ViMer: A Visual Debugger for Mercury

M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder
School of Comp. Sci and Soft. Eng
Monash University, 3800
Australia

{mcam,mbanda,marriott,pmoulder}@csse.monash.edu.au

ABSTRACT

ViMer is a visual debugging environment for Mercury programs which has three main contributions. First, it employs a new execution tree representation, the layered AND-OR tree, which we believe provides a better way of visualizing backtracking in AND-OR-like trees. Second, it uses incremental constraint-solving to efficiently draw and incrementally update the visualization of the execution tree. And finally, it borrows techniques from standard tracers (such as the use of spy points to reduce the amount of tree nodes, and the placement of restrictions on the amount of information stored at each node) that help keep the tool efficient while still providing enough information for debugging.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

Keywords

execution trees, visualization, incremental tree layout

1. INTRODUCTION

There has been a significant amount of research into debugging of logic and constraint logic programming languages (see for example, [3, 16, 12] and their references). In particular, this research has resulted in several sophisticated visual debugging tools such as the Transparent Prolog Machine [7] (TPM) developed for Prolog, Explorer [16] developed for Oz, APT [6] developed for CIAO, and the Execution Tree Viewer (ETV) [1] developed for PrologIV. However, the picture is far from perfect, and in practice most LP programmers use simple textual debuggers (also called *tracers*) little better than those provided twenty years ago.

One reason for this might be the usual reluctance by programmers to stop using already-familiar environments. However, we believe there are three other reasons for their lack of acceptance by programmers in the logic programming community. The first reason is that none of the trees employed by these tools to display execution flow is ideal: TPM and APT employ AORTA trees (a variant of AND-OR trees) which do not provide adequate visualization of the execution of non-deterministic programs, while Explorer and ETV use SLD trees which are good for displaying the execution of non-deterministic programs but are not well-suited for displaying execution of deterministic programs. The second problem not adequately addressed in those tools is how to efficiently re-layout the execution tree when this is displayed incrementally. This is particularly problematic for AND-OR tree based visualizations in which backtracking can greatly modify the tree not only by adding but also by eliminating nodes. And finally, we believe these tools store and manipulate too much information. ETV and APT are off-line tools, i.e., they compute the complete execution tree and the variable values associated to each node before creating the visualization. TPM and Explorer can both incrementally display the execution tree as it is created but both still store too much information about nodes in the tree (Explorer can forget some information, but only a level at a time).

This paper presents ViMer, a debugging environment for Mercury specifically designed to overcome the problems identified above thanks to three novel features. First, ViMer uses a new representation for the execution tree, the layered AND-OR tree. It is similar to an AORTA diagram for deterministic programs, but uses "redo" layers to visualize backtracking, which we believe provide a more intuitive and complete visualization. Second, it uses incremental constraint-solving algorithms to efficiently recompute the layout of the execution tree as it is built incrementally. This allows optimal utilization of limited screen space at every point in the visualization. And finally, it borrows several techniques from standard tracers that help keep the tool practical, i.e., reasonably efficient while still providing enough information for debugging.

In particular, like Explorer and TPM, ViMer builds the execution tree incrementally as the user steps through the execution. Unlike previous tools, ViMer uses two mechanisms that obviate the need for memorizing every variable binding information across the whole execution: "spy variables" which involves selective memorization of variable bindings, and a 'retry' command which involves limited re-execution of the tree.

Furthermore, ViMer allows the user to indicate the predicates of interest, similarly to how most tracers provide "spy-

PPDP'03, August 27–29, 2003, Uppsala, Sweden.

