11. Tools for Constraint Visualization: The VIFID/TRIFID Tool

M. Carro and M. Hermenegildo

Technical University of Madrid, 28660-Madrid, Spain.
{mcarro,herme}@fi.upm.es

Summary.

Visualization of program executions has been used in applications which include education and debugging. However, traditional visualization techniques often fall short of expectations or are altogether inadequate for new programming paradigms, such as Constraint Logic Programming (CLP), whose declarative and operational semantics differ in some crucial ways from those of other paradigms. In particular, traditional ideas regarding the behavior of data often cannot be lifted in a straightforward way to (C)LP from other families of programming languages. In this chapter we discuss techniques for visualizing data evolution in CLP. We briefly review some previously proposed visualization paradigms, and also propose a number of (to our knowledge) novel ones. The graphical representations have been chosen based on the perceived needs of a programmer trying to analyze the behavior and characteristics of an execution. In particular, we concentrate on the representation of the run-time values of the variables, and the constraints among them. Given our interest in visualizing large executions, we also pay attention to abstraction techniques, i.e., techniques which are intended to help in reducing the complexity of the visual information.

11.1 Introduction

As mentioned in the previous chapter, program visualization has focused classically on the representation of program flow (using flowcharts and block diagrams, for example) or on the data manipulated by the program and its evolution as the program is executed. In that chapter we discussed issues related to the visualization of the "programmed control" or "programmed search" part of the execution of constraint logic programs [11.17, 11.19]. In this chapter we focus on methods for displaying the contents of variables, the constraints among such variables, the evolution of such contents and constraints, and abstractions of the proposed depictions. For simplicity, and because of their relevance in practice, in what follows we will discuss mainly the representation of finite domain (FD) constraints and variables, although we will also mention other constraint domains and consider how the visualizations designed herein can be applied to them.

11.2 Displaying Variables

In imperative and functional programming there is a clear notion of the values that variables are bound to (although it is indeed more complex in the case of higher-order functional variables). The concept of variable binding in LP is somewhat more complex, due to the variable sharing which may occur among Herbrand terms. The problem is even more complex in the case of CLP, where such sharing is generalized to the form of equations relating variables. As a result, the value of C(L)P variables often is actually a complex object representing the fact that each variable can take a (potentially infinite) set of values, and that there are constraints attached to such variables which relate them and which restrict the values they can take simultaneously.

Textual representations of the variables in the store are usually not very informative and difficult to interpret and understand.[1] A graphical depiction of the values of the variables can offer a view of computation states that is easier to grasp. Also, 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 be either animated (i.e., time in the program is depicted in the visualization also as time) or laid out spatially as a succession of pictures. The latter allows comparing different behaviors easily, trading time for space.

Since different constraint domains have different properties and characteristics, different representations for variables may be needed for them. In what follows we will sketch some ideas focusing on the representation of variables in Finite Domains, but we will also refer briefly to the depiction of other commonly used domains.

11.2.1 Depicting Finite Domain Variables

As mentioned before, Finite Domains (FD) are one of the most popular constraint domains. FD variables take values over finite sets of integers which are the domains of such variables. The operations allowed among FD variables are pointwise extensions of common integer arithmetic operations, and the allowed constraints are the pointwise variants of arithmetic constraints. At any state in the execution, each FD variable has an active domain (the set of allowed values for it) which is usually accessible by using primitives of the language. For efficiency reasons, in practical systems this domain is usually an upper approximation of the actual set of values that the variable can theoretically take. We will return to this characteristic later, and we will see how taking it into account is necessary in order to obtain correct depictions of values of variables.

A possible graphical representation for the state of FD variables is to assign a dot (or, depending on the visualization desired, a square) to every

[1] Also note that some solvers maintain, for efficiency or accuracy reasons, only an approximation of the values the variables can take.

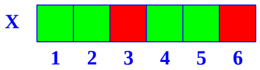


Fig. 11.1. Depiction of a finite domain variable

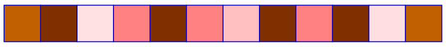


Fig. 11.2. Shades representing age of discarded values

possible value the variable can take; therefore the whole domain is a line (respectively, a rectangle). Values belonging to the current domain at every moment are highlighted. An example of the representation of a variable X with current domain $\{1, 2, 4, 5\}$ from an initial domain $\{1 \ldots 6\}$ is shown in Figure 11.1. More possibilities include using different colors / shades / textures to represent more information about the values, as in Figure 11.2 (this is done also, for example, in the GRACE visualizer [11.20]).

Looking at the static values of variables at only one point in the execution (for example, the final state) obviously does not provide much information on how the execution has actually progressed. However, the idea is that such a representation can be associated with each of the nodes of the control tree, as suggested in Chapter 10, i.e., the window that is opened upon clicking on a node in the search-tree contains a graphical visualization for each of the variables that are relevant to that node. The variables involved can be represented in principle simply side to side as in Figure 11.6 (we will discuss how to represent the relations between variables, i.e., constraints, in Section 11.3).

