block diagrams, for example) or on the data managed by the program and its evolution as the program executes (see, for example, [BDM97] or "Animated Sorting Algorithms", under `http://java.sun.com/applets/`).

It turns out that previously proposed visualization techniques cannot capture completely the nature of the computations performed in newly devised programming paradigms (CLP, in our case). Capturing the computation spirit is important, because one of the motivations of program visualization is providing an intuitive, human-oriented view of what could otherwise be presented as raw, possibly numeric, data.

Visualizing execution has been used for a series of purposes (which can, of course, be applied to CLP as well), among which we may cite:

- Program and system debugging: in this case it is crucial that the programmer can obtain a clear view of the program state at every moment from the picture displayed. This is complementary to other methods (e.g., assertions [AM94, DNTM89, DNTM88, BDD+97] and text-based debugging [Byr80]), and can be viewed, in some sense, as a graphical front-end to old text-based debuggers.

- Tuning and optimizing programs and programming systems (which may be termed as performance debugging). They are intended to be complementary to other well-known methods as, for example, profiling statistics.

- Teaching and education. Some applications to this end have already been developed and tested, using different approaches (see, form example, [EB88, Kah96]).

A good representation design of the data managed by the program, or of its execution, if that is the case, is key for achieving a useful depiction in any of the above targets. A problem which appears with graphical representations of medium-sized program executions is the huge amount of information that might be available. To cope successfully with these cases, an *abstraction* of the view is needed; in this abstraction the picture should show the most interesting characteristics, which may be different in each case, without cluttering the display with unneeded (for the purpose at hand) details.

In this paper we will explore some ideas related to the topics mentioned above: the displaying of the execution, the data, and abstractions of those depictions. The aim of the visualization paradigms we will discuss is quite broad — i.e., we are not committing exclusively to teaching, or to debugging —, but our main focus will be debugging, both for correctness and for performance. Therefore, some characteristics desirable in educational approaches may be absent. We will divide the visualization paradigms into three categories (which, as we will see, can coexist together seamlessly, and even be used together to help a better visualization): visualizing the execution flow / control of the program, visualizing the actual variables (i.e., representing their runtime values), and visualizing constraints among variables. The three views are amenable to be *abstracted* in order to untangle many characteristics which are shown with too much detail.

2 Visualizing Control

One of the main characteristics of declarative programming is the absence of explicit control. Although this theoretical property results in many advantages regarding, for example, program analysis, actual programs are executed with fixed evaluation rules,[1] and different (but otherwise correct) programs aimed at achieving the same results can exhibit different behaviors concerning their performance, even when executed in the same environment. On the other hand, understanding those evaluation rules is a must in order to write efficient programs (including termination, which obviously affects correctness); and even when they are fully understood, having a good visualization of the program execution (probably combined with other tools) helps to uncover bugs otherwise difficult to track down.

[1] Fixed in the sense that they can be deterministically known at every point in time, but maybe not known statically.

2.1 Looking at Program Execution

Performance debugging aims at discovering which are the sources of unexpected low performance in a program. When addressing performance debugging, the visualization of the execution profile (which, for LP programs traditionally takes the form of a search tree) can help in deciding which computation patterns have been executed more often; the shape of the tree informs the programmer about which parts dominate execution time, assuming similar time for every node (corresponding to `calls`, `redos` and `failures`). When more accuracy is needed, it might be worth using a sort of time measure: for example, drawing execution trees in a way such that the distance between a node and its children reflects the elapsed time. This was the primary execution view provided by VisAndOr [CGH93], a graphical tool aimed at displaying and understanding parallel execution of logic programs. Figure 1 shows a snapshot of a VisAndOr window, in which an or-parallel execution is drawn; time runs from top to bottom, and parallel tasks are exhibited as vertical lines. These lines represent, using color and different thickness, which processor executes each task and the state of the task: running, waiting to be executed, or finished.

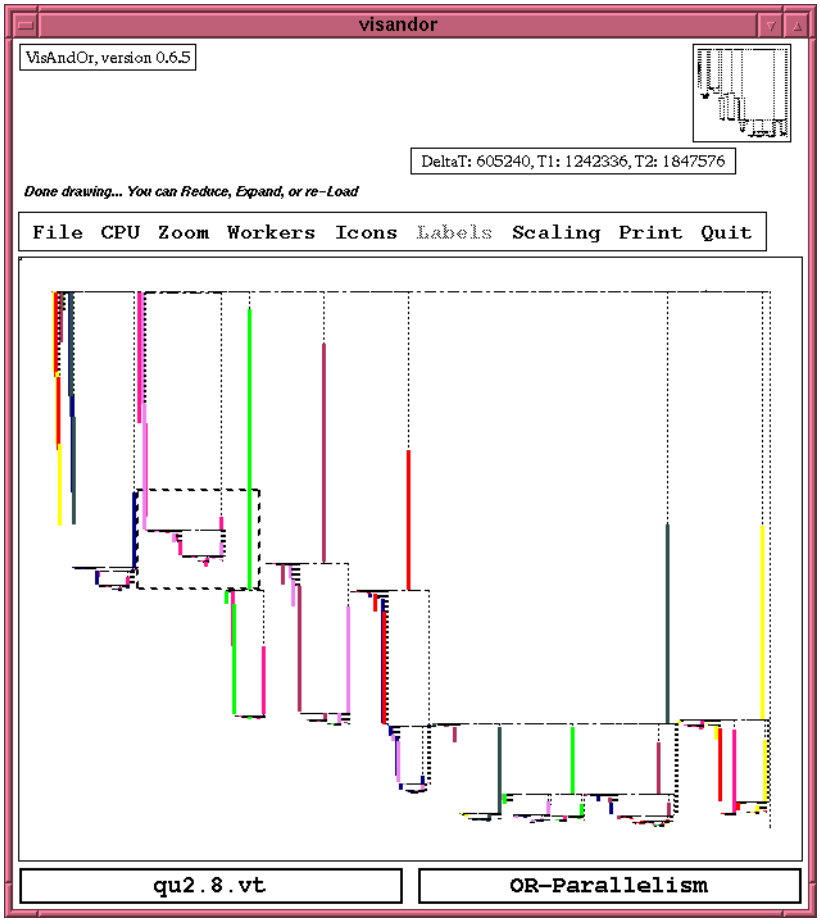


Figure 1: Visandor showing an execution in time space

In this case, the length of the vertical lines reflects accurately a measure of the time spent by the task. VisAndOr allowed switching to a *events space*, in which every event in the execution (say, the creation, the start, and the end of a task, among others), take the same amount of space. Note that the structure of the execution is easy to see, but the notion of the time is lost. The same execution represented in Figure 1 is depicted in Figure 2, but using the events space. In the usual tree representation for the execution of logic programs, events can be associated to the `calls` made in the program, and the most common representation is in events space. A representation in time space would enlarge or shorten the edges of the tree to reflect the time elapsed between the corresponding events. Time elapsed between calls, viewed as spatial distance