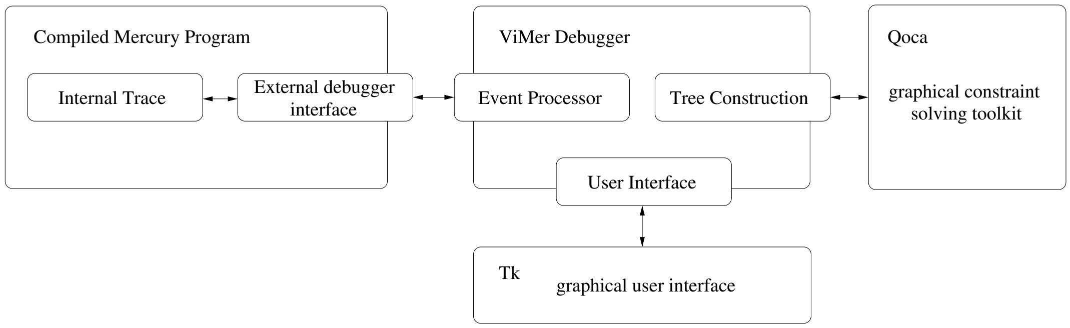


Figure 1: Overview of ViMer

points" (also known as break-points), with execution of other predicates remaining hidden. This behaviour, which is commonly supported by standard tracers and by more sophisticated monitoring tools such as Opium [5], is vital to reduce the size of the internal tree data structures, the amount of communication between the debugger and the instrumented program, and the size of the tree shown to the user.

The rest of the paper proceeds as follows. Section 2 provides an overview of the Mercury language and its support for ViMer. Section 3 provides a brief look at earlier approaches for visualizing execution and details the approach taken by ViMer: the layered AND-OR tree. Section 4 details the choices made in ViMer to increase efficiency, while Section 5 explains how constraint solving is used to visualize the tree. Section 6 quickly reports on the status of the implementation and presents other features. Section 7 presents the results of our experimental evaluation, and Section 8 concludes.

2. OVERVIEW OF MERCURY AND ViMer

The logic programming language Mercury [18] has been designed to support programming in the large. It requires the programmer to declare the *type*, *mode* and *determinism* of each exported predicate. This information is used to provide stricter error checking at compile time, and to create specialized more efficient versions of the predicates, called *procedures*. Modifications to the original source code include literals being normalized and reordered, clauses being transformed into a single clause disjunction, and disjunctions in which only one branch is known to succeed (since selection is based on the value of a ground variable) being transformed into *switches*.

When debugging is enabled, the execution of a Mercury program is represented as a sequence of events. These events can be placed into two categories: external and internal. External events (call, redo, exit, fail and exception) deal with the execution moving from one procedure to another. Internal events (disjunctions, negations, if-then-elses and switches) illustrate the flow of execution internal to the procedure. Information about each event includes a unique ID, the associated procedure call and predicate, the depth of the call, the type of the event (call, disjunction, etc.), the line number of the related literal within the source code, and information about the event's location within the original clause.

Mercury provides a quite flexible *external debugger interface* which allows its users to, for example, step through each event, examine its details and those of the current program state, skip a fixed number of steps, jump to the next event matching given criteria (e.g. only about a given set of procedures), and re-execute from a particular event[1]. The external interface also allows the debugger to obtain the names and values of the variables associated to the last event.

Figure 1 shows the three major components of ViMer and how they interconnect with (a) Mercury's external debugger interface and (b) the C++ constraint solving toolkit QOCA [11], which was specifically designed for interactive graphical applications. The *event processor* module is responsible for receiving the event information and determining the necessary adjustments to the tree structure (if any). The *tree construction* module supports the storage, construction and manipulation of the internal tree representation, and uses QOCA to compute the tree layout. Its implementation is very generic and it is used for displaying both the execution tree and data structures. Finally, the *user interface* module uses the Tcl/Tk graphical toolkit [13] to implement the system's user interface and draw the execution tree on-screen. The current implementation consists of approximately 12000 lines of newly-written and 1500 lines of modified Mercury code.

It is important to mention that the Mercury distribution already includes three debuggers. The first is a standard procedural tracer. The second is a declarative debugger built on top of the tracer which, upon indication from the user of an incorrect trace event, attempts to find a parent event which caused the error by using the programmer as an oracle. The third and final debugger is Morphine [9], a programmable command line interface which can be used both for interactively monitoring and debugging Mercury executions.

These debuggers are textual in nature and mostly orthogonal to ViMer since they focus on different problems. Furthermore, they could be combined with ViMer to obtain a more powerful and flexible tool. In the case of the standard procedural tracer, the combination would allow the user to build and explore the tree using the perhaps more familiar trace environment. In the case of the declarative debugger, the combination could be used to better pinpoint the cause of a bug by, for example, highlighting the path traversed backwards in search of the event that caused the error. In the case of Morphine, the combination can be per-

[1] Re-execution is not correct in the presence of side-effects such as I/O. In those cases a warning is issued and the user is required to confirm the re-execution.

formed through Morphine's flexible `collect` predicate (as illustrated by [8]) and would provide the user with enormous flexibility regarding event storage, manipulation and visualization. For example, it can be used to highlight events associated to particular predicates, store and display the value of variables of interest, etc. Such combinations are, however, outside the scope of this paper.

