An Architecture for Visualizing the Execution of Parallel Logic Programs

Mike Brayshaw

Human Cognition Research Laboratory
The Open University
Milton Keynes, England, MK7 6AA.
Tel: [+44] (0)908 65 5015
FAX: [+44] (0)908 65 3744
email: mc_brayshaw@vax.acs.open.ac.uk

Abstract: This paper describes the development of an architecture and implementation of a graphical tracing system for the parallel logic programming language PARLOG. Novel features of the architecture include a graphical execution model of PARLOG; a range of representational techniques that allow the user a choice of perspective and granularity of analysis; and ongoing work on graphical tools that provide user-defined visualisations of their programs, either before the program is run, or afterwards by demonstration from a textual trace. The aims of the architecture are threefold: (1) to aid program construction and debugging by providing an informative graphical trace of the program's execution; (2) to provide the user with a choice of representational techniques, at a preferred level of granularity; and (3) to allow users to define their own visualisations, that more truly map onto their conception of the problem, and which support the way they wish to view the execution information.

1. Introduction

One of the central tenets of our current research on visualisation is that different visualisation techniques are optimal for different tasks and purposes. Any representation necessarily makes some things explicit at the expense of others. In this work I aim to show how using different presentation styles, which show different aspects of the execution process, can produce a practical visual model of a PARLOG virtual machine, and one that can benefit teaching, tracing, and also debugging.

In this current work on PARLOG I am interested in letting users get their own perspective on the program, and one that suites their needs best. This has resulted in the development of a repertoire of representation techniques that give the user complimentary perspectives, using different levels of detail and abstraction. In addition users are able to come up with their own perspectives using special visualisation defining tools.

Background material to the current discussion can be found in the following. PARLOG is described in Gregory (1987) and Conlon (1989). A good review of program visualisation techniques can be found in Myers (1990) and Utter and Pancake (1989). A visual programming style front-end for PARLOG has been proposed by Ringwood (1989). Conlon and Gregory (1990) present an impressive system for the textual tracing of PARLOG, including the ability to closely monitor (incremental) I/O, processes, and communication channels.

This paper will first briefly introduce a basic model of the visual execution of PARLOG (see Brayshaw, 1990a for a fuller treatment). This model will then be generalised to show how we can embed it within a series of visual stories of a program's execution. There is an inherent problem in many representation systems that in defining a representation some things are made explicit, others implicit, some things highlighted, some hidden. By providing different prespectives on the execution behaviour of the program we will attempt to demonstrate how a fuller and more informed view of the program can be gained from looking at the program as its different faces are revealed in the series of visualisations. Then I shall discuss how we can further utilise these representational techniques so that users can modify them and produce their own visualisations, before finally adding some conclusions. All the representations use the same execution model, however the information is presented from different perspectives, emphasising different aspects of execution.

2. A visual execution model of PARLOG

PARLOG provides an interesting test case for the use of program visualisation, not only to try and generalise our own earlier work on visualisating Prolog (Eisenstadt and Brayshaw, 1987, 1988; Brayshaw and Eisenstadt, 1988, 1991), but also to deal with the special dynamics of parallelism and associated problems like *starvation* and *deadlock* (e.g. see Ringwood, 1988). The basic model is built around the concept of the *node* representing a *process*, and the *shading* of the nodes indicating the process *state* (as in fig. 1 below).

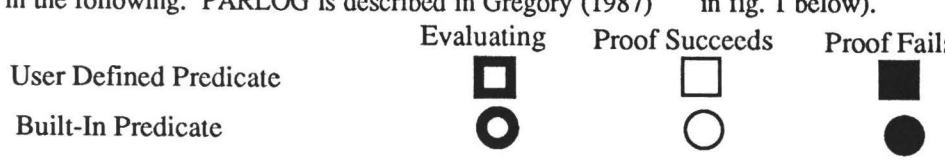

Figure 1. The node shading indicates process state .

This then is the core. To it we now must add a more detailed execution model. The one I use is adopted from Conlon (1989), because it provides a good conceptual model for novices and experts alike of the workings of PARLOG. It considers process evaluation to proceed in four stages, namely *test/commit/output/spawn*, which are discussed in detail below.

Test The clauses of the goal are tested to see if they are candidates. Clauses are tested via OR-Parallel search. For each clause, two tests, of head and guard, are carried out by AND-Parallel search. Clauses can be thought of as racing to be the first clause to satisfy both head and guard tests.