in the tree, would provide with feedback about the complexity of head unifications and, in a CLP setting, addition and partial solving / simplification of constraints.

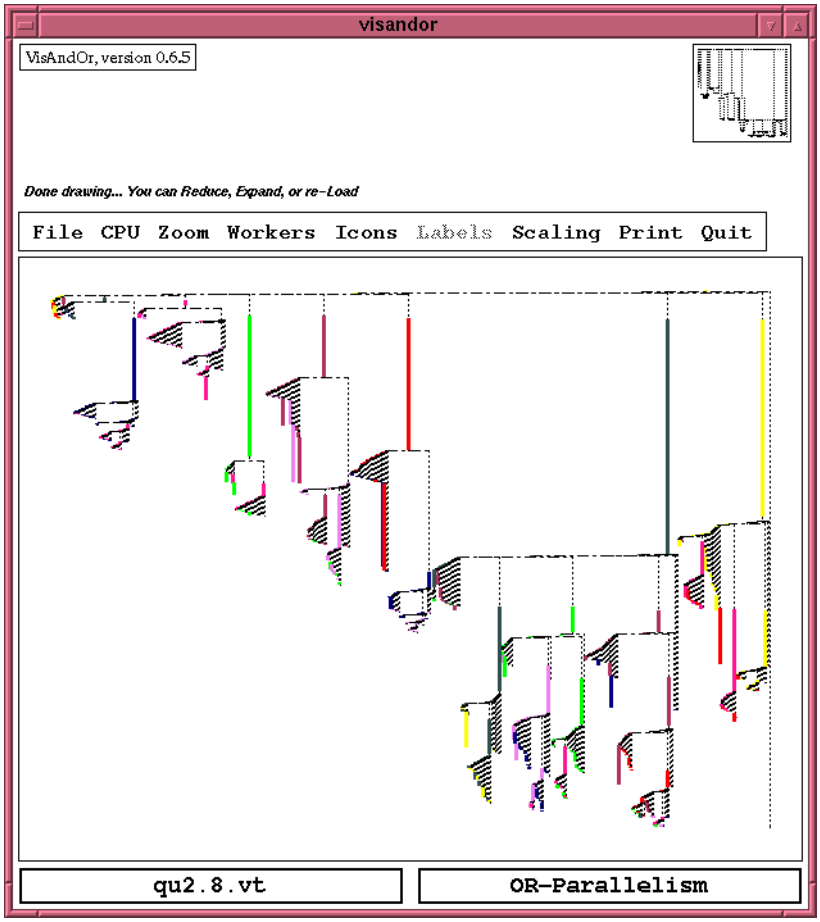


Figure 2: Visandor showing an execution in events space

APT (*A Prolog Tracer*)[Lue97] is a prototype tool based based on the Transparent Prolog Machine (TPM) [EB88], which visualizes the execution of Prolog programs. It has been implemented and is under testing at UPM. APT inherits from the TPM the use of an augmented AND-OR tree, in which AND and OR branches are somewhat compressed and take up less vertical space.

APT displays an animation of the whole search tree (Figure 3), and allows replaying the execution step by step. The state of each node (not yet called, call not yet finished, failure) is shown by means of a color code. It also offers the possibility of expanding nodes and showing the program source with the actual substitutions for the variables — which are also animated, and change as the program executes (Figure 4).

The concept underlying APT is quite general, and, although it has been implemented with Prolog and Herbrand terms in mind, it is possible (and has been planned) to extend it to work also with constraints. From a user point of view, the interface would remain essentially the same, but clicking on a node would bring up a representation similar to that of Figure 4, but showing also the values of the variables and the constraints attached to them, possibly using depictions similar to those suggested in Sections 3 and 4.

2.2 Looking at the Enumeration Process

Some solvers (e.g., those for finite domain constraints) perform internal decisions regarding domain splitting for branch-and-bound methods, or choose paths for constraint propagation, or, at least, give the user the possibility of choosing among several heuristics for the enumeration of solutions. Showing how these pro-