Note that each node of the search-tree often represents several internal steps in the solver (e.g., propagation is not seen by the user, or reflected in user code). The visualization associated to a node can thus represent either the final state of the solver operations that correspond to that search-tree node, or the history of the involved variables through all the internal solver (or enumeration) steps corresponding to that node.

Also, in some cases, it may be useful to follow the evolution of a set of program variables throughout the program execution, independently of what node in the search-tree they correspond to (this is done, for example, in some of the visualization tools for CHIP [11.1]). This also requires a depiction of the values of a set of variables over time, and the same solutions used for the previous case can be used.

Thus, it is interesting to have some way of depicting the evolution in time of the values of several variables. A number of approaches can be used to achieve this:

– An animated display which follows the update of the (selected) variables step by step as it happens; in this case, time is represented as time. This makes the immediate comparison of two different stages of the execution difficult, since it requires repeatedly going back and forth in time. However, the advantage is that the representation is compact and can be useful for understanding how the domains of the variables are narrowed. We will return to this approach later.

– Different shadings (or hues of color) can be used in the boxes corresponding to the values, representing in some way how long ago that value has been removed from the domain of the variable (see Figure 11.2, where darker squares represent values removed longer ago). Unfortunately, comparing shades accurately is not easy for the human eye, although it may give a rough and very compact indication of the changes in the history of the variable. An easier to interpret representation would probably involve adjusting the shades so that the human brain interprets them correctly when squares of different shades surround it.
– A third solution is to simply stack the different state representations, as in Figure 11.3. This depiction can be easily shrunk/scrolled if needed to accommodate the whole variable history in a given space. It can represent time accurately (for example, by reflecting it in the height between changes) or ignore it, working then in *events space* (see Chapter 10), by simply stacking a new line of a constant height every time a variable domain changes, or every time an enumeration step is performed. This representation allows the user to perform an easier comparison between states and has the additional advantage of allowing more time-related information to be added to the display.

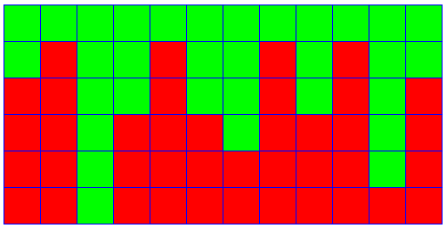


Fig. 11.3. History of a single variable (same as in Figure 11.2)

The last approach is one of the visualizations available in the VIFID visualizer, implemented at UPM, and which, given a set of variables in a FD program, generates windows which display states (or sets of states) for those variables. VIFID can be used as a visualizer of the state in nodes of the search-tree, or standalone, as a user library, in which case the display is triggered by spy-points introduced by the user in the program. Figure 11.15 shows an example of such an annotated program, where the lines calling `log_state/1` implement the spy-points, and Figure 11.4 shows a screen dump of a window generated by VIFID presenting the evolution of selected program variables in a program to solve the queens problem for a board of size 10. Each column in the display corresponds to one program variable, and are labeled with the name of the variables on top. In this case the possible values are the row numbers in which a queen can be placed. Lighter squares represent values still in the domain, and darker squares represent discarded values.

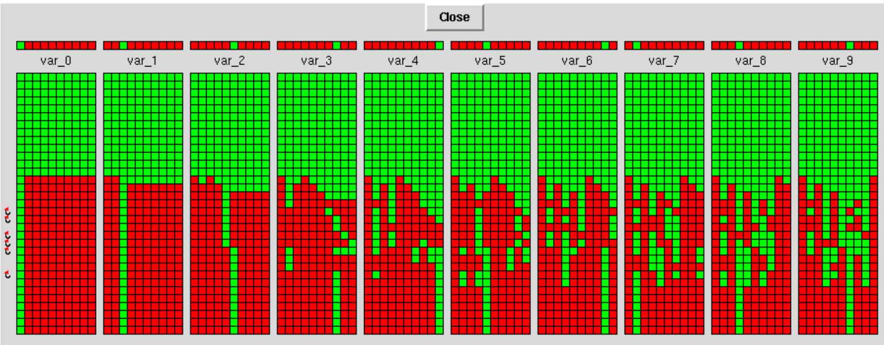


Fig. 11.4. Evolution of FD variables for a 10 queens problem

Each row in the display corresponds to a spy-point in the source program, which caused VIFID to consult the store and update the visualization. Points where backtracking took place are marked with small curved arrows pointing upwards. It is quite easy to see that very little backtracking was necessary, and that variables are highly constrained, so that enumeration (proceeding left to right) quite quickly discarded initial values. VIFID supports several other visualizations, some of which will be presented later in the chapter.

Some of the problems which appear in a display of this type are the possibly large number of variables to be represented and the size of the domain of each variable. Note that the first problem is under control to some extent in the approach proposed: if the visualization is simply triggered from a selected node in the search-tree, the display can be forced to present only the relevant variables (e.g., the ones in the clause corresponding to that node). In the case of triggering the visualization through spy-points in the user program, the number of variables is under user control, since they are selected explicitly when introducing the spy-points. The size of the domains of variables is more difficult to control (we return to this issue in Section 11.4.1). However, note that, without loss of generality, programs using FD variables can be assumed to initialize the variables to an integer range which includes all the possible values allowable in the state corresponding to the beginning of the program.[2] However, being able to deduce a small initial domain for a variable allows starting from a more compact initial representation for that variable. This in turn will allow a more compact depiction of the narrowing of the range of the variable, and of how values are discarded as the execution proceeds. Other abstraction means for coping with large executions are discussed in Section 11.4.1.