Commit The process commits to the first clause to pass the test stage. All other input matching or guard evaluation, associated with other possible candidate clauses, ceases. The commit stage marks the end of the race between the guards.

Output Once a clause has been committed to, any output arguments can then be bound.

Spawn The process then reduces to the sub-goals in the chosen clause body. As a result, concurrent sub-processes are spawned, one for each of the subgoals. The process succeeds if all the sub-processes succeed. If however one of the conjunctive sub-processes fails, then the overall process goal fails. If the spawned (reduced to) goal is a leaf node in the execution space, then the call succeeds immediately and the goal succeeds.

Figure 2 considers how this maps onto a simple database query.

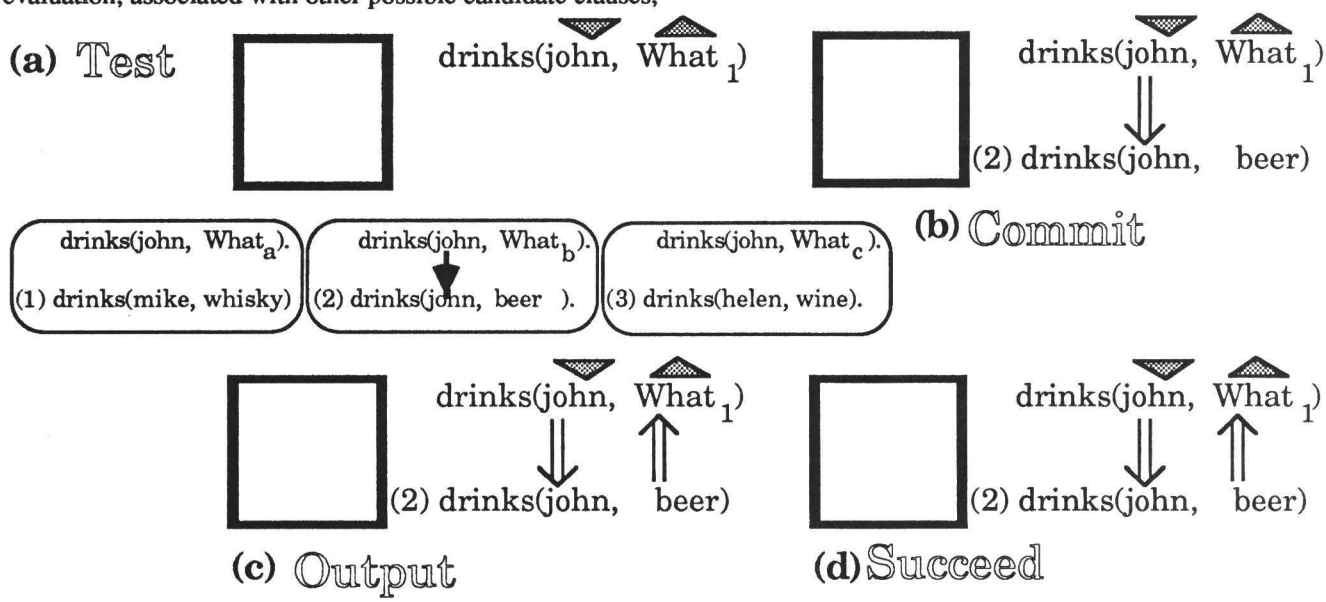

Figure 2. An example of the basic execution model adopted from (Conlon, 1989).

Let us now consider the actions of PARLOG when doing this simple query. On attempting a goal, the process tests in parallel the head and the guard of all the clauses of the goal. In figure 2, we see that we are attempting to prove that **drinks(john,What)** is true against the database **drinks(mike, whisky)**, **drinks(john, beer)**, and **drinks(helen, wine)**. At the top of the figuure we see the test phase of a goal evaluation (a). The icon on the left shows we are currently evaluating a goal. The goal itself is drawn to the right of it. The mode declarations (whether an argument is input or output) are shown by the shaded arrows above the goal. These correspond to the modes of the respective arguments. ▼ indicates input, and ▲ output. Thus we can see that **drinks**'s first argument must be input, and its second output. When we match the heads in parallel we produce a local evaluation space for each clause, as shown in the three boxes below the main process state icon. We can thus see at a glance what the different possible binding enviroments are. The goal is shown at the top of the box, the attempted head match below. Database clauses are indexed by number on the left. Variables are shown renumbered by subscripts, to avoid name clashes. In the main body of the clause the renaming is by number, however in the transient parallel evaluation spaces of the evaluation test phase, we show the temporary variable bindings subscripted by letter. Finally, arrows are used to show data-flow and pattern matching, in the same manner as Eisenstadt and Brayshaw (1988). Thus we can see that of the three possible clauses, only the second clause **drinks(john,beer)** matches the goal **drinks(john, What$_1$)**.