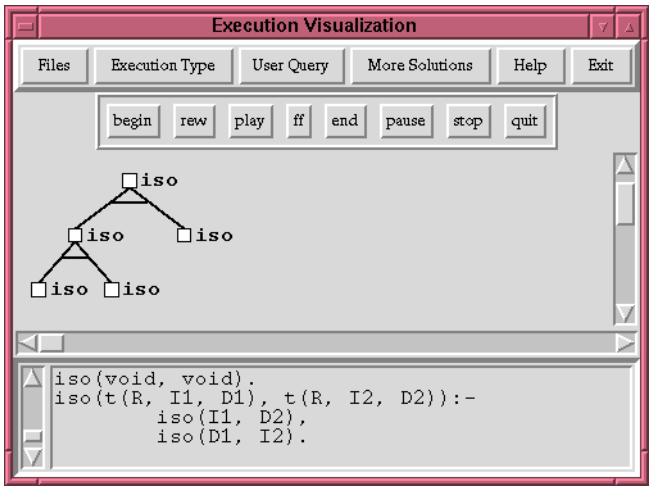


Figure 3: A small execution tree, as shown by APT

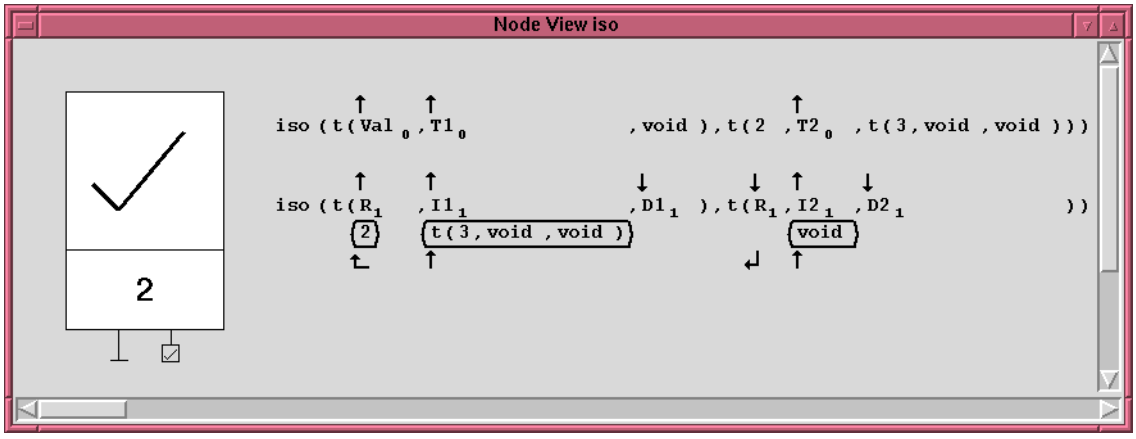


Figure 4: Detailed view of a node (Herbrand domain)

cedures behave helps not only in evaluating the solver and its performance for a given class of problems, but also in deciding how a program can be improved to show better results when using a solver: it can be expected that setting up the program constraints in a different way (i.e., different constraints which model the same problem, or using more or less redundant constraints, etc.) will have an impact on the performance. Those differences can be more easily understood if we have an intuitive representation of the work internally performed by the solver.

2.3 Visualization and Verification

Coupling control visualization with program annotations (which represent user assertions on properties of the program) is also interesting: even if these assertions are not an executable part of the program, they might be checked by the (visual) debugger and (if violated) highlighted in the search tree. This will also help the programmer to know exactly in which part of the execution this assertion was not respected and, with a view of the actual instantiation state, why this has been so. The visualization of the variables and constraints can be interfaced with the visualization of the search tree by allowing the user to "point'n'click" a node and bringing up a visualization window (perhaps under the control of a different application) which shows the variables / constraints active at the moment in history in which the node was clicked. APT has this possibility, but integrated into a single, monolithic tool. However, a very interesting characteristic is

that this view of the variables is updated as the execution of the program continues.

3 Displaying Variables

Variables in C(L)P programs are more difficult to represent than variables in other languages: they can have not only definite values, but also a set of possible values, and constraints attached to them which relate several variables and which restrict the values they can take simultaneously.

Textual representations of the variables in the store are usually not very informative and not easy to interpret and understand.[2] A graphical depiction of the values of the variables should offer a clearer view of the computation state at any point. If we wish to follow the history of the program (which is another way of understanding the program behavior, but focusing on the data evolution), it is desirable that the graphical representation could be either animated (then time in the program is represented as time) or laid out so that different behaviors can be easily compared (and then time is traded for space).

Since every constraint domain has different properties and semantics, different representations for variables will be needed for them. In what follows we will sketch some ideas dealing with the representation of variables in common constraint domains.

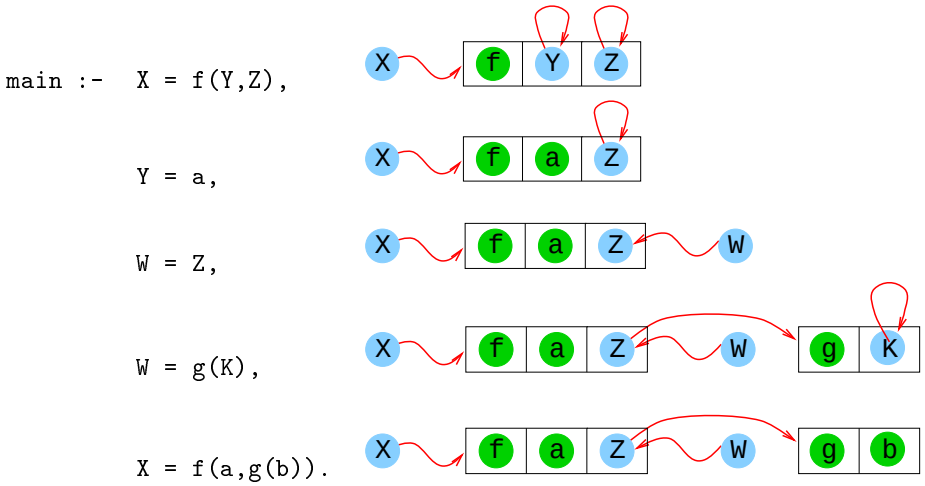


Figure 5: Representing the creation of a Herbrand term

3.1 Herbrand Terms

Herbrand terms can be written as first order terms; a graphical representation of those first order terms is as a n-ary tree, in which the root is the name of the n-ary functor, and the subtrees are the trees corresponding to the arguments. Whereas this is well suited for ground terms, free variables, which may have equality constraints among them, are not well represented (unless we resort to a giving names to every variable in an homogeneous fashion; a drawback of this is that it does not favor a "graphical" view which directs the eye effortlessly to the right point). A possibility is representing this sharing as just another edge (thus transforming the tree into an acyclic graph), and even, taking an approach closer to usual implementation designs, having a variable to point to itself. Figure 5 represents the step by step creation of a Herbrand term in a Prolog clause. Rational trees (as those supported by Prolog II, III, IV) can be also represented in a similar way — but in this case the graph can contain cycles, although it cannot be a general graph.