The following sections look in detail at the more novel aspects of ViMer.

3. THE LAYERED AND-OR TREE

This section presents the new execution tree representation used by ViMer: the layered AND-OR tree. It first provides a brief look at the two most common approaches for visualizing execution trees: SLD-trees and AND-OR trees. It then presents a variation of AND-OR trees, the AORTA diagrams, which are the basis of our layered trees. Then, the basic characteristics of the layered AND-OR tree and their application to deterministic executions are presented. Finally, the characteristics of layered AND-OR trees are compared to those of the AORTA diagrams for the case of non-deterministic programs.

3.1 Earlier approaches for visualizing execution

The aim of the visualization is to provide the user with a compact and clear view of the execution flow. Most visualization tools for logic programs are based on (variations of) either SLD trees [10] or AND-OR trees [2]. SLD-trees display conjunctions vertically and disjunctions horizontally while AND-OR trees display both conjunctions and disjunctions horizontally by alternating AND and OR nodes. Figure 2 uses both approaches to illustrate the execution of goal `a` for a simple deterministic program (i.e., all its predicates have at most one answer) which contains some backtracking.

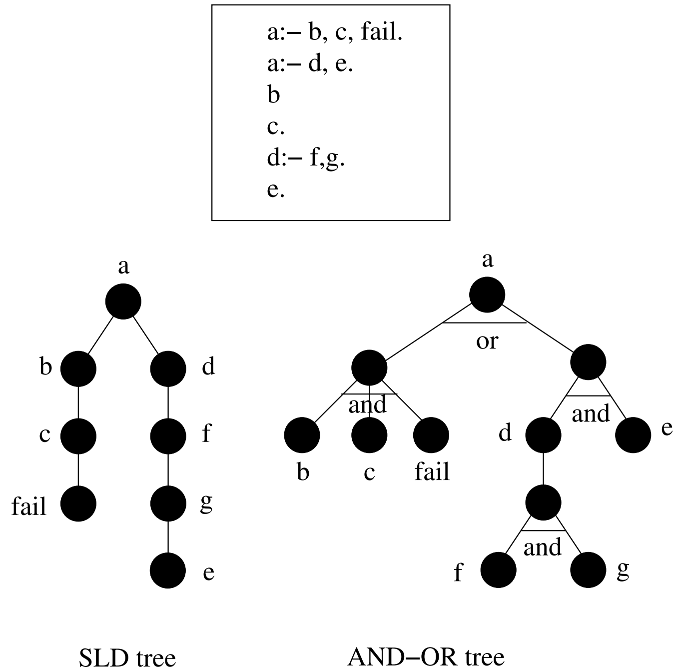


Figure 2: SLD versus AND-OR tree formats

Both formats have advantages and disadvantages. SLD-trees provide a very clear representation of non-deterministic programs since backtracking simply leads to new branches in the tree. This is true not only for shallow backtracking (the kind represented in Figure 2 where a solution is ultimately found for the predicates) but also for deep backtracking, i.e., that in which no answers are found for a predicate and execution revisits a previously successful call in search of alternative solutions. That is why a variation of the SLD-tree is commonly used in constraint logic programming to visualize variable choices: each layer in the tree represents a variable and each node represents a choice of value for that variable. Therefore, each successful branch will show one possible solution. This is the visualization used, for example, in [17].

However, executions with little backtracking yield a thin, tall SLD tree, which achieves neither a compact representation nor good insight into the execution flow. This is a particular problem for Mercury, since most Mercury procedures are deterministic. AND-OR trees, on the other hand, give rise to broader trees when representing the execution of deterministic code (see Figure 2). They also make it easier to identify the beginning and end of a particular predicate call since this corresponds to the subtree under the associated node. However, as detailed in [14] they have several problems, such as their inability to display execution of non-deterministic code which might imply the deletion of previously successful branches.

For example, let us modify the program of Figure 2 by eliminating the `fail` literal from the first clause of predicate `a`. Let us then assume that after successfully obtaining the first answer for predicate `a`, the execution proceeds executing some other goal `z` but, at some point, an error causes the execution to backtrack until it finds the second solution for `a`. Then, the tree already displayed for `z` will have to be deleted to allow for the new execution. Furthermore, even if no new solution was found, one needs to at least mark the displayed tree in some way to indicate the failure. These deficiencies led the authors of [7] to define the AORTA diagram: a variation of the AND-OR tree better suited to display execution of non-deterministic code.