In the top part of figure 2, we can see that only clause two of **drinks** matches the database. As a result, clause two wins the test phase and is committed to. We write the name of the clause next to the node indicating the state of the process, and *commit* at that choice point. Any output variables can now be unified, as we see figure 2(c). Finally we *spawn* any new sub-processes in the body of the clause. As **drinks** is defined as a simple fact, the evaluation now succeeds as we see in the final snapshot, and hence we can see that the overall evaluation has succeeded with the result **drinks(john, beer)**.

The view we have just given is a very fine-grained account of the executon process. We can however develop a much coarser grained model, suitable for monitoring much larger programs. We do this simply by omitting details of the

unification of the program and considering instead just the name of the process and its state, indicated by the icon on the left-hand-side. If I want to see more information about the unification, the different possible binding environments in the test phase, or the data-flow arrow, I can use the same technique of zooming we introduced in (Eisenstadt and Brayshaw, 1987), to reveal more fine-grained information. Also notice that because the coarse grained representation is only a superset of the fine-grained view, I can arbitrarily interleave coarse and fine traces. I will demonstrate this notation further when we incorporate this account in different representational models.

3. Integrated multiple representations of PARLOG programs

So far we have only introduced a model of individual processes. We now need to say how these individual views can be related together to give a model of execution that considers a program made up of many of these nodes. All the views that are discussed are tightly integrated. The user can readily swap from one to the other. Coarse grained views can be opened out into fine-grained ones, and vice verca, by mouse-clicking on the nodes. All the views of the program also have a replay panel, so the trace history can be wound back to the beginning and browsed through, running it forward and back as the user sees fit. The basic modus operandi of the current system is that the user runs the program and then gets a visualisation of it after the run, the user being able to choose different representation style as appropriate. Other representations can be called up from a menu or the user can choose to build new representations (or use an existing user-defined visualisation).

An AND/OR tree gives us our kernel prepresentation style. Amongst its strengths are its good mapping to the source code and compactness. Our basic representation is introduced below.

