Visualization Designs for Constraint Logic Programming*

Manuel Carro Manuel Hermenegildo
Computer Science School — Technical University of Madrid
Boadilla del Monte
28660 Madrid — Spain
Ph. +34-91336-7455 — Fax +34-91336-6595
{mcarro,herme}@fi.upm.es

Abstract

We address the design and implementation of visual paradigms for observing the execution of constraint logic programs, aiming at debugging, tuning and optimization, and teaching. We focus on the display of data in CLP executions, where representation for constrained variables and for the constrains themselves are seeked. Two tools, *VIFID* and *TRIFID*, exemplifying the devised depictions, have been implemented, and are used to showcase the usefulness of the visualizations developed.

Keywords: Logic Programming, Constraint Logic Programming, Visualization, Debugging, Performance, Abstraction of Visual Representations.

1 Introduction

Program visualization has been classically used by computer scientists for many different purposes, including teaching, debugging, and optimization. However, classical program visualizations are often too dependent on the programming paradigms they were devised for, and do not adapt well to the nature of the computations performed in other paradigms (e.g., visualization of concurrent programs focuses on aspects which are not present in sequential programming). In particular, Constraint Programming nature differs radically from that of other programming paradigms and the visualization should address different problems. Moreover, it appears that the needs of CP practitioners are also different from those using other paradigms.

In any case, a good pictorial representation is fundamental to achieve a useful visualization. Thus, it is important to devise depictions which are well suited to the characteristics of CLP data and control. In addition, a recurring problem in the graphical representations of even medium-sized executions is the huge amount of information that is usually available. To cope successfully with these undoubtedly relevant cases, *abstractions* of the representations are needed. Ideally, such abstractions should show the most interesting characteristics (according to the particular objectives of the visualization process, which may be different in each case), without cluttering the display with unneeded details.

2 Constraint Logic Programming in a Nutshell

Constraint Programming (CP) is "one of the most exciting developments in programming languages of the last decade" [MS98]. CP refers to programming using the equations which characterize the solution to a problem. These equations, which are similar to the arithmetical ones, can range over a wide gamut of domains: integers, reals, terms (i.e., data structures), strings, sets,... A CP system would automatically (and incrementally) solve these equations, therefore yielding a solution to the initial problem. This approach has undoubtedly much to do with, on one hand, mathematics itself,

*This work has been partially supported by Spanish EDIPIA Grant (CICYT TIC 99-1151).

and, on the other hand, Operation Research. But, from a practical point of view, it departs from them in two ways: the ability to set up dynamically the equations which model a problem (and probably retract some of them at some point and add new ones), and the use of domains not usually found in mathematics.

Constraint Logic Programming (CLP) [Van89] merges the constraint-based approach with the Logic Programming (LP) ideas, resulting in a highly synergetic combination. The properties of logic programming variables (single assignment, unification) and the control usually implemented in LP (automatic search procedures with backtracking, goal delaying) fit particularly well within constraint programming. The result is a family of languages which naturally extends LP in a unified framework (such that LP can be seen as a case of the more general CLP), patching up some weaknesses (especially in arithmetic) found in LP languages.

One of the most useful CLP domains is finite domains: finite domain variables range over finite sets of integers, and the operations allowed between them are pointwise extensions of the regular arithmetic operations and relations. Although there is no elimination procedure similar to that of linear equations over the reals, a set of equations in finite domains can always be decided to have or not solution, ultimately thanks to the finiteness of the domains, using a mixture of simplification, value propagation, and labeling (i.e., assignment of values to variables). We will focus mainly, due to their practical importance, in finite domain variables.

3 Displaying Constrained Variables

The concept of variable binding in CLP is more complex than in imperative and functional languages: the value of a CLP variable is actually a complex object representing a (potentially infinite) set of values plus the constraints attached to the variable which relate it with the rest of the variables. Textual representations are usually not very informative and difficult to interpret and understand, and a graphical depiction of the variables can offer a view that is easier to grasp. Also, if we wish to follow the history of the program it is desirable that the graphical representation be either animated or laid out spatially as a series of pictures. The latter allows comparing different behaviors easily, trading time for space.