AORTA diagrams differ from AND-OR trees in two main ways. First, nodes are replaced by *procedure status boxes*, which indicate the goal status (succeeded, failed, etc.), how many clauses are in the definition of the procedure, and which clause is being processed. A 'tick' in the boxes represents successful execution, a cross represents failure, a question mark represents an unknown outcome due to current execution. The boxes of all procedures in the body of a clause are connected to a smaller box representing the clause itself. The second modification is to use the clause branch in place of the OR nodes, thus allowing AORTA diagrams to "somewhat" remove an extra layer of nodes[2]. Clause branches ended in a horizontal line indicate failure, those ending in a box indicate they have been tried (the box itself might contain a tick, a question mark, etc.), and those with no entry represent untried clauses. Figure 3 shows the AORTA diagram associated to the search tree in Figure 2.

[2]The layer is not really eliminated but the small size of the clause nodes allows a reduction in space. The layer can be completely eliminated by using "the long distance view".

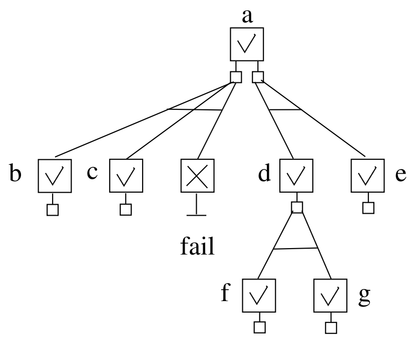


Figure 3: AORTA diagram

AORTA diagrams have some appealing aspects: they provide a more compact format and produce meaningful trees when visualizing deterministic code. However, as we will see in Section 3.3, they can still be confusing when dealing with deep backtracking.

3.2 Layered AND-OR tree: Basic representation

Our visualization format, the layered tree, is also based on the AND-OR tree. However, layered trees use three basic node types: regular predicate call nodes, disjunct nodes, and switch nodes. While call nodes are used to visualize external events (call/redo, and exit/fail/exception), disjunct and switch nodes are used to visualize internal events[3]. The tree is constructed entirely from trace information.

The process of building the basic tree for a program execution that does not involve backtracking is relatively simple. Events may result in the addition of nodes to the tree, a change in a node's status, or a change in the *current* node (where execution is presently at). Each call event produces a new node (labelled with the predicate name), which is added to the tree. The logical context of the call (e.g. whether it is made from within a negation or disjunction) is also shown, using lighter-coloured icon nodes as parents. In all cases the new node becomes the current node. On an exit or fail or exception event, the current point moves back up the tree, and the status of nodes below this point are changed to exited or failed or exception thrown, accordingly.

Like AORTA trees, layered trees display success and failure indicators at nodes. This is done by using the colour of the vertical bar above each node: blue denotes called but not yet exited, green denotes called and exited successfully, red denotes called and failed, and orange denotes called and exited with an exception. The predicate name is used to represent each node in the tree, and the current node is drawn in blue. We believe colour provides a clearer view and is a reasonable approach now that colour monitors are commonplace. Note that since our screenshots are published here in black and white, these indicators are not visible. We have instead thickened any called but not yet exited (usually blue) nodes to highlight the current state of execution, and annotated exited nodes with a tick or cross to indicate success or failure (as in Figure 6). Layered trees provide no information regarding clauses which have not been explored yet. We believe this information is not very useful, clutters the screen, and can become a serious problem in the presence of a high number of clauses. Furthermore, disjunct nodes are used to represent the different clauses. This is because Mercury's compiler merges all clauses of a predicate into a single clause with a disjunction.

[3]The external debugger interface in released versions of Mercury does not include enough information to accurately include if-then-else nodes in the tree. We have added the necessary support to our private version of Mercury, but haven't yet modified ViMer to make use of this information.

3.3 Backtracking

Let us now look at the issue of visualizing deep backtracking, i.e., a situation in which a *redo* event occurs and execution revisits a previously successful predicate call in search of alternative solutions. The difficulty lies in the fact that, as opposed to the case of SLD-trees, part of the AND-OR tree already displayed has to be modified (erased, marked, etc.) to reflect the fact that it has been backtracked over. The only previous work that provided a detailed representation of backtracking in AND-OR tree based formats was the Transparent Prolog Machine's AORTA tree backtracking representation.