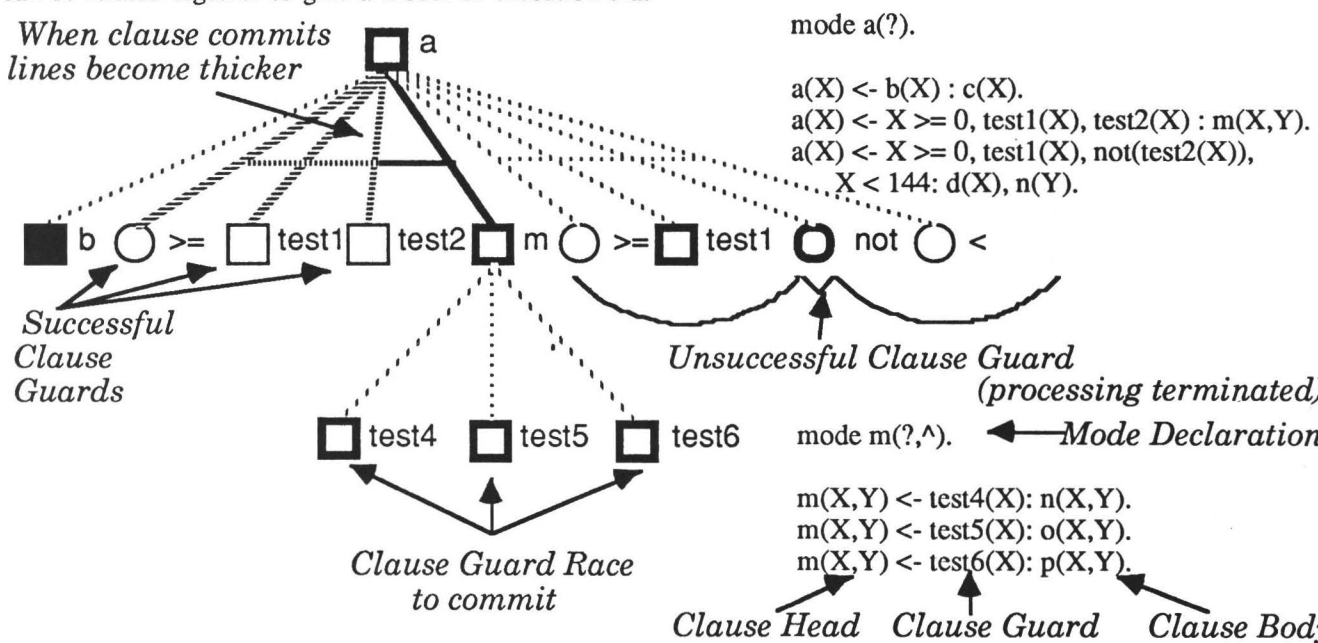

Figure 3. The basic AND/OR tree representations employed.

On the left we see the basic notation we use; *ANDs* are distinguished from *ORs* by the conjunction bar linking arcs in the tree. Alternative clauses can be viewed via ORs and conjoined sub-processes in a clause body as ANDs. Guards are distinguished by characteristic line stippling. Once the processes commits to one particular choice, the successful guards change to a broader hashed line style to distinguish them from the surrounding unsuccessful ones. These unsuccessful nodes are left in whatever state they were in when the commitment took place. The fact that one of the clause guards is shown as being successful necessarily implies that these others have not been. Program dynamics can be revealed by using the replay panel which can show what happens and when. When a clause commits processes are shown in whatever state they were in when they were terminated, as this can be informative about how and why a particular process failed/successfully committed at a particular point. Using dynamics, suspended goals can be see as evaluating root nodes, their lack of activity showing dynamically the suspension, and graphically we reinforce this model by showing a line underneath to distinguish them from non-suspended nodes.

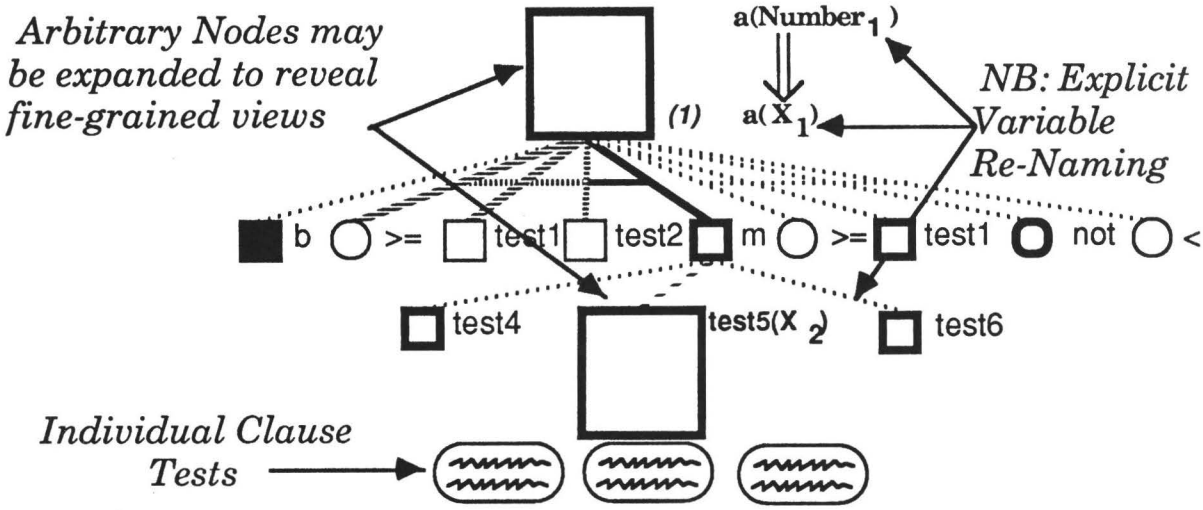

Figure 4. Mixing coarse and fine-grained views.