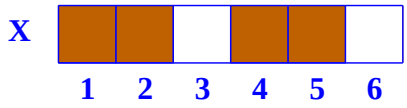


Figure 6: Finite Domain depiction

3.2 Finite Domains

Finite Domains (FD) are one of the most popular constraint domains. A FD variable can take any set of (integer) values among a finite set of them. Without loss of generality, FD variables can be assumed to be initialized at the beginning of the program to an integer range which includes all the possible values in the problem being tackled (in the default case, variables can be assumed to be initialized to the whole domain). It is interesting to show not only the current (set of) values the variable can take, but also the whole (initial) range of allowed values. This shows how the domain of the variable is being narrowed throughout the execution. A possible representation assigns a dot (c.f., a square) to every possible value the variable can take; therefore the whole domain is a line (c.f., a rectangle). Values inside the current domain at every moment are highlighted. An example of a variable (X) with current domain $\{1, 2, 4, 5\}$ from an initial domain $[1..6]$ is shown in Figure 6.

More possibilities [Mei96] include using different colors / shades / textures to represent more information about the values (e.g., how long ago they have been discarded from the domain of a variable).

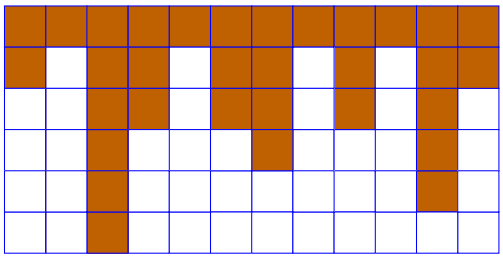


Figure 7: History of a variable

Representing the history of a variable through the program execution gives hints regarding when a given value is discarded (or not discarded), or whether some values where active in the variable domain longer than others. A temporal history of the variable can of course be shown with an animated display; but this disallows the immediate comparison of two different stages — going back and forth in time would be necessary; notwithstanding, it can be useful for understanding how he domains of the variables are narrowed.

Figure 8: Shades representing age of discarded values

Different degrees of shade in the values, representing how long that value has been removed from the domain of the variable could be used, but comparing them accurately is not easy for the human eye (although it may give a rough indication of the changes in the history of the variable; see Figure 8, where lighter grays are values removed more long ago). An alternative solution is simply stacking representations such as that in Figure 6, thus trading time for space (as done in visualization tools implemented by COSYTEC). They can be easily shrunk if needed to accommodate the whole variable history in a given space, and can reflect accurately the time (issuing time stamps with every change in the current domain) or ignore it, just stacking a variable representation every time its domain changes, or every time a enumeration step is performed.

[2] Moreover, some solvers maintain, for efficiency reasons, only an approximation of the values the variables can take.

An example sketching the latter idea is shown in Figure 7; a bigger example, using ASCII text and taken from the a program to solve the 12 queens problem is shown in Figure 9. Every column corresponds to a variable; for each variable, an asterisk represents a possible value inside the domain, and a blank space, a value ruled out by the active constraints. Vertical lines delimit the initial range of the variables. Every variable corresponds to a column, and its possible values are the row numbers in which a queen could be placed at every constraint addition and enumeration step.

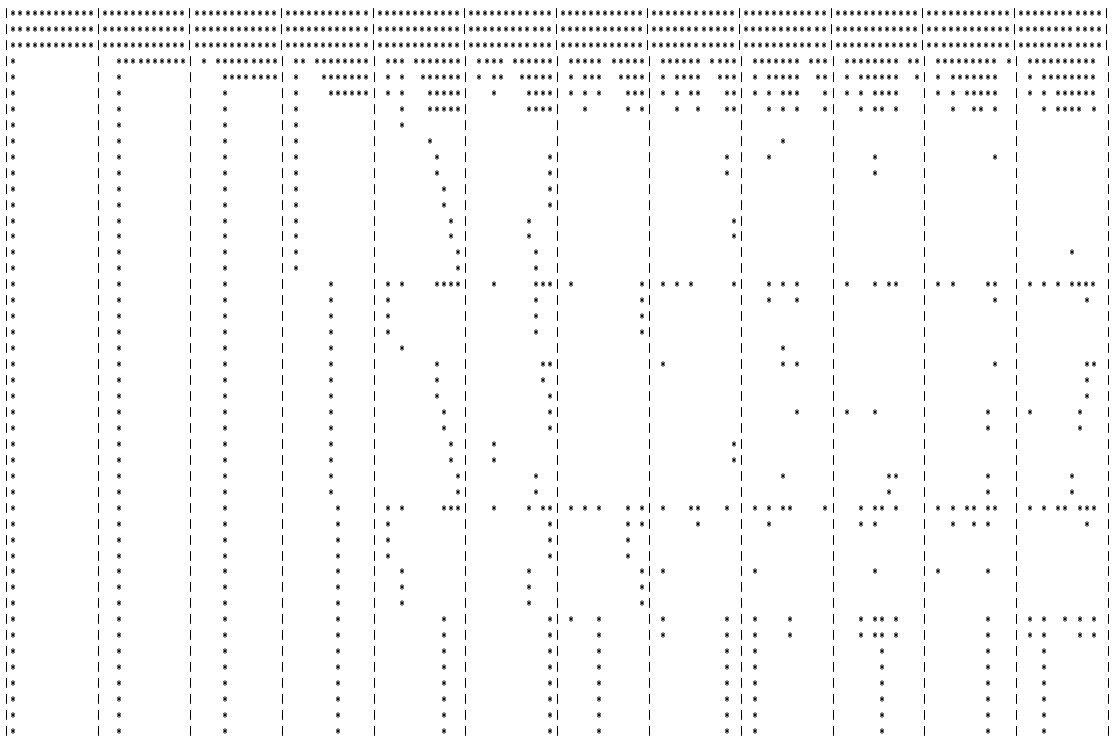


Figure 9: Evolution of FD variables for a 12 queens problem

3.3 Intervals

In some sense intervals are similar to finite domain variables: the constraints allowed to them are basically the same, but the set of values allowed is, in general, infinite and continuous. In practice, and due to computer limitations, this is not so, and representation similar to that of finite domains could be adopted. The main difference among interval and finite domains are the operations allowed between elements: intervals usually allow non-linear arithmetical constraints for which a procedure of solution is not known, which makes the solvers to be incomplete. Thus, the visualization of the actual domain[3] will normally be an upper approximation of the domain which mathematically would correspond according to the constraints which have been introduced by the program: trying to enumerate all the values the variable can take is, in most cases, impractical. That is a reason why an exact display of the intervals is not actually possible.