[2] In the default case, variables can be assumed to be initialized to the whole domain.

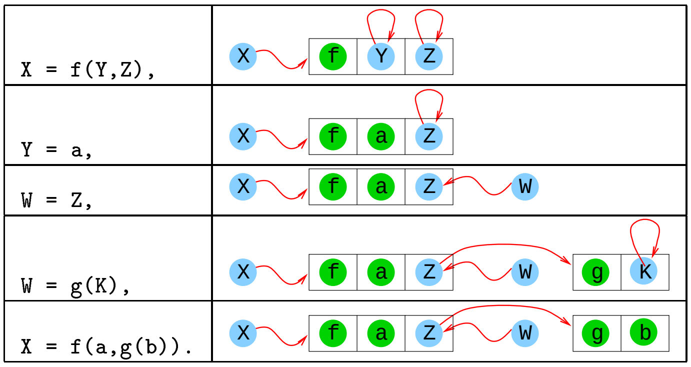


Fig. 11.5. Alternative depiction of the creation of a Herbrand term

11.2.2 Depicting Herbrand Terms

Herbrand terms can always be written textually, or with a slightly enhanced textual representation. An example is the depiction of nodes in the APT tool, discussed in Chapter 10. They can also be represented graphically, typically as trees. A term whose main functor is of arity n is then represented as a tree in which the root is the name of this functor, and the n subtrees are the trees corresponding to its arguments. This representation is well suited for ground terms. However, free variables, which may be shared by different terms, need to be represented in a special way. A possibility is to represent 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 free variable to point to itself. This corresponds to a view of Herbrand terms as complex data structures with single assignment pointers. Figure 11.5 shows a representation using this view of the step by step creation of a complex Herbrand term by a succession of Herbrand constraints. Rational trees (as those supported by Prolog II, III, IV) can also be represented in a similar way—but in this case the graph can contain cycles, although it cannot be a general graph.

11.2.3 Depicting Intervals or Reals

In a broad sense, intervals resemble finite domains: the constraints and operations allowed in them are analogous (pointwise extensions of arithmetic operations), but the (theoretical) set of values allowed is continuous, which means that an infinite set of values are possible, even within a finite range. Despite these differences, visual representations similar to those proposed for finite domains can be easily used for interval variables, using a continuous line instead of a discrete set of squares. An important difference between intervals and finite domains is that intervals usually allow non-linear arithmetic operations for which a solution procedure is not known, which forces the solvers

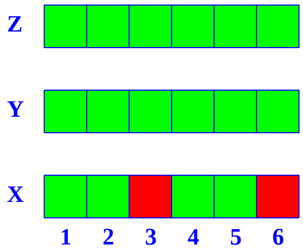


Fig. 11.6. Several variables side to side

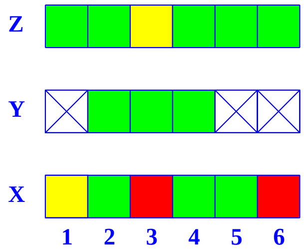


Fig. 11.7. Changing a domain

to be incomplete. Thus, the visualization of the actual domain[3] will in general be an upper approximation of the actual (mathematical) domain. As a result, an exact display of the intervals is not possible in practice. But the approach for showing the evolution in time (Figure 11.4) and for representing the constraints (Section 11.3) is still valid, although in the former case some means for dealing with phenomena inherent to solving in real-valued intervals (e.g., slow convergence of algorithms) should be taken.

11.3 Representing Constraints

In the previous section we have dealt with representations of the values of individual variables. It is obviously also interesting to represent the relationships among several variables as imposed by the constraints affecting them. This can sometimes be done textually by simply dumping the constraints and the variables involved in the source code representation. Unfortunately, this is often not straightforward (or even possible in some constraint domains), can be computationally expensive, and provides too much level of detail for an intuitive understanding.

Constraint visualization can be used alternatively to provide information about which variables are interrelated by constraints, and how these interrelations make those variables affect each other. Obviously, classical geometric representations are a possible solution: for example, linear constraints can be represented geometrically with dots, lines, planes, etc., and nonlinear ones by curves, surfaces, volumes, etc. Standard mathematical packages can be used for this purpose. However, these representations are not without problems: working out the representation can be computationally expensive, and, due to the large number of variables involved the representations can easily be n-dimensional, with $n \gg 3$.