In figure 3 all the nodes that we dealt with were coarse-grained descriptions. As we noted earlier we can however introduce fine grained information as well. To do this we can zoom in on selected nodes. In figure 4 above, we can see the effect of doing this for the nodes **a** and **test5**. In both cases we now have our fine-grained view of execution again. Notice also, that at the moment that we have taken the snapshot in figure 4, **test5** is in the process of its *test* phase. We can see the individual clause evaluation environments below the bottom of the process node. The environments for **a** are not shown by default, other than when the process was in its test phase, but can be brought up again and displayed as at the point of commitment, by mouse-clicking on the node.

Another description of PARLOG can be in terms of many intercommunicating processes. According to such a model, PARLOG can be thought of as a kind of object-oriented language. To support this view we have developed a process communication model, illustrated in figure 5.

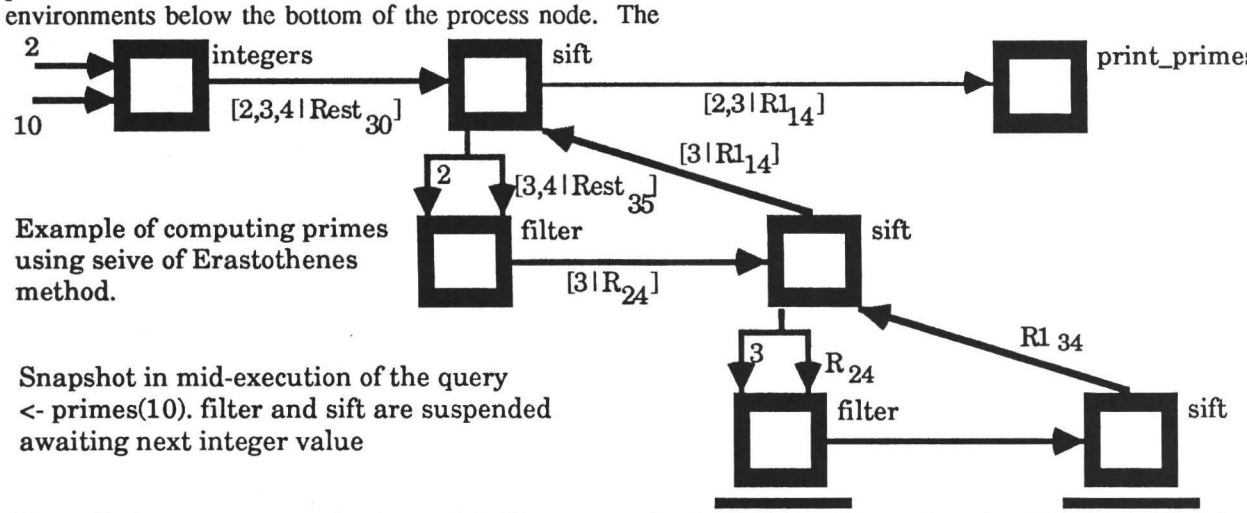

*Figure 5. A process communication model. We can see what data is being communicated and between whom, by looking at what information is on which channel. Notice, the re-occurring shapes caused by recursive **filter** and **sift** processes*

In this model we view each process as a node, and link the processes together according to the data-flow between them. On these links we show either the data-flow that has occurred in an incremental fashion, or just the new elements, depending upon the user's preference. The nodes in the diagram are actually the same as those in the tree based views that we have been looking at previously. In the trees, each process was depicted by a node whose visual state descriptor mapped onto that of the process. We can use the same notation in the data-flow diagrams. We can view these data-flow diagrams as horizontal cross sections of the trees. However, the logical control shown in the trees is removed. The data-flow diagram need not be made up of simple cross-sections; the slices can come from arbitrary parts of the program, hence arbitrary process communication can be seen. It is important to remember, however, that the relation between the two representations is essentially one of perspective; one representation emphasises control, making data-flow implicit, the other pulls data-flow between processes out front, but hides the overall control relations between the processes.

A final representation style we employ is based on the metaphor introduced in (Domingue and Eisenstadt, 1989). This metaphor aims to make repetetive or cycle-based behaviour explicit. In PARLOG, constructs like recursion allow us to conceptualise a program carrying out some form

of repetetive process. Although we could detect this type of behaviour in the other traces it would be implicit within the trace representation we are employing. The *cycle table* aims to make this explicit. The basic form is to consider a list of items to viewed on a per-cycle basis and place these items down one axis of the table. Down the other we consider the individual cycles. The cross section of these two axis then tells us what event happenned on what cycle to what item. In PARLOG we have defined a default mapping between recursive processes and the cycle table. Likewise we can also look at guards on each invocation of a processes, so that each new guard becomes a new conceptual "cycle". Doing this allows us to compare the behaviour of the guard on each sucessive invocation.