3.4 Linear Constraints

Obvious representations for linear constraints are dots, lines, plains, etc. The problem is that they easily go n-dimensional, and it does not seem that such a representation (even if it is accomplished in the cleanest

[3] Not only the representation, but also the internal representation, from which the graphical depiction is drawn.

graphical way possible) be really informative — or even possible. And, usually, it only makes any sense when it is shown together with the values of other variables.

4 Representing Constraints

Viewing relationships which hold among variables can be accomplished by simply dumping the constraints and the variables involved in the source code representation. Unfortunately, this is not straightforward, or even possible in some constraint domains, and provides too much level of detail for an intuitive understanding. Most of the characteristics we want to know will remain hidden under a lot of information.

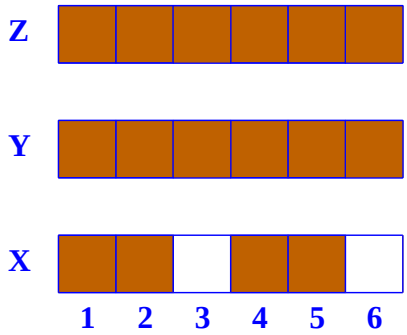


Figure 10: Several variable domains plotted

Constraint visualization should provide information about which variables are interrelated by the constraints, and how these variables affect each other. A generic possibility which takes advantage of the representation of the actual values of a variable is displaying several (or all) the variables, and allowing the user to update the value of one (or several) variable(s), and seeing how other variables are affected by this change. The update of the variables can be performed using the graphical interface, or adding manually a constraint involving the variables which need to be updated and which reflects the desired update.

In what follows we will focus on FD variables. The reasons for doing that is that Finite Domains are very widely used, and very useful for many purposes. Interval arithmetic, as we argued before, can use a representation similar to the finite domains one.

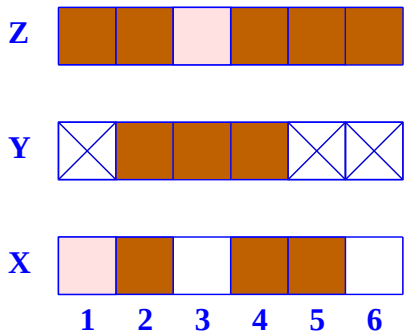


Figure 11: Changing a domain

We will use the following constraint (**C1**) in the examples which follow:

$$\mathbf{C1} : X \in \{1..6\} \wedge X \neq 6 \wedge X \neq 3 \wedge Z \in \{1..6\} \wedge Z = 2X - Y \wedge Y \in \{1..6\} \tag{0.1}$$

Figure 10 shows the actual domains of FD variables `X`, `Y`, and `Z` subject to the constraint **C1**. Boxes in grey represent points inside the domain of the variable, and white boxes stand for values not compatible with

the constraint(s). We can explore how changes in the domain of one variable affect the others by changing the domain of a variable. For example, we may discard the values 1, 5, and 6 from the domain of `Y`, which boils down to representing the constraint **C2**:

$$\mathbf{C2} : \mathbf{C1} \wedge Y \neq 1 \wedge Y \neq 5 \wedge Y \neq 6 \tag{0.2}$$

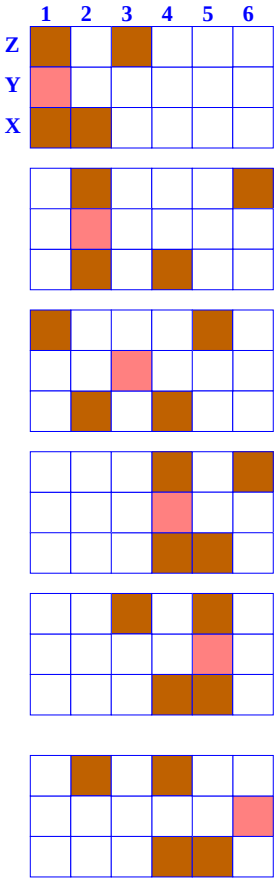


Figure 12: Enumerating `Y`, accumulating the enumeration of `X` and `Z`

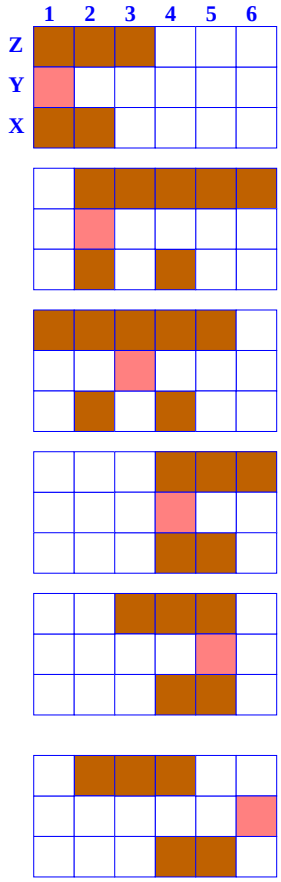


Figure 13: Enumerating `Y`, representing solver domains for `X` and `Z`

The variables are now represented in Figure 11. Values directly disallowed by the user are shown in Figure 11 as crossed boxes; values discarded by the effect of this constraint are shown in a lighter shade. Clearly, these changes could have been introduced either by writing down directly the constraint or with the help of a graphical interface tool. Allowing this to be performed interactively (perhaps using a slide bar to quickly select values of variables) would give the user an intuition of the relationships among variables. In this example we see that the domains of both `X` and `Z` are affected by this change, and so they depend on `Y`. A more detailed inspection can be done by leaving just one element in the domain of `Y`, and seeing how the range of `X` and `Z` is updated (Figures 12 and 13). In Figure 12, the squares highlighted for `X` and `Z` correspond to all the values which would result from an enumeration of `X` and `Z` for every value of `Y`, put together in a row. Figure 13, on the other hand, represents the idea the solver has about the domains of `X` and `Z` (both figures were computed using the same solver). There are more allowed values for `Z`, due to the approximation the solver keeps internally. The comparison of both graphs gives as well an idea of the accuracy of the solver: a lower accuracy in the update of the variables when adding constraints (which makes this addition cheaper) may not pay off because failure is delayed, or because enumeration becomes more costly.