A general solution which takes advantage of the representation of the actual values of a variable (and which is independent of how this representation

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

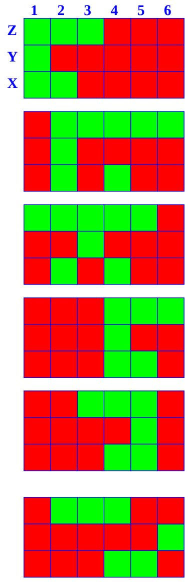


Fig. 11.8. Enumerating `Y`, representing solver domains `X` and `Z`

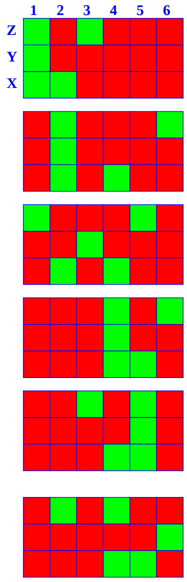


Fig. 11.9. Enumerating `Y`, representing also the enumerated domains for `X` and `Z`

is actually performed) is to use projections to present the data piecemeal and to allow the user to update the values of the variables that have been projected out, while observing how the variables being represented are affected by such changes. This can often provide the user with an intuition of the relationships linking the variables (and detect, for example, the presence of erroneous constraints). The update of these variables can be performed interactively by using the graphical interface (e.g., via a sliding bar), or adding manually a constraint, using the source CLP language.

We will use the constraint **C1**, below, in the examples which follow:

$$\mathbf{C1} : X \in \{1..6\} \wedge X \neq 6 \wedge X \neq 3 \wedge Z \in \{1..6\} \wedge Z = 2X - Y \wedge Y \in \{1..6\} \quad (11.1)$$

Figure 11.6 shows the actual domains of FD variables `X`, `Y`, and `Z` subject to the constraint **C1**. As before, lighter boxes represent points inside the domain of the variable, and darker boxes stand for values not compatible with the constraint(s). This representation allows the programmer to explore how changes in the domain of one variable affect the others: an update of the domain of a variable should indicate changes in the domains of other variables related to it. For example, we may discard the values 1, 5, and 6 from the domain of `Y`, which boils down to representing the constraint **C2**:

$$\mathbf{C2} : \mathbf{C1} \wedge Y \neq 1 \wedge Y < 5 \tag{11.2}$$

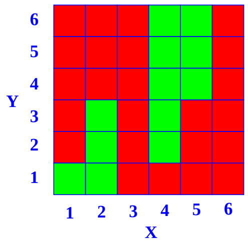


Fig. 11.10. X against Y

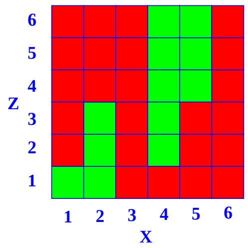


Fig. 11.11. X against Z

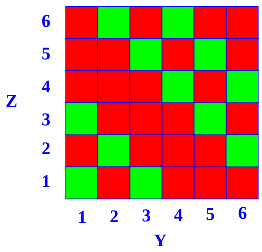


Fig. 11.12. Y against Z

Figure 11.7 represents the new domains of the variables. Values directly disallowed by the user are shown as crossed boxes; values discarded by the effect of this constraint are shown in a lighter shade. In this example the domains of both `X` and `Z` are affected by this change, and so they depend on `Y`. This type of visualization (with the two enumeration variants which we will comment on in the following paragraphs) is also available in the VIFID tool.

Within this same visualization, a more detailed inspection can be done by leaving just one element in the domain of `Y`, and watching how the domains of `X` and `Z` are updated. In Figures 11.8 and 11.9 `Y` is given a definite value from `1` (in the topmost rectangle) to `6` (in the bottommost one). This allows the programmer to check that simple constraints hold among variables, or that more complex properties (e.g., that a variable is made definite by the definiteness of another one) are met.

The difference between the two figures lies in how values are determined to belong to the domain of the variable. In Figure 11.8, the values for `X` and `Z` are those kept internally by the solver, and are thus probably a safe approximation. In Figure 11.9, the corresponding values were obtained by enumerating `X` and `Z`, and the domains are smaller. Both figures were obtained using the same constraint solver, and comparing them gives an idea of how accurately the solver keeps the values of the variables. For several reasons (limitation of internal representation, speed of addition/removal of constraints, etc), quite often solvers do not keep the domains of the variables as accurately as it is possible. Overconstraining a problem may then help in causing an earlier failure. On the other hand, overconstraining increments the number of constraints to be processed and the time associated to this processing. Comparing the solver-based against the enumeration-based representation of variables helps in deciding whether there is room for improvement by adding redundant constraints.

A static version of this view can be obtained by plotting values of pairs of variables in a 2-D grid, which is equivalent to choosing values for one of