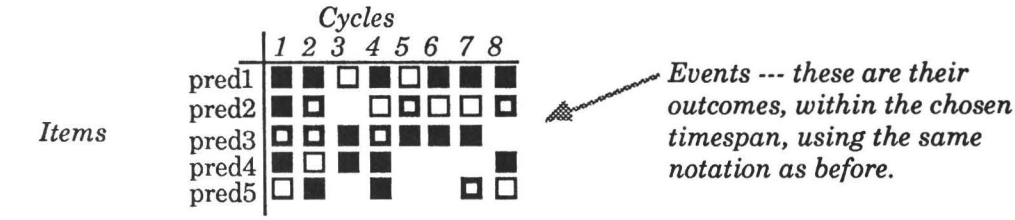

Figure 6. A schematic view of the cycle based representation, call the *cycle table*.

These representation styles form the basis of our library. Thus currently we can distinguish between three different execution metaphors and two types of granularity. In the next section we will discuss how to use these execution metaphors with new canonical units defined by the user. Moreover, the architecture is open to the extent that we can expand the system by embracing other metaphors of execution, whilst still maintaining our execution model and levels of granularity. The important notion has been the ability to see execution from different perspectives: goal reduction, process communication, and iteration.

4. User Defined Visualisations

One of the important proposed features of this system is the provision for users to be able to define their own visualisations. This section will describe possible ways we are exploring to go about doing this. Taking any one of the representations as a template, users should be able to program a new mapping between their program and the trace they have generated, thereby producing a new visualisation, of their own design. We have here tried to design additional tools that let the user write this mapping more easily, without getting involved in heavy code production to program the debugger. To this end we will discuss two tools. The first lets the user define visualisation by selecting nodes from the call graph of the program before it is run. Thus the programmer can, *a priori*, decide what trace to see of his/her program, design the trace, and then run the program. The second method works post execution. It allows the user to take a textual trace of the program and describe the mapping between representation template and textual trace in terms of it instances, as shown in the trace. Thus the user can define his/her own visualisation based on a posthumous analysis of the trace. In addition however, the most crucial concept that distinguishes these user defined mappings is that they need not be in terms of individual nodes in the underlying trace. Instead, groups of nodes can be considered as an individual cliché, and mapped as one into the representation template. Thus higher level descriptions of the program can be rendered in the visualisation by showing the mapping these higher level events have to the underlying symptomatic footprint they leave in the program trace. By thus analysing the program trace and detecting the user defined footprints, these events can then be mapped into the representation template chosen, and a user defined visualisation produced. This is implemented as a sequential search of the trace history. However, as relative temporal information is also preserved, simulated concurrency is therefore possible. How this proceeds depends on the representation chosen, as we will discuss below.

In trees and process communication views arbitary mappings may be produced by describing the footprints of individual nodes. Thus nodes in the tree need not be individual processes, but instead represent much larger clichés about the program. The way we propose to go about doing this is as follows. The template consists of an abstract model of the representation with a series of roles (e.g.nodes, links etc.) which the user has to instantiate with a series of generators. These are defined as cliché patterns to look for in the trace history, each such cliché is said to be defined by the characteristic *footprint* it leaves behind in the trace when it occurs. Each trace footprint has a script associated with it, that is defined using one of the two techniques we will discuss below. For each process, the tracer searches the trace history of the process and its spawned subprocesses. At specific points in the script, the script can allow for nested cliché patterns. Thus a script could say that between the call and exit of a particular process, go and look for clichés that occurred within this particular subtree. In this manner hierarchical abstractions can be produced.

For cycle tables the process is slightly different. Here, there are three critical things to define. The first is what event marks the beginning of a new conceptual cycle. Then what are the items (termed *agents*) that we wish to consider on these cycles. Finally we have to define what are the events we wish to see. The algorithm then computes the following. It first looks for cycle events, starting from a particular root process, and searching downwards. Once it has found the intervals that make up the cycle, it then looks for each *agent*, to see which events clichés happenned to it in that particular cycle. If more than one event occurred the default is to show the last. Earlier ones can be picked up using the replay notion introduced earlier. Note that the dynamics of this display can be (though by no means need to be) very different from the parallel dynamics of the PARLOG code underneath. In other words, this model can be used to map a different conceptual model of the machine, and indeed might be used to *serialise out* a particular piece of

code, if that helped the user's conceptualisation of the problem.