3.1 Depicting Finite Domain Variables

FD variables are instantiated to an initial domain, which is narrowed as equations are incrementally added and as the constraint system is simplified (either by algebraic rewriting or by the labeling procedure). 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 means of language primitives. For several reasons (space limitations, speed of addition/removal of constraints, etc.) this domain is usually represented using an upper approximation of the actual set of values that the variable can take.

A possible graphical representation is to assign a dot (or, depending on the visualization desired, a square) to every possible value a variable can take, highlighting those values in the current domain. 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 1.

It is extremely interesting to follow the evolution of a set of program variables throughout the execution. Probably the most useful portray is to simply stack the different state representations, as in Figure 2. It can reflect time accurately (for example, by mapping it to the height between changes) or ignore it by simply stacking a new row of a constant height every time a variable domain changes or

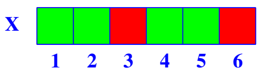


Figure 1: Depiction of a FD variable

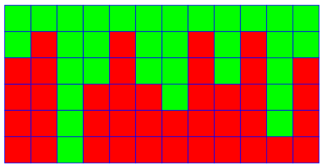


Figure 2: History of a single variable

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

program:-
        Variables = .... ,
        Names = .... ,
        open_log(Variables, Names, Handle),           %% Added
        constrain_values(Variables, Handle),
        log_state(Handle),                            %% Added
        visual_labeling(Variables, Handle),
        close_log(Handle).                            %% Added
```

Figure 3: An annotated program skeleton

an enumeration step is performed. This representation allows the user to perform an easy comparison between states and has the additional advantage of allowing more time-related information to be added to the display. Other possibilities we will not explore include animating the display, so that time is represented as such, or using different color hues or shades of grey.

The *stacking* approach is one of the visualizations available in *VIFID*. *VIFID* is a Prolog library which represents the state of variables as instructed by spy-points introduced by the user in the program. Figure 3 shows an skeleton example of such an annotated program. The `open_log/3` primitive initializes the `Handle` data structure which contains the `Variables` to be observed and their `Names`. `close_log/1` takes the necessary actions in order to finish the visualization (e.g., closing a file, sending appropriate messages to the windows, etc.). The actual step-by-step depiction of the current state is made by the `log_state/1` primitive. It contacts with the visual side of the tool in order to communicate the current state of the variables and update the windows.

An important part of the CLP execution is the labeling phase, which tries to assign values to the variables which are compatible with the existing constraints. This labeling is usually performed by a builtin, which receives a list of variables and an indication of the labeling strategy. We visualize