Let us use the program shown in Figure 4 to illustrate this mechanism. Tree A in Figure 5 shows the AORTA tree representing the execution flow up to the point in which both `a(X)` and `c(X)` have succeeded, binding `X` to 1, and the call to `b(1)` has subsequently failed due to the failure of `d(1)`.

```
p :- a(X), b(X).
a(X) :- c(X).        b(X) :- d(X), e(X).
c(1).                d(2).
c(2).                d(3).
c(3).                e(3).
```

Figure 4: Example logic program with backtracking

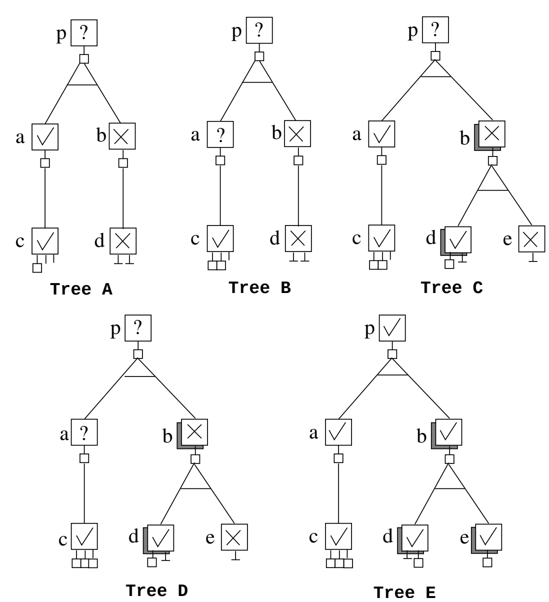


Figure 5: AORTA trees for example program with backtracking

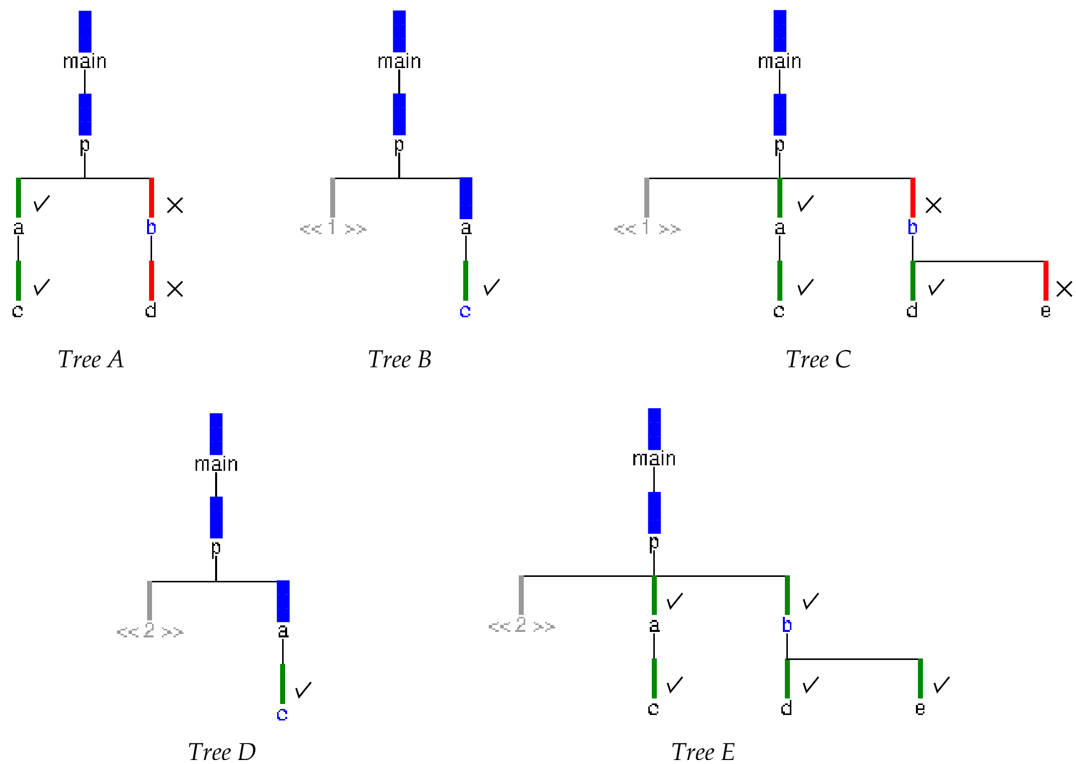


Figure 6: Layered trees for example program with backtracking

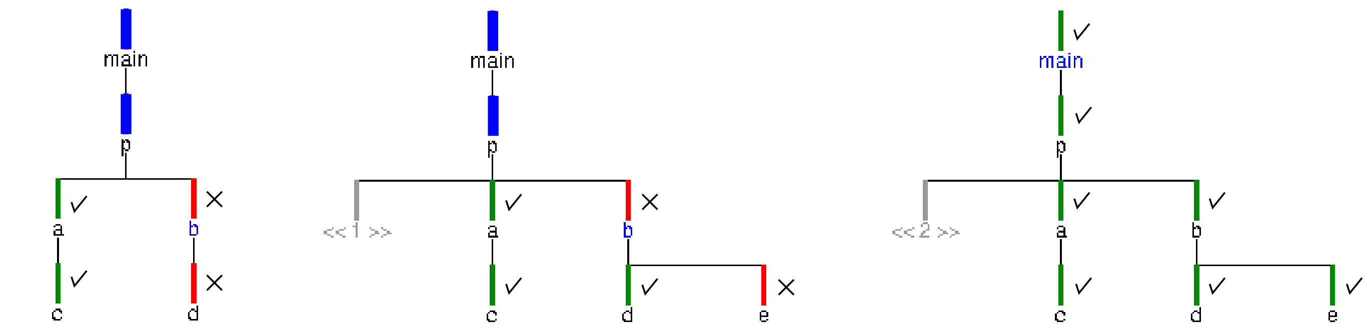
Up to this point, the execution flow is very clear since it has been strictly left-to-right. But once backtracking takes place the diagram already displayed needs to be modified. And the modification not only consists in adding new nodes, as is the case in SLD-trees, but also in modifying already displayed nodes. Firstly, the visualization has to show how the execution backtracks to the last choice-point left at `c(X)`, and succeeds when trying its second clause. Tree B shows the associated AORTA tree. Note how in this representation it is already difficult to distinguish between nodes which are currently live (`p, a` and `c`) and those which have already been backtracked over (`b` and `d`).

Tree C shows the execution up to the point in which `a(X)` has finished successfully binding `X` to 2, and the call `b(2)` has also finished with a *fail b* event after `e(2)` fails. Note that the *ghost status boxes* behind nodes `b` and `d` use depth to distinguish between an old call to the literals (`b(1)` and `d(1)`) and the current one (`b(2)` and `d(2)`). AORTA diagrams allow the user to step through these different calls by mouse-clicking on the ghost. This rewinds the execution replay to the time point when the previous invocation occurred. Note that even though ghost status boxes are useful in