In order to make the definitional process of these mappings more easy, we are constructing the following two tools.

4.1 Graphical techiniques for pre-execution trace construction

This is a generalisation of a technique developed for Prolog in (Brayshaw, 1990b). The user is first asked to choose a representational style. For whatever style they choose, they have to fill in the role fillers in the template they wish to use. They can do this by identifying the footprint in terms of the program's call graph. The call graph plots out the potential execution space of a program. From it the user can then specify which nodes in the execution space they wish to group together, and in what calling structure, in order to specify the footprint of the cliché they are interested in. The footprint is produced by specifying a pattern of nodes from the call tree. The footprint they are defining is that pattern in the final execution trace that constitutes an instance of the concept they are defining. Each time they choose a node they are prompted to further conditionalise their choice by adding a script to that node. The final script for the footprint is the conjunction of the scripts for the individual nodes. Variable naming is unique to the entire script and not just local to an individual node script, so constraints between local nodes can be so recongnised. The language for the scripts is Prolog, since this is the language in which the tracer is embedded. In order to write the script the user is given a menu based authoring tool.

The tool allows the user to say what the patterns in the trace for this footprint are. Legal patterns are derived from the execution descriptors originally designed to describe Prolog execution (Eisenstadt, 1985), but here generalised to PARLOG. They include a call (with optional reference to a particular clause), call success or failure, and the options to be specific about the type of failure. To the left of this menu is the general script editing area. Users can type into this if they so wish, or edit what is there. Additional Prolog goals can further constrain the script. Additionally however the user can edit the script by means of a menu. Suppose we selected an example node called **a** in the call graph. The default script is that for this pattern to hold, **a** is simply called, and we can express this by the line **call(a(_,_,_))**. However, we can make this pattern much more specific. We can see any calls to specific clauses of **a** (namely clauses 1,2, and 3), which resulted in failure due to sub-process failure. To do this and create the script, we have to choose the appropriate options from the menu, and then specify to a separate dialog what are the numbers of the clauses we wish to consider. Nodes can be added to a footprint by repeatedly selecting them from the call graph and defining scripts in this way. The calling structure specified from the call graph between the individual nodes is also made a precondition of the script. Once a cliché definition is complete a double click ends the definition process. Where appropriate the user can call upon an icon editor to define a special symbol for that cliché or choose to use the built in representational styles. This definitional processes can be repeated to define any number of clichés.

4.2 Defining visualisations from the textual trace of a progam

This second technique attempts to allow the user to work from a textual trace of the program. The textual trace has the same symptom descriptors as we saw in the previous section, again adopted from Eisenstadt(1985). Users can define a cliché, and associate it with the role filler of a template, by first choosing a representation style and a particular role. They can then define the cliché footprints by selecting instances of these footprints as they have manifested themselves in the runtime trace of the program. Several instances can be presented as examples of the footprint. Where the new is information is more specific, this is then added to the definition of the script defining the footprint. When generalisations are involved the tracer has a series of syntactic rules it follows (true automated generalisations not being possible without a model of the language semantics). So for example when two examples differ by an argument containing different literals, it is assume that the value doesn't matter and the values are replaced by a variable. Likewise the user can choose to automatically hollow out a term as short hand for making the example more generic. During the whole operation, the script that is being developed can be seen and edited, and augmented by additional constraints in Prolog. Once the scripts are defined, and depending upon the representation chosen, the new definitions are used to try and produce a new visualisation of the program. We see this process of visualisation production as potentially being an iterative one so that if the resulting views are not to the programmers' liking they can go back and edit the scripts to produce new ones. Scripts developed from the call graph descriptions can also be re-edited in this fashion, and in light of the actual program execution as seen in either the textually based traces, or the four graphical based views.