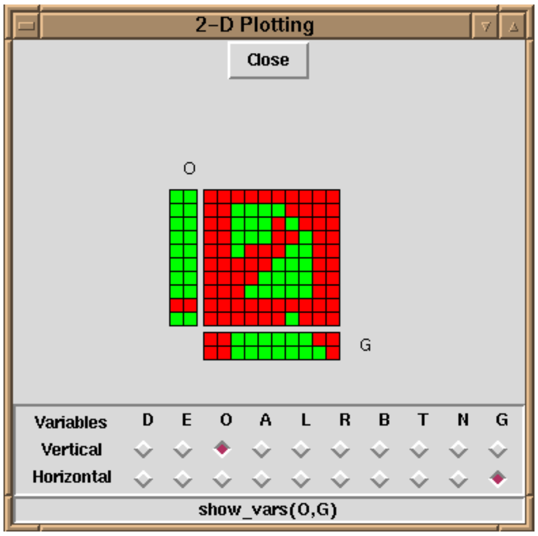


Fig. 11.13. Relating variables in VIFID

them and looking at the allowed values for the other. This is schematically shown in Figures 11.10, 11.11, and 11.12, where the variables are subject to the constraint **C2**. In each of these three figures we have represented 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 of `X` against `Y` and `X` against `Z` (Figures 11.10 and 11.11) 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 11.12, 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`. Furthermore, 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 presumably decrement the other—and this is actually the case, from constraint **C1**. A VIFID window showing a 2-D plot appears in Figure 11.13; the check buttons at the bottom allow the user to select the variables to depict.

Note that, in principle, more than two variables could be depicted at the same time: for example, for three variables a 3-D depiction of a "Lego object" made out of cubes could be used. Navigating through such a representation (for example, by means of rotations and *virtual tours*), does not pose big implementation problems on the graphical side, but it may not necessarily give information as intuitively as the 2-D representation. The usefulness of such a 3-D (or n-D) representation is still a topic of further research—but 3-D portraits of other representations are possible; see Section 11.4.2. On the other hand, we have found very useful the possibility of changing the value of one (or several) variables not plotted in the 2-D grid, and examine how this affects the values of the current domains of the plotted variables.

11.4 Abstraction

While representations which reflect all the data available in an execution can be acceptable (and even didactic) for "toy programs," it is often the case that they result in too much data being displayed for larger programs. Even if an easy-to-understand depiction is provided, the amount of data 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 is to be highlighted.

Note that the depictions presented so far already incorporate some abstractions: when using VIFID, the user selects the interesting variables and program points via the spy-points and the window controls. If it is interfaced with a tree representation tool (as, for example, the one presented in Chapter 10), the variables to visualize come naturally from those in the selected nodes. In what follows we will present several other ideas for performing abstraction, applied to the graphical representations we have discussed so far. Also, some new representations, which are not directly based on a refinement of others already presented, will be discussed.

11.4.1 Abstracting Values

While the problem of the presence of a large number of variables can be solved, at least in part, by the selection of interesting variables (a task that is difficult in itself), another problem remains: in the case of variables with a large number of possible values, representations such as those proposed in Section 11.2.1 can convey information too detailed to be really useful. At the limit, the screen resolution may be insufficient to assign a pixel to every single value in the domain, thus imposing an aliasing effect which would prevent reflecting faithfully the structure of the domain of the variable. This is easily solved by using standard techniques such as a canvas that is larger than the window, and scrollbars, providing means for zooming in and out, etc. A "fish-eye" technique can also be of help, giving the user the possibility of zooming precisely those parts which are more interesting, while at the same time trying to keep as much information as possible condensed in a limited space. However, these methods are more "physical" approaches than true conceptual abstractions of the information, which are richer and more flexible.

An alternative is to perform a more semantic "compaction" of parts of the domain. As an example of such a compaction, which can be performed automatically, consider associating consecutive values in the domain of a variable to an interval (the smallest one enclosing those values) and representing this interval by a reduced number of points. A coarser-level solution, complementary to the graphical representation, is to present the domain of a variable simply as a number, denoting how many values remain in its current domain, thus providing an indication of its "degree of freedom". A similar approach can be applied to interval variables, using the difference between the maximum and minimum values in their domains, or the total length of the intervals in their domains.

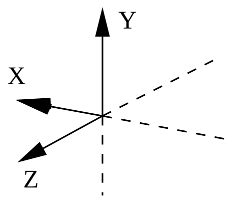
Another alternative for abstraction is to use an application-oriented filtering of the variable domains. For example, if some parts of the program are trusted to be correct, their effects in the constraint store can be masked out by removing the values already discarded from the representation of the variables, thus leaving less values to be depicted. E.g., if a variable is known to take only odd values, the even values are simply not shown in the representation. This filtering can be specified using the source language—in fact, the constraint which is to be abstracted should be the filter of the domain of the displayed variables.

Note that this transformation of the domain cannot be completely automated easily: the debugger may not have any way of knowing which parts of the program are trusted and which are not, or which abstraction should be applied to a given problem. Thus, the user should indicate, with annotations in the program [11.9, 11.4] or interactively, which constraints should be used to abstract the variable values. Given this information, the actual reduction of the representation can be accomplished automatically. Warnings could be issued by the debugger if the values discarded by the program do not correspond to those that 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.

11.4.2 Domain Compaction and New Dimensions

Besides the problems in applications with large domains, the static representations of the history of the execution (Figure 11.4) can also fall short in showing intuitively how variables converge towards their final values, again because of the excess of points in the domains, or because an execution shows a "chaotic" profile. The previously proposed solution of using the *domain size* as an abstraction can be applied here too. However, using raw numbers directly in order to represent this abstraction to the user is not very useful because it is not easy for humans to visualize arrays of numbers. A possible solution is to resort to shades of gray, but this may once again not work too well in practice: deducing a structure from a picture composed of different levels of brightness is not straightforward, and the situation may get even worse if colors are added.