indicating that backtracking has occurred, they do not provide a clear link identifying which ghost nodes are associated with the same "redo layer" in the tree, that is, which nodes become hidden together since the moment execution started to backtrack (first redo event) until the moment forward execution was resumed (first non-redo event). In the example, this means that the ghost node of `b` cannot be linked to that of `d`. Furthermore, since nodes are not erased, if the tree corresponding to the second invocation is different from that associated to the first one, it can be difficult to see which nodes correspond to which invocation.

Tree D shows the execution up to the point in which `b(2)` has failed, execution returns to the last choice-point left by the disjunction within `c(X)`, and successfully exits binding `X` to 3. Once again, it is difficult to distinguish between nodes which are currently live (`p, a` and `c`) and those which have already been backtracked over (`b, d` and `e`). Finally, Tree E shows the execution up to the point in which `a(X)` exits binding `X` to 3, and `b(3)` succeeds. A ghost box is now present behind the `e` node, since more than one invocation of `e/1` now exists. Again, a quick look at the tree gives no clue regarding when and how this backtracking occurred and whether it is related to that of nodes `b` and `d`.

Our tool provides a different backtracking representation which addresses the above concerns: the execution tree is *layered*, that is, it is divided into numbered redo-layers, each of which represents the tree state before one or more *redo* events. This provides a clear link between prior invocations of a predicate, whilst also only showing the specific related subtree for each call. Furthermore, all nodes backtracked over in a single redo-layer are erased from the displayed tree and integrated into a single backtrack (or ghost) node representing that layer. Let us illustrate this by considering the layered trees in Figure 6 which show the same execution steps as those in Figure 5.