A static version of this procedure, which shows how two variables are related, can be obtained by plotting their values in a 2-D grid; this is tantamount to choosing values for one of them and looking at the allowed values for the other. This is shown in Figures 14, 15, and 16, where the variables are subject to the constraint **C2**. In every one of these three plots we have chosen to represent a different pair of variables. From these representations we can deduce that the values `X = 3` and `X = 6` are not feasible, regardless the values of `Y` and `Z`. It turns out also that the plots `X` against `Y` and `X` against `Z` (Figures 14 and 15) are identical. From this, one might guess that perhaps `Y` and `Z` have necessarily the same value, i.e., that the constraint `Z = Y` is enforced by the store. This possibility is discarded by Figure 16, in which we see that there are values of `Z` and `Y` which are not the same, and which in fact correspond to different values of `X`. Even more, the slope of the highlighted squares on the grid suggests that there is an inverse relationship between `Z` and `Y`: incrementing one of them would decrement the other.

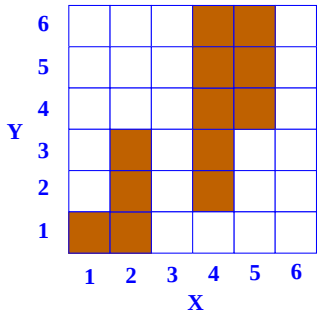


Figure 14: Two variables plotted against each other

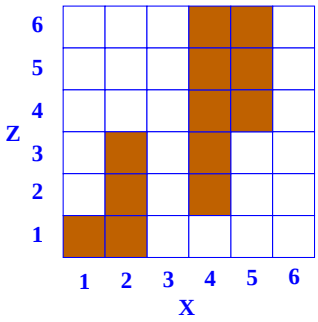


Figure 15: Two variables plotted against each other

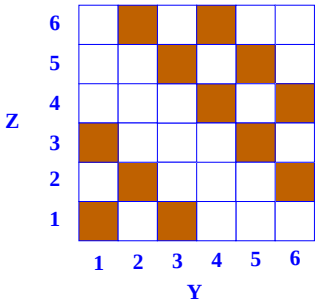


Figure 16: Two variables plotted against each other

In this representation it could be as well possible to change the value of one (or several) variables not plotted in the grid, and examine how this affects the values of the current domains of the variables in the grid. More variables could in principle be depicted at the same time, but the usefulness of such a representation is still unclear: for example, for three variables a 3-D depiction of cubes would be needed; navigating through it (for example, by means of rotations and *virtual tours*), does not pose implementation problems on the graphical side, but it does not seem to give information as intuitively as the 2-D representation.

5 Abstraction

So far we have been concerned with more or less stylized, intuitive representations of program and data states (some of which have been developed and already used elsewhere), but which reflect all the data available (at least, at the level of understanding of the programmer). This can be acceptable for "toy programs", but it is unavoidable that, when a large example with many variables and/or constraints, too much data is to be displayed. This, even if an easy-to-understand depiction is provided, can overwhelm the user with an unwanted level of detail, and with the burden of having to navigate through it.

This can be alleviated by abstracting the information presented. Here, "abstracting" refers to a process which allows a user to focus on interesting properties of the data available. Different abstraction levels and/or techniques can in principle be applied to any of the aforementioned graphical depictions, depending on which property we want to highlight.

In what follows we will present several ideas for performing this abstraction, applied to the graphical representations we have discussed so far. Alternative representations, which are not based on refining others already presented, will be discussed.

6 Abstracting Control

The tree mentioned in Section 2 for representing the search performed gives a good representation of the space being traversed. But it has the drawback of being too explicit, and taking up too much space and

showing too much detail to be useful — medium-sized computations can easily go to, literally, millions of nodes. A means of abstracting this view is desirable.

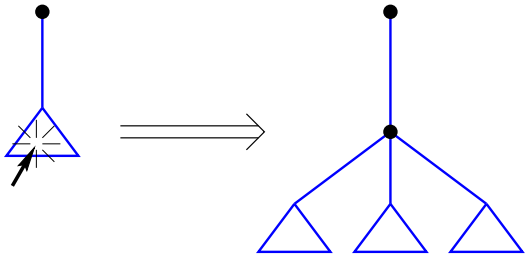


Figure 17: Exposing hidden parts of a tree

An obvious way to cope with a very large number of objects (nodes and links) in the limited space provided by a screen is simply squeezing them to fit into the available space; this can be made uniformly, or with a selection which changes the compression ratio in different parts of the image. The former has the drawback that we lose the capacity to see the details of the execution when necessary. The latter seems more promising, since there might be parts of the tree which the user is not really interested in watching in detail (for example, because they belong to parts of the program which have been already tested, and (s)he is not interested in looking at them).

An example of a tool which compresses part of the search tree is the VISTA tool for the visualization of concurrent logic programs [Tic92]. This compression is performed automatically, at the points of greater density of objects — near the leaves. But this disallows blowing up those parts if a greater detail is needed. An alternative possibility [Mar97] is allowing the user to slide virtual magnifying lenses which provide with a sort of fish-eye transformation and give both a global view (because the whole tree is shrunk to fit in a window) and a detailed view (because selected parts of the tree are zoomed out to greater detail). Providing at the same time a compressed view of the whole search tree, in which the area being zoomed is clearly depicted, can also help to locate the place we are looking at; this option was already present in VisAndOr, where the canvas could be zoomed out, and the window on it was represented as a dotted square in a reduced view of the whole execution (Figure 18.

Another possibility to avoid cluttering up the display is to allow the user to hide parts of the tree (see Figure 17) [Sch97]. This actually allows for the selective exploration of the tree (i.e., in the cases where a call is being made to a predicate known to be correct, or whose performance has already been tested). Whereas this avoids having too many objects at a time, feedback on the relative sizes of the subtrees is lost. It can be recovered, though, by tagging the collapsed subtrees with a mark which measures the relative importance of the subtrees. This "importance" can range from execution time to number of nodes, number of calls, number of added constraints, number of fixpoint steps in the solver, etc.; a different measure would lead to a different abstraction point of view. Possible tagging schemes are raw numbers attached to the collapsed subtrees (indicating the concrete value measured under the subtree) or different shades of gray (which should be automatically re-scaled as subtrees are collapsed/expanded; see Figure 19).