```
visual_labeling([], _Handle).
visual_labeling([Q|Qs], Handle):-
        labeling([Q]),
        log_state(Handle),
        visual_labeling(Qs, Handle).
```

Figure 4: The `visual_labeling/2` library predicate

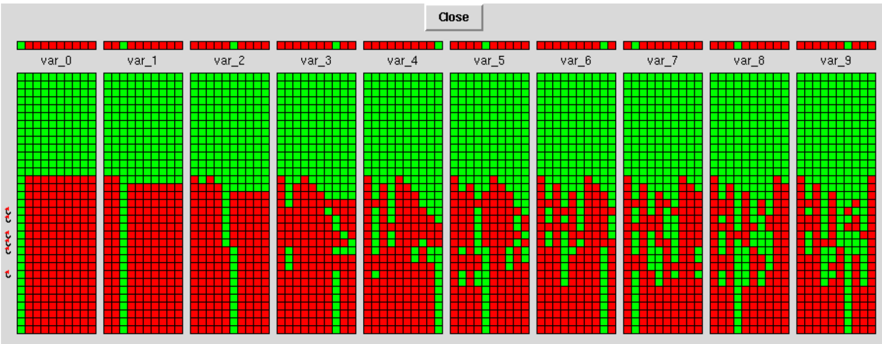


Figure 5: Evolution of FD variables for a 10-queens problem

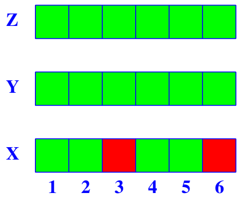


Figure 6: Several variables side to side

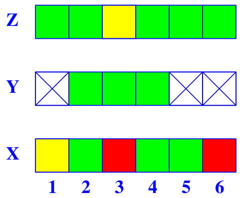


Figure 7: Changing a domain

the evolution of the variables during labeling by recoding this builtin so that the state of the variables is logged after each labeling step. Figure 4 shows an example implementation, which receives the list of variables to label and the `Handle` to the visualization and performs a tailored labeling. It is a simplified code for illustration purposes, but it clarifies how this (and other primitives) can be interfaced with the visual tools without too much effort.

Figure 5 shows a screen dump of a window generated by *VIFID* presenting the evolution the state variables when solving the *Queens* problem for a board of size 10. Each column in the display corresponds to one program variable. 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. Each row in the display corresponds to a spy-point in the source program, and points where backtracking happened are marked with small hooks. It is straightforward 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.

4 Representing Constraints

It is obviously interesting to represent the relationships among several variables as imposed by the constraints affecting them. Textual representation 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. Moreover, in general there are many states of the variables which meet the restrictions imposed by the constraints. A general solution which takes advantage of the representation of the actual values of a variable (and which is independent of how this representation is actually performed) is to use projections to present the data piecemeal and to allow the user to update the values of the projected variables, while observing how the variables being shown are affected by such changes. This can often give the user an intuition of the relationships linking the variables (and detect, for example, the presence of erroneous constraints). We will use the constraint **C1**, below, in the examples which follow:

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

Figure 6 shows the domains of FD variables `X`, `Y`, and `Z` subject to **C1**. An update of the domain of a variable should induce changes in the domains of other related variables. For example, we may discard the values 1, 5, and 6 from the domain of `Y`, which boils down to representing the constraint $\mathbf{C2} \equiv \mathbf{C1} \wedge Y \neq 1 \wedge Y < 5$

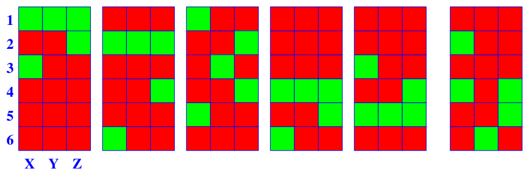


Figure 8: Enumerating `Y`, representing enumerated domains for `X` and `Z`

Figure 7 represents the new domains of the variables. Values directly disallowed by **C2** 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 user-driven visualization is also available in the *VIFID* tool. A more detailed inspection can be done by leaving just one element in the domain of a variable, and watching how the domains of other variables are updated. In Figure 8 `Y` is given a definite value from `1` (in the leftmost rectangle) to `6` (in the rightmost 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.

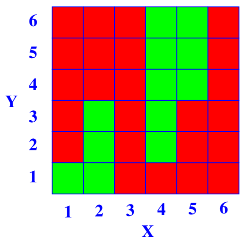


Figure 9: `X` against `Y`

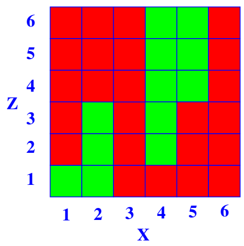


Figure 10: `X` against `Z`