A better option is to use the number of active values in the domain as coordinates in an additional dimension, thus leading to a 3-D visualization. A possible meaning of each of the dimensions in such a representation appears in Figure 11.14. As in Figure 11.4, two axes correspond to time and selected variables: time runs along the **Z** direction, and every row along this dimension corresponds to a snapshot of the set of FD variables which have been selected for visualization. In each of these rows, the size of the domain of the variable (according to the internal representation of the solver) is depicted as the dimension **Y**.



X The CLP(FD) variables.

Y An abstraction of the variable: the size of its domain.

Z Time.

Fig. 11.14. Meaning of the dimensions in the 3-D representation.

Figure 11.15 shows a CLP(FD) program for the `DONALD + GERALD = ROBERT` puzzle; we will inspect the behavior of this program using two different orderings, defined by the `order/3` predicate. A different series of choices concerning variables and values (and, thus, a different search-tree) will be generated by using each of these options. The program (including the labeling routines) was annotated with calls to predicates which act as spy-points, and log the sizes of the domains of each variable (and, maybe, other information pertaining to the state of the program) at the time of each call. This information is unaffected by backtracking, and thus it can also keep information about the choices made during the execution. The `Handle` used to log the data may point to an internal database, an external file, or even a socket-based connection for on-line visualization or even remote debugging.

Figure 11.16 is an execution of the program in Figure 11.15, using the first ordering of the variables. The variables closer to the origin (the ones which were labeled first) are assigned values quite soon in the execution and they remain fixed. But there are backtracking points scattered along the execution, which appear as blocks of variables protruding out of the picture. There is also a variable (which can be viewed as a white strip in the middle of the picture) which appears to be highly constrained, so that its domain is reduced right from the beginning. That variable is probably a good candidate to be labeled soon in the execution. Some other variables apparently have a high interdependence (at least, from the point of view of the solver), because

```
:- use_module(library(clpfd)).
:- use_module(library(visclpfd)).

dgr(WhichOrder, ListOfVars):-
    ListOfVars = [D,O,N,A,L,G,E,R,B,T]
    order(WhichOrder, ListOfVars, OrderedVars),         %% Added
    open_log(dgr, ListOfVars, Handle),                  %% Added
    domain(OrderedVars, 0, 9),
    log_state(Handle),                                  %% Added
    D #> 0,
    log_state(Handle),                                  %% Added
    G #> 0,
    log_state(Handle),                                  %% Added
    all_different(OrderedVars),
    log_state(Handle),                                  %% Added
    100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
    100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
    100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
    log_state(Handle),                                  %% Added
    user_labeling(OrderedVars, Handle),
    close_log(Handle).

order(1, [D,O,N,A,L,G,E,R,B,T], [D,G,R,O,E,N,B,A,L,T]).
order(2, [D,O,N,A,L,G,E,R,B,T], [G,O,B,N,E,A,R,L,T,D]).
```

Fig. 11.15. The annotated `DONALD + GERALD = ROBERT` FD program.

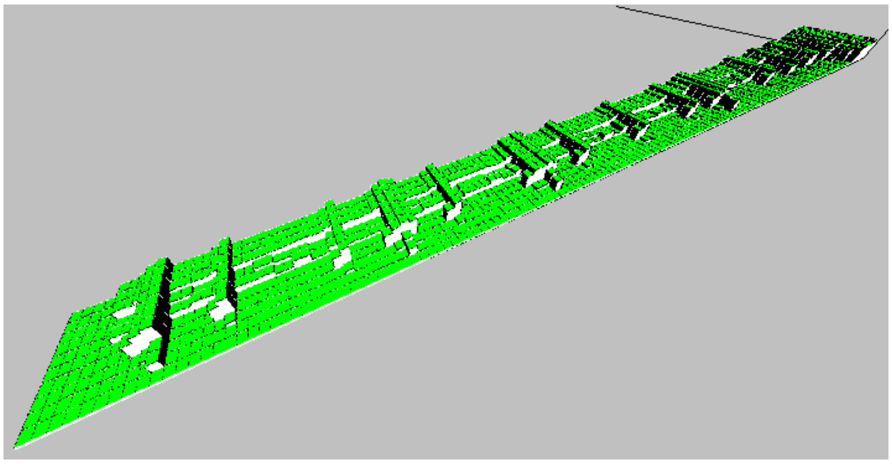


Fig. 11.16. Execution of the `DONALD + GERALD = ROBERT` program, first ordering.