5. Scaling Issues and Other Approaches

Text tracing systems can be thought of as a type of program visualisation system. To convice yourself of this just consider the amount of layout, structuring and indenting that makes them readable. Like their graphical conterparts, to be successful they really have to present the required information in an acceptable manner. Thus any visualisation system, be it textual or graphical, has to think about scaleability, and when you consider scaleability, you have to consider the types of representations used. We have discussed ways of tackling the scaling issue for graphics elsewhere (Brayshaw and Eisenstadt, 1991). The point I want to make here is that there isn't some special dichotomy which destinguishes textual visualisations from graphical visualisations. Both are required to provide sufficient information, and *selective* information. Exhaustive program traces, in either mode, can be too unwieldy to use, if not useless, and take too long. For either system, what is required is an informative trace of a limited part of the

program. I therefore believe that it is misleading to think that somehow text traces are "practical" while graphics ones not. They are different ways of exploring the information space, and what makes program visualisation exciting is finding what possibilities graphics affords for openning up new routes.

6. Conclusions.

In this paper I have argued for the use of different perspective techniques to visualise PARLOG. These have focused on an execution model, embedded within three metaphoric description of the system (goal reduction, process communication, and iteration), and portrayed either at fine, coarse, or user defined levels of detail. These techniques are currently being realised in an implementation that has been described in this paper. The system is written in Prolog using an extended and modified version of the PARLOG meta-interpreter for a subset of the language developed in Pinto (1987).

The current development is particularly focussing on realising the type of flexibility of representation style discussed in section 4. In particular, one desirable future development would be to make the whole of the definitional process of these new footprints graphical, and minimise the amount of "programming" envolved.

7. Acknowledgement

This work is currently supported by MRC/SERC/ESRC UK Joint Research Council Grant #89/CS31, the support of which is gratefully acknowledged. The author would also like to thank Marc Eisenstadt for many discussions about visualisation and program debugging, and for detailed comments on an earlier draft of this paper.

8. References

Brayshaw, M. Visual Models of PARLOG Execution. September 1990. *TR-64*, Human Cognition Research Laboratory, The Open University, Milton Keynes, England.

Brayshaw, M. Visualizing Cyclic Behaviour in Prolog. *TR-66*, Human Cognition Research Laboratory, The Open University, Milton Keynes, England, Dec'90.

Brayshaw, M. and Eisenstadt, M. Adding Data and Procedural Abstraction to the Transparent Prolog Machine. In R.A. Kowalski and K.A. Bowen (Ed.), *Logic Programming* Cambridge, MA: MIT Press, 1988.

Brayshaw, M. and Eisenstadt, M. A Practical Graphical Prolog Tracer. *International Journal of Man-Machine Studies.*, 1991

Conlon, T. *Programming in PARLOG*. Wokingham, UK: Addison-Wesley, 1989.

Conlon, T. and Gregory, S. Debugging Tools for Concurrent Logic Programming, *TR-90-22*, Computer Science Department, University of Bristol, October 1990.

Domingue, J. and Eisenstadt, M. A New Metaphor for the Graphical Explanation of Forward Chainging Rule Execution, *Proceedings of International Joint Conference on Artificial Intelligence* (IJCAI-89), Detroit, 1989 .

Eisenstadt, M., Retrospective Zooming: a knowledge based tracing and debugging methodology for logic programming. *Proceedings of the Ninth International Conference on Artificial Intelligence (IJCAI-85)*. Los Angeles: Morgan Kaufmann, 1985.

Eisenstadt, M., and Brayshaw, M. Graphical debugging with the Transparent Prolog Machine (TPM). *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*. Los Angeles: Morgan Kaufmann, 1987.

Eisenstadt, M., and Brayshaw, M. The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, *5*(4),pp. 277-342, 1988.

Gregory, S. *Parallel Logic Programming in PARLOG: The language and its implementation*. Wokingham, UK: Addison-Wesley, 1987.

Myers, B.A. Taxonomies of visual programming and program visualisation. *Journal of Visual Languages and Computing*, **1**(1), 1990.

Pinto, H. Implementing Meta-Interpreters and Compilers for Parallel Logic Languages in Prolog, *AIAI-PR-14*, University of Edinburgh, 1987.

Ringwood, G.A. PARLOG86 and the Dining Logicians. *Communications of the ACM.*, **31**(1), pp. 10-25., January 1988.

Ringwood, G.A. Predicates and Pixels, *New Generation Computing*, **7**, pp.59-80, 1989.

Utter, P.S. and Pancake, C.M. Advances in Parallel Debuggers:New Appraoches to Visualisation. Cornell Theory Center, *CTC89TR18 12/89*, Cornell University, MY, 1989.