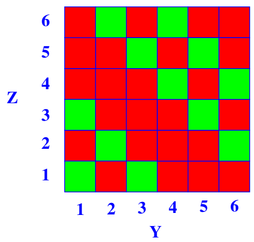


Figure 11: `Y` against `Z`

A static version of this view can be obtained by plotting values of pairs of variables in a 2-D grid. This is schematically shown in Figures 9, 10, and 11, where the variables are subject to the constraint **C2**. From these representations we can deduce that the values `X = 3` and `X = 6` are not feasible, regardless of the values of `Y` and `Z`. It turns out also that the plots of `X` against `Y` and `X` against `Z` (Figures 9 and 10) 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 entailed by the store. This possibility is discarded by Figure 11, in which we see that there are consistent pairs where `X` $\neq$ `Z`. 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 10-Queen 2-D plot appears in Figure 12; the check buttons at the bottom allow the user to select the variables to depict.

5 Abstraction for Constraint Visualization

It is often the case that executions of large programs result in too much data being displayed. Even if an easy-to-understand depiction is provided, the amount of data can overwhelm the user with an unwanted level of detail. Abstraction is a method to cope with this problem.

5.1 Abstracting Values

While the presence of a large number of variables can be partially solved by a careful selection of variables, another problem remains: representations of variables with a large number of possible values can convey information too detailed. At the limit, the screen resolution may be insufficient to assign a pixel to every value in the domain. This is easily solved by using scrollable canvases, providing means for zooming, fish-eye views, etc. That was the approach taken in the VisAndOr tool [CGH93] aimed at showing the parallel execution of logic languages.

However, these methods are more "physical" approaches than true conceptual abstractions of the information, which are richer and more flexible.

An alternative is to use an application-oriented filtering of the variable domains. For example, if some parts of the program are trusted, their effects can be masked out by removing the values already discarded from the representation of the variables: 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.

Another alternative is to perform a more semantic "compaction" of parts of the domain. As an example, consider presenting 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". This idea is the basis of our next visualization.

5.2 Domain Compaction and New Dimensions

Besides the problems in applications with large domains, the static representations of the history of the execution (Figure 5) 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. A better option is to use the number of active values in the domain as coordinates

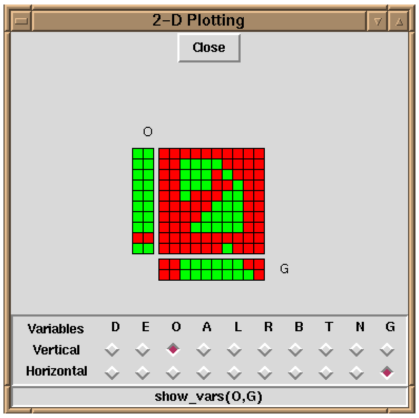
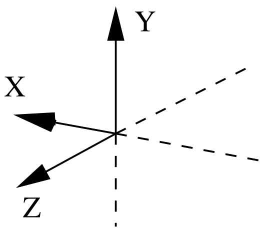


Figure 12: Relating variables in *VIFID*



X The CLP(FD) variables.

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

Z Time.

Figure 13: Meaning of the dimensions in the 3-D representation.

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

dgr(ListOfVars):-
    ListOfVars = [G,O,B,N,E,A,R,L,T,D],
    open_log(ListOfVars, Handle),                  %% Added
    domain(ListOfVars, 0, 9),
    log_state(Handle),                             %% Added
    D #> 0,
    log_state(Handle),                             %% Added
    G #> 0,
    log_state(Handle),                             %% Added
    all_different(ListOfVars),
    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
    visual_labeling(ListOfVars, Handle),
    close_log(Handle).
```