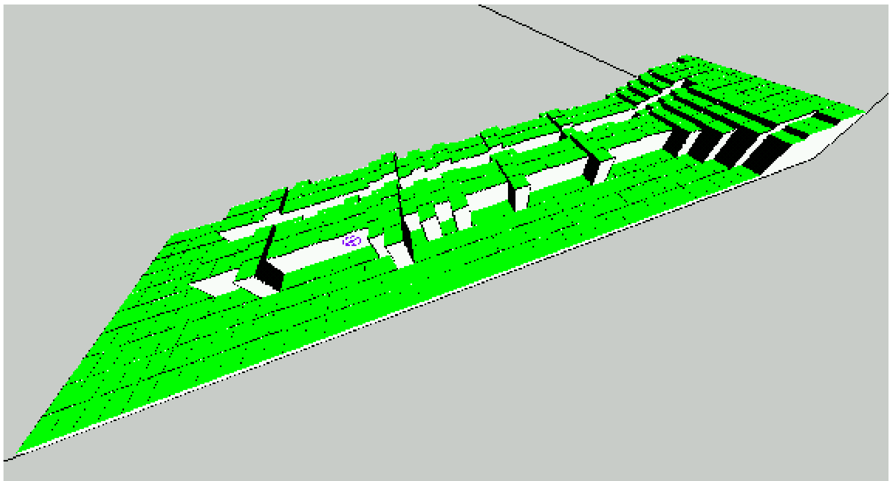


Fig. 11.17. Execution of the `DONALD + GERALD = ROBERT` program, second ordering

in case of backtracking, the change of one of them affects the others. This suggests that the behavior of the variables in this program can be classified into two categories: one with highly related variables (those whose domains change at once in the case of backtracking) and a second one which contains variables relatively independent from those in the first set.

Another execution of the same program, using the second ordering, yields the profile shown in Figure 11.17. Compared to the first one, there are fewer execution steps, but, of course, the classification of the variables is the same: the whole picture has the same general layout, and backtracking takes place in blocks of variables.

These figures have been generated by a tool, TRIFID, integrated into the VIFID environment. They were produced by processing a log created at run-time to create a VRML depiction with the ProVRML package [11.24], which allows reading and writing VRML code from Prolog, with a similar approach to the one used by PiLLoW [11.7]. One advantage of using VRML is that sophisticated VRML viewers are readily available for most platforms. The resulting VRML file can be loaded into such a viewer and rotated, zoomed in and out, etc. Additionally, the log file is amenable to be post-processed using a variety of tools to analyze and discover characteristics of the execution, in a similar way as in [11.16]. Another reason to use VRML is the possibility of using hyper-references to add information to the depiction of the execution without cluttering the display. In the examples shown, every variable can be assigned a hyperlink pointing to a description of the variable. This description may contain pieces of information such as the source name of the variable, the actual size of its domain at that time, a profile of the changes undergone by that particular variable during the execution, the number of times its domain

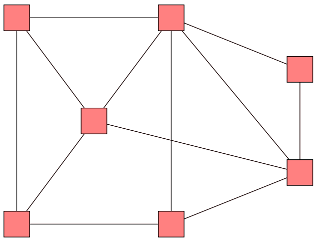


Fig. 11.18. Constraints represented as a graph

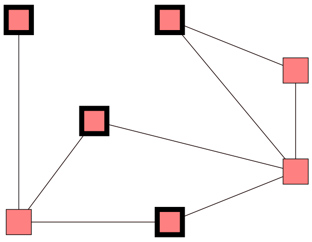


Fig. 11.19. Bold frames represent definite values

has been updated, the number of times backtracking has changed its domain, etc. Using the capability of VRML for sending and receiving messages, and for acting upon the receipt of a message, it is possible to encode in the VRML scene an abstraction of the propagation of constraints as it takes place in the constraint solver, in a similar way as the variable interactions depicted by VIFID.

11.4.3 Abstracting Constraints

As the number and complexity of constraints in programs grow, if we resort to visualizing them as relationships among variables (e.g., 2-D or 3-D grids plus sliding bars to assign values for other variables, as suggested in Section 11.3), we may end up with the same problems we faced when trying to represent values of variables, since we are building on top of the corresponding representations. 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 relates to others. However, it is not always easy to deduce from them how variables are related to each other, due to the lack of accuracy (inherent to the abstraction process) in the representation of the variables themselves.

A different approach to abstracting the constraints in the store is to show them as a graph (see, e.g., [11.22] for a formal presentation of such a graph), where variables are represented as nodes, and nodes are linked iff the corresponding variables are related by a constraint (Figure 11.18)[4]. This representation provides the programmer with an approximate understanding of the constraints that are present in the solver (but not exactly *which* constraints they are), after the possible partial solving and propagations performed up to that point. Moreover, since different solvers behave in different ways, this can provide hints about better ways of setting up constraints for a given program and constraint solver.

[4] This particular figure is only appropriate for binary relationships; constraints of higher arity would need hypergraphs.

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 loosely connected among them, it may be worth to set up the tightly connected sections and making a (partial) enumeration early, to favor more local constraint propagation, and then link (i.e., set up constraints) the different regions, thus solving first locally as many constraints as possible. In fact, identifying sparsely connected regions can be made in an almost automatic fashion by means of clustering algorithms. For this to be useful, a means of accessing the location in the program of the variables which appears depicted in the graph is needed. This can as well help discover unwanted constraints among variables—or the lack of them.

More information can be embedded in this graph representation. For example, weights in the links can represent various metrics related to aspects of the constraint store such as the number of times there has been propagation between two variables 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 shown, for example, as different degrees of thickness or shades of color. Variables can also have a tag attached 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.