7 Abstracting Values

In real examples a variable can have a very wide range of possible values. A representation as that proposed in Section 3.2 can convey too much information to be really useful: the individual values might be too numerous to be clearly seen, or to allow deducing any structure from them; actually, the screen resolution can be too coarse to assign a pixel to every single value in the domain, thus imposing an aliasing which would prevent faithfully reflecting the structure of the domain of the variable. Zooming out would lead to the same situation found when drawing big execution trees: the need for navigation prevents a perception of the "big picture".

Again, a "fish-eye" technique can be of help, giving the user the possibility of zooming precisely those parts which are of more interest for him/her, while at the same time trying to keep as much information as

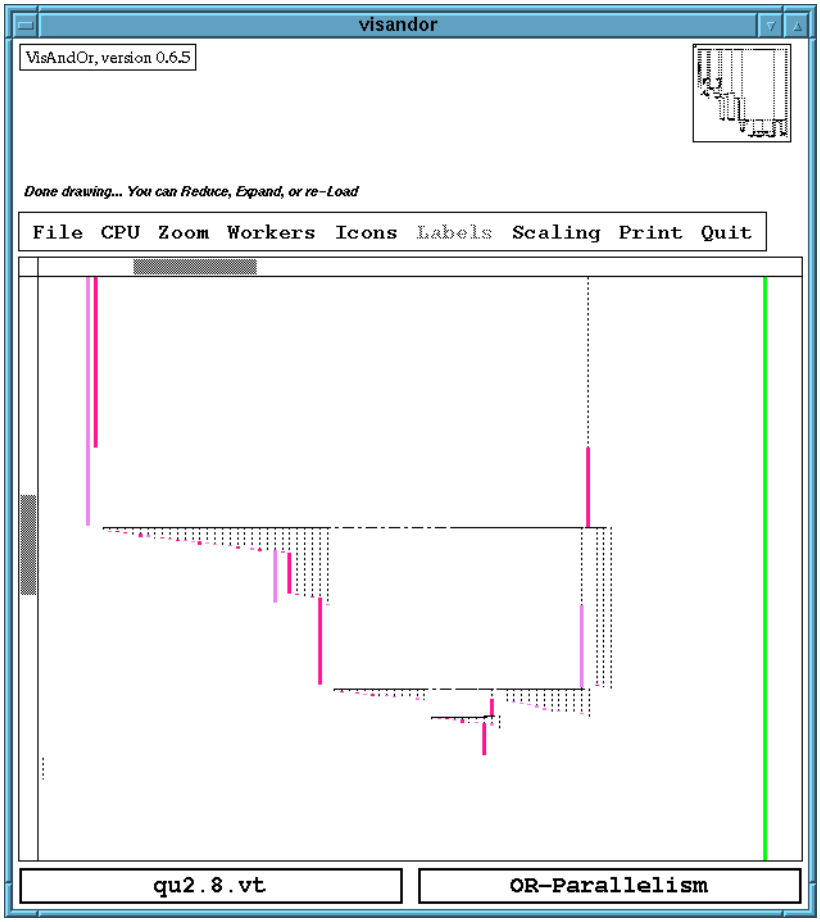


Figure 18: Zoomed view in VisAndOr

possible condensed in a limited space. Similarly, compacting parts of the domain is possible. An automatic compaction could be performed by associating consecutive values in the domain of a variable to an interval (the smallest closing interval) represented by a single point. At a coarser level, a number associated to every variable and representing the number of points in the current domain of the variable could indicate its degree of freedom. This can also be applied to interval variables, using the difference between the maximum and minimum values in its domain, or the total length of the intervals in its domain.

Another alternative for abstraction uses a more application-oriented representation. If some parts of the program are trusted to be correct, the effects of those parts in the constraint store can be masked out by removing them from the representation of the variables, thus leaving less values to be depicted. For example, if a variable is known to take only odd values, the even values are simply filtered and not shown in the representation. In other words, it is the representation of the domain which is changed, to accommodate it to the values the variable can eventually take.

This transformation of the domain space cannot be automatically performed: the debugger has no means of knowing which parts of the program are trusted, and which are not, or which abstraction should be applied to every problem. It has to be told, either with annotations in the program, or by specifying interactively by the user, which constraints should be used to abstract the variable values. Warnings could be issued by the debugger if the values discarded by the program do not correspond to those the user (or the annotations in the program) want to remove: if this happens, a sort of "out of domain" condition can be raised. This condition does not mean necessarily that there is an error: the user may choose not to show uninteresting values which were not (yet) removed by the program.

It may happen that sometimes the user is not really familiar with the program — or that, simply, the

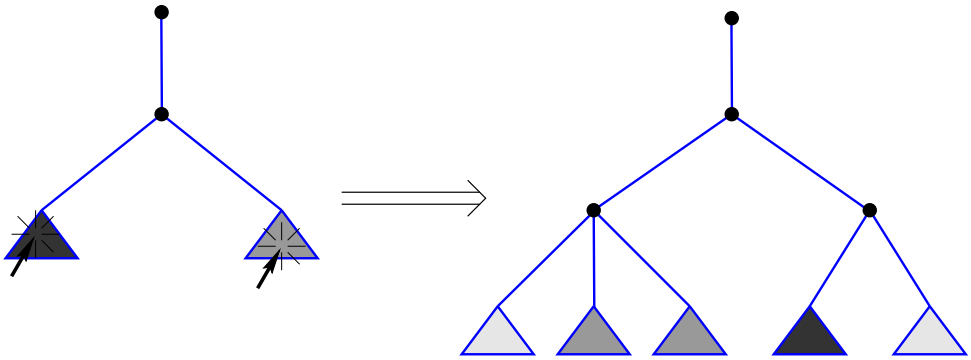


Figure 19: Abstracting parts of a tree

program is so complicated that deciding how to refine the domains of the variables is a hard problem in itself. To help the user in this case, a set of "standard" domain abstractions can be provided with the debugger. Clear candidates for these domain transformations are the already existent domains for abstract interpretation-based analysis (e.g., signs domain, definiteness).