Figure 14: The annotated `DONALD + GERALD = ROBERT` FD program.

in an additional dimension. Figure 14 shows a CLP(FD) program for the `DONALD + GERALD = ROBERT` puzzle. The program was annotated with calls to predicates which act as spy-points, and log the sizes of the domains of each variable at the time of each call.

Figure 15 is an execution of the program in Figure 14. The variables closer to the origin (the ones which were labeled first) are assigned values quite soon in the execution and they remain fixed. There are backtracking points scattered along the execution, which appear as blocks of variables protruding out of the picture. There is also a variable (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): in case of backtracking, the change of one of them affects the others. This suggests that the variables in this program can be classified into two categories: one with highly related variables (those whose domains change at once in case of backtracking) and a second one which contains variables relatively independent from those in the first set.

These figures have been generated by a tool, *TRIFID*, integrated into the *VIFID* environment. They were produced by using the ProVRML package [SCH99], which allows reading and writing VRML code from Prolog, with a similar approach to the one used by PiLLoW [CH97]. One advantage

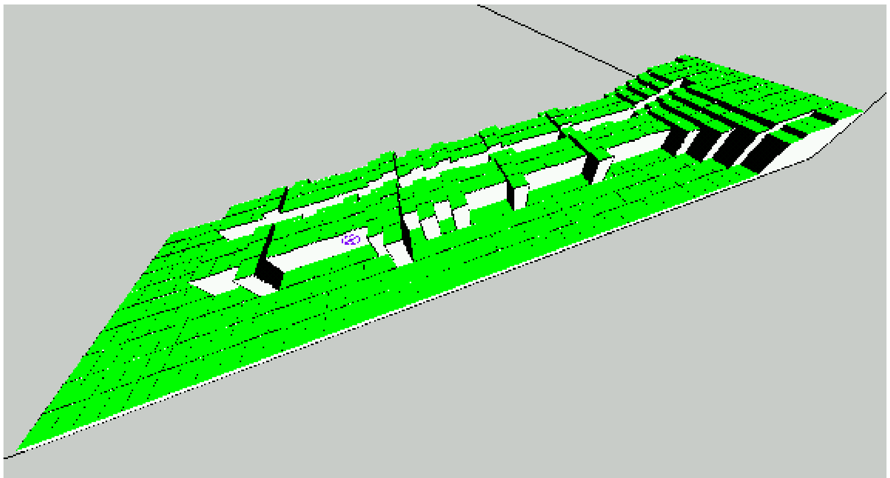


Figure 15: Execution of the `DONALD + GERALD = ROBERT` program

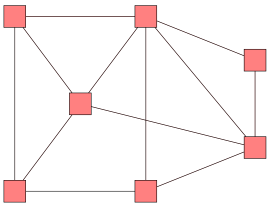


Figure 16: Constraints represented as a graph

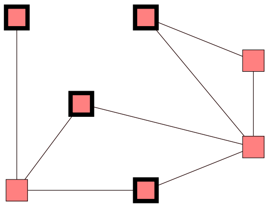


Figure 17: Bold frames represent definite values

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. Another reason to use VRML is the possibility of using hyper-references to add information and animation to the depiction of the execution without cluttering the display.

5.3 Abstracting Constraints

As the number and complexity of constraints in programs grow visualizing them as relationships among variables may cause the same problems we faced when trying to represent values of variables. The solutions suggested for the case of representation of values are still valid 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 in their representation.

A different approach to abstracting the constraints in the store is to show them as a graph [MR91] where variables are represented as nodes which are linked iff the corresponding variables are related by a constraint (Figure 16)[1]. This representation provides the programmer with an approximate understanding of the constraints in the solver (but not exactly *which* constraints they are). 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. 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 interconnected, it

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

may be worth to set up the tightly connected sections and make a (partial) enumeration early, to favor local constraint propagation, and then link the different regions. Animation can reflect propagation and how variables acquire a definite value. In Figure 17 some variables became definite, and as a result some constraints them are not shown any more: this reflects the idea of a system being progressively simplified, and also visualizes how backtracking affects the constraint store. 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).

6 Implementation Details

VIFID and *TRIFID* are implemented in Prolog and Tcl/Tk, and rely on a few primitives to open socket connections and to spawn and communicate with other processes (primarily for the Tcl/Tk part). *VIFID* is completely interactive, and since the library has direct access to the program variables, the user can update them on the fly. The execution can continue after updating, but the user does not have to commit to this update: a RESET button forces the program to backtrack to the point where the update was made. Only a few rutines commonly used throughout the execution were written directly in Tcl/Tk. The flexibility of Tcl/Tk was enough, since most of the windows have a simple layout. The speed of Tcl/Tk was not much of a problem, except when the number of objects in the window became very large. Overall, the tool was strong enough to be used routinely, and the visualization was found to be useful and easy to understand.

TRIFID shares many ideas with *VIFID*: it is also a Prolog library which scans the variables it has access to, but instead of starting an interactive 3-D visualization, we decided to take advantage of a Prolog to VRML interface and generate VRML. Gathering the data was not a computational problem; instead, we found troubles related with the size of the generated files[2], and with the speed of the VRML visualizers (freely) available.