The picture displayed can be animated and change as the solver proceeds. This can reflect, for example, propagation taking place between variables, or how the variables lose their links (constraints) with other variables as they acquire a definite value. In Figure 11.19 some variables became definite, and as a result the constraints between them are not shown any more. The reason for doing so is that those constraints are not useful any longer: this reflects the idea of a system being progressively simplified. It may also help to visualize how backtracking is performed: when backtracking happens, either the links reappear (when a point where a variable became definite is backtracked over and a constraint is active again in the store), or they disappear (when the system backtracks past a point where a constraint was created).

Further filtering can be accomplished by selecting which types of constraints are to be represented (e.g, represent only “greater than” constraints, or certain constraints flagged in the program through annotations). This is quite similar to the domain filtering proposed in Section 11.4.1.

11.5 Conclusions

We have discussed techniques for visualizing data evolution in CLP. The graphical representations have been chosen based on the perceived needs of a programmer trying to analyze the behavior and characteristics of an execution. We have proposed solutions for the representation of the run-time values of the variables and of the run-time constraints. In order to be able to deal with large executions, we have also discussed abstraction techniques, including the 3-D rendition of the evolution of the domain size of the variables. The proposed visualizations for variables and constraints have been tested using two prototype tools: VIFID and TRIFID. These visualizations can be easily related, so that tools based on them can be used in a complementary way, or integrated in a larger environment. In particular, in the environment that we have developed, each tool can be used independently or they can all be triggered from a search-tree visualization (e.g., APT).

VIFID (and, to a lesser extent, TRIFID which is less mature) has evolved into a practical tool and is are quite usable by itself as a library which can be loaded into a number of CLP systems. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed by Cosytec for the CHIP system, and which are described in other chapters.

Acknowledgements

The authors would like to thank the anonymous referees for their constructive comments. Also thanks to Abder Aggoun and Helmut Simonis and other DiSCiPl project members for many discussions on constraint visualization.

The implementation of VIFID was done by José Manuel Ramos [11.23]. The prototype implementation of TRIFID was done by Göran Smedbäck.

The material in this chapter comes mainly from DiSCiPl project reports, parts of which were also presented at the 1999 Practical Application of Constraint Technologies and Logic Programming Conference in London [11.24], and at the AGP'98 Joint Conference on Declarative Programming [11.8].

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

11.1 A. Aggoun and H. Simonis. Search Tree Visualization. Technical Report D.WP1.1.M1.1-2, COSYTEC, June 1997. In the ESPRIT LTR Project 22352 DiSCiPl.

11.2 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.

11.3 R. Baecker, C. DiGiano, and A. Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

11.4 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.

11.5 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, Linköping, Sweden, May 1997. U. of Linköping Press.

11.6 L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Workshop on Logic Programming*, Debrecen, 1980.

11.7 D. Cabeza and M. Hermenegildo. WWW Programming using Computational Logic Systems (and the PILLOW/CIAO Library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.

11.8 M. Carro and M. Hermenegildo. Some Design Issues in the Visualization of Constraint Program Execution. In *AGP'98 Joint Conference on Declarative Programming*, pages 71–86, July 1998.

11.9 The CLIP Group. Program Assertions. The CIAO System Documentation Series – TR CLIP4/97.1, Facultad de Informática, UPM, August 1997.

11.10 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.

11.11 M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *Journal of Logic Programming*, 19,20:351–384, 1994.

11.12 Mireille Ducassé. A General Query Mechanism Based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Programming Language Implementation and Logic Programming, PLILP'92*, volume 631 of *LNCS*, pages 400–414. Springer-Verlag, 1992.

11.13 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.

11.14 Massimo Fabris. CP Debugging Needs. Technical report, ICON s.r.l., April 1997. ESPRIT LTR Project 22352 DiSCiPl deliverable D.WP1.1.M1.1.

11.15 J.M. Fernández. Declarative debugging for BABEL. Master's thesis, School of Computer Science, Technical University of Madrid, October 1994.

11.16 M. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 724–734. Springer-Verlag, August 1996.

11.17 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

11.18 K. Kahn. Drawing on Napkins, Video-game Animation, and Other ways to program Computers. *Communications of the ACM*, 39(8):49–59, August 1996.

11.19 Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

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

11.21 Sun Microsystems. Animated Sorting Algorithms, 1997. Available at `http://java.sun.com/applets/`.

11.22 U. Montanari and F. Rossi. True-concurrency in Concurrent Constraint Programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 694–716, San Diego, USA, 1991. The MIT Press.

11.23 J.M. Ramos. VIFID: Variable Visualization for Constraint Domains. Master's thesis, Technical University of Madrid, School of Computer Science, E-28660, Boadilla del Monte, Madrid, Spain, September 1998.
11.24 G. Smedbäck, M. Carro, and M. Hermenegildo. Interfacing Prolog and VRML and its Application to Constraint Visualization. In *The Practical Application of Constraint Technologies and Logic programming*, pages 453–471. The Practical Application Company, April 1999.