There are other views which are not straightforward to compute (they need definitely low-level support from the solver and the engine), but which may be of help in debugging applications, and are not related with the values of the variables themselves, but rather with the computation environment: for example, the number of constraints that variable is involved in. Exploring these may be an interesting issue.

8 Abstracting Constraints

As the number and complexity of constraints in programs grow, looking at constraints expressed as text becomes useless; if we resort to visualizing constraints depicted as relationships among variables (e.g., 2-D grids or sliding bars to assign values, as suggested in Section 4), we may end up with the same problems we had when trying to represent values of variables, since we are building on top of them. The solutions suggested for the case of representation of values are still valid (fish-eye view, abstraction of domains, ...), and can give an intuition of how a given variable relate to other(s); however, it is not easy to deduce from them how variables are related to each other.

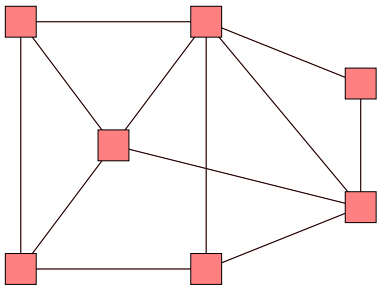


Figure 20: Constraints represented as a graph

A different approach to abstracting the constraints in the store is showing them as a graph, where variables are represented as nodes, and there is a link between two nodes iff the corresponding variables are related by a constraint (Figure 20). This representation gives the programmer a grasp at understanding what are the constraints inside the solver, after the possible partial solving and propagations performed so far. Moreover, since different solvers behave in different ways, this can provide with hints about better ways of setting up constraints for a given program and constraint solver.

The topology of the graph can be used to decide whether a reorganization of the program is advantageous; for example, if there are subsets of nodes in the graph with a high degree of connectivity, but those subsets are sparsely connected among them, it might be worth setting up the highly connected sections before, to favor a more local constraint propagation, and then link the different regions, in order to solve locally as many constraints as possible. For this, of course, a means of accessing the location in the program of the variables which appear depicted in the graph is needed. This can as well help to discover unwanted constraints among variables — or the lack of them.

More information can be embedded in this graph representation: for example, weights can be added to the links in order to represent on the graph various metrics related to aspects of the constraint store we want to know: the number of times there has been propagation between two variables [Stu97], or the number of constraints relating them. The weights themselves need not be expressed as numbers attached to the edges, but can take instead a visual form: they can be expressed, for example, as different degrees of thickness or shades of color.

Variables can also have attached a tag which gives visual feedback about interesting features: for example, the actual range of the variable, or the number of constraints (if it is not clear from the number of edges departing from it) it is involved in, or the number of times its domain has been updated.

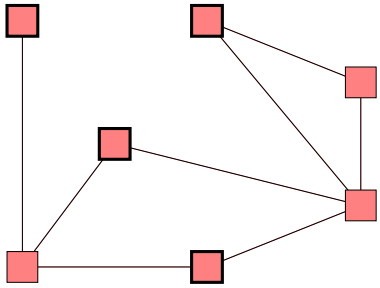


Figure 21: Variables with bold frames acquired a definite value

If the graph is animated, as the solver progresses towards a solution, the picture displayed can change dynamically as well. This can reflect, for example, propagation taking place between variables, or how the variables lose their links (constraints) with other variables as they become ground,[4] acquiring a definite value. In Figure 21 some variables became ground, and, while variables which do not have yet definite values keep their constraints with ground variables, the constraints between ground variables are not shown any more. The reason for doing that is that those constraints are not useful any longer and, in fact, it reflects the idea of a system being progressively simplified. This helps as well to seeing how backtracking is performed. When that happens, either the links appear again (when a point where a variable became ground is backtracked over, and a constraint is back active in the store), or they disappear (when the system backtracks past a point where a constraint was created).

In any of the above cases we might as well filter which kind of constraints are shown: for example, only linear constraints, or only constraints involving FD variables, or only constraints of the form Exp_1 $\langle relationship \rangle$ Exp_2, for any $\langle relationship \rangle$ supported by (a subset of) the source language.

9 Conclusions

We have presented several graphical representations aimed at a better understanding of the execution of constraint logic programs, both from the point of view of the program and the data. This understanding will eventually result in an improved capability to pinpoint where possible bugs are located in a program.

To this end, we have focused on the visualization of three aspects: the execution flow of the program, the runtime values of the variables, and the constraints. Each one of them has unique characteristics in constraint logic programming, and a good graphical representation needs taking into account what is the feedback really useful for the programmer. We have devised visualization paradigms for these three views,

[4] If this representation is actually the desired one.

keeping them intimately related, so that tools implementing them can be used together in a complementary way.

Since a potential problem in program visualization is the huge amount of data to be represented, we have also tackled the problem of abstracting this data in such a way that interesting properties and are retained. Different abstraction techniques, intended to preserve and highlight some properties, were presented.

Bibliography

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[BDM97] R. Baecker, C. DiGiano, and A. Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

[Byr80] L. Byrd. Understanding the control flow of prolog programs. In S.-A. Tärnlund, editor, *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[DNTM88] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. pages 573–581, 1988.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In (H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[EB88] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 1988.

[Kah96] K. Kahn. Drawing on napkins, video-game animation, and other ways to program computers. *Communications of the ACM*, 39(8):49–59, August 1996.

[Lue97] Angel López Luengo. APT: implementación de un visualizador gráfico de la ejecución de prolog. Master's thesis, T.U. of Madrid (UPM), Facultad de Informática, Madrid, 28660, 1997. In Preparation.

[Mar97] Kim Marriot. Personnal communication, September 1997.

[Mei96] M. Meier. Grace User Manual, 1996. Available at `http://www.ecrc.de/eclipse/html/grace/grace.html`.

[Sch97] Christian Schulte. Oz Explorer: A Visual Constraint Programming Tool. In Lee naish, editor, *ICLP'97*. MIT Press, July 1997.

[Stu97] Peter Stuckey. Personnal communication, September 1997.

[Tic92] Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.