7 Related Work

Early work in constraint visualization was made for Eclipse [ECR93]; the GRACE system [Mei96] represented the values of constrained variables as we did in Section 3.1, connected to a Byrd box model for program debugging. Additional information was encoded using different color shades. More recently, the DisCiPl project [DHM00] fostered the use of visualization and assertion-based debugging tools.

Some constraint applications need to set up complex relationships among the variables. In those cases a visualization which mimics the initial problem helps in mapping problems in the constraint solving to the original problem. The Global Constraint visualization tool [SABB00] does precisely this, by incorporating special visualizations tailored to some of the complex constraints available in the CHIP system. Although this gives an intuitive representation, it needs the user to map the problem to one of these *standard* complex constraint templates.

The visualization of constraint networks, proposed here as an constraint abstraction amenable of being treated and studied, was implemented at a different level in the Constraint Investigator [TM99], interfaced with the Oz Explorer [Sch97]. This proposal visualizes a graph which is close to the implementation. The ability to expand and collapse the constraint net and to filter the variables increases the

[2]More precisely, the VRML visualizers had problems with that!

tool usefulness in the case of big executions. It gives a good representation of the store, but probably needs some further structure to represent complex problems.

8 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 among them. In order to be able to deal with large executions, we have also discussed some 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 integrated prototype tools: *VIFID* and *TRIFID*. *VIFID* and, to a lesser extent, *TRIFID*, which is less mature, have evolved into a practical system. Also, some of the views and ideas proposed have since made their way to other tools, such as those developed for the CHIP system [SA00].

About the Authors

Manuel Hermenegildo received his Masters and Ph.D. degrees in Computer Engineering from the University of Texas at Austin. He is currently Full Professor at the Dept. of Artificial Intelligence in the School of Computer Science of the Technical University of Madrid (UPM), Spain, where he leads the Computational Logic, Implementation, and Parallelism (CLIP) group[3]. He has participated in several E.U. funded (ESPRIT) and locally funded research projects as research group leader or coordinator (for ESPRIT project PARFORCE). Previously he was project leader at the Microelectronics and Computer Technology Corporation (MCC) research center and Adjunct Professor at the Department of Computer Science of the University of Texas at Austin.

His research interests include advanced compilation techniques, parallelizing compilers, high-performance and distributed computing, concurrent and constraint logic programming, program execution visualization, and parallel computer architecture. He has published more than one hundred papers in conferences and journals on these topics. He is also area editor of ACM Transactions on Programming Languages and Systems and Theory and Practice of Logic Programming and editorial advisor of the Journal of New Generation Computing.

Manuel Carro received his Masters degree in Computer Science from the School of Computer Science of the Technical University of Madrid (UPM). He is currently assistant professor at the department of Languages and Systems and is finishing his Ph.D. degree under the supervision of Manuel Hermenegildo. He has been involved in several Spanish and international projects (in particular, ESPRIT projects PEPMA, PARFORCE, ACCLAIM, VOCAL, and DiSCiPl, Fulbright project ECCOSIC, and CICYT projects IPLD, ELLA, and currently EDIPIA).

His current research interests include abstract machines for parallel execution, visualization of constraint logic programming, implementation techniques and object oriented and distributed computing for logic and constraint logic languages.

[3] http://clip.dia.fi.upm.es/

References

[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.

[CH97] 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.

[DHM00] P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming*. Number 1870 in LNCS. Springer-Verlag, September 2000.

[ECR93] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

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

[MR91] 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.

[MS98] Kim Marriot and Peter Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[SA00] H. Simonis and A. Aggoun. Search Tree Visualization. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[SABB00] H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Complex Constraint Abstraction: Global Constraint Visualization. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS. Springer-Verlag, September 2000.

[Sch97] Christian Schulte. Oz explorer: A visual constraint programming tool. In Lee Naish, editor, *ICLP'97*. MIT Press, July 1997.

[SCH99] 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.

[TM99] Tobias Müller. Practical Investigation of Constraints with Graph Views. Poster in International Conference on Logic Programming, December 1999.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.