As you can see, both formats produce a similar Tree A. Layered Tree B is, however, quite different from the AORTA one: since the subtree resulting from the invocation of `b(1)` has been backtracked over and no longer forms part of the current proof tree, it is erased from the display and replaced by a single backtrack node. This node is labelled with its *redo frame number*, that indicates which layer of the tree



Redo frame 1 *Redo frame 2* *Redo frame 3*

Figure 7: Three redo-layers after execution of example logic program

(and which backtracking event) it is associated with. Layered Tree C is also quite different from the corresponding AORTA tree. First, the position of its single backtrack node clearly indicates when the backtracking occurred. Second, the tree associated to `b(2)` is a completely new tree which contains no nodes associated to previous invocations thus avoiding any confusion and reducing the complexity of the displayed tree. Tree D shows again the advantages of erasing the part of the tree which has been backtracked over, resulting in a tree which clearly shows the fact that the third choice in the disjunction is being tried. Finally, Tree E provides the ViMer tree associated to the success of `p`.

ViMer also provides a mechanism for moving between the different redo-layers, both during execution and once it is complete. We believe this provides a better understanding of the execution flow during backtracking, with each intermediate layer representing a failed proof tree, and the final layer representing the final proof tree. Figure 7 shows the three layers produced by the complete execution.

We would like to finish this section by discussing three issues. Firstly, we believe our modifications to the AORTA tree are orthogonal to the particular choice of visual representation (treemaps, hyperbolic trees, 3D, etc.) since our modifications indicate how nodes connect to each other. However, we believe 3D visualization would be the most adequate representation where each of the different redo-layers could be actually displayed using the third dimension.

Secondly, we would like to mention a related form of tree visualization, the re-computation tree, presented in [15] as a modified version of the AND-OR tree. The modifications are specifically designed to represent parallel executions combining both AND- and OR-parallelism, and in which goal re-computation is used during AND-parallel execution, i.e., if the AND-parallel goal $(a\&b)$ is executed, b is recomputed for each answer of a. The re-computation tree associates a special node to each (re)computation, displaying a different solution for a together with the complete execution of b. Two slight variations of the re-computation tree (the C-tree and the VACE tree) are also presented differing mainly on how the common parts of a's execution are displayed. There exists a clear relationship between the (re)computation nodes in the re-computation tree and the ghost nodes in our layer tree, which each (re)computation node essentially corresponding to a redo layer. However, while re-computation nodes only appear in AND-parallel conjunctions, our redo-layers can appear in any conjunction. Thus, we believe it is best to only show the most recent layer with the previous nodes collapsed into a single ghost node which indicates the position at which backtracking occurred.

The final issue is the display of built-ins. Mercury has very few built-ins, since most built-ins traditionally provided by other Prologs are actually provided as library predicates in Mercury, and are thus treated by ViMer as any other predicate. Two standard Mercury built-ins are `true` and `fail`, which Mercury treats as an empty conjunction and disjunction, respectively. As a result, there is no event specifically associated with them, and they are not visualized by ViMer. Built-in unifications and comparisons do not by default generate a trace event, but recent releases of mercury provide a compile-time option to generate events for these predicates, which then appear to ViMer as normal predicates. Higher-order built-ins, such as `call(p)`, are treated in Mercury identically to `p` by itself, and are therefore also treated identically by ViMer (there is no separate node for `call` as distinct from p). Finally, it is interesting to note that in Mercury the usual predicates for returning all solutions to a goal, such as `solutions`, are defined in a library and are therefore displayed using the default mechanism. This means that each solution gives rise to a different redo layer. We are currently investigating alternative representations for these predicates, perhaps by treating the different choices similarly to a conjunction. Such a specialised representation might also be useful for other highly-disjunctive predicates, such as those used for variable labelling in constraint logic programming applications.

4. TRACER-RELATED TECHNIQUES

Any debugger needs to traverse the large quantities of events usually produced by real-sized programs and store their associated information. In order to have an idea of the overhead introduced by this, we used Mercury's external debugger interface to perform five tests on two different Mercury programs. The first example program is an insertion-sort program sorting a list of twenty words. Its execution involves 657 events. The second is a program which solves a logic puzzle. Its execution involves 17,712 events. Note that both of these executions are small, with real-sized Mercury programs easily producing thousands or even millions of events. The following table shows the results (in seconds) of executing these programs in the following five situations:

with debugging disabled; with debugging enabled but not run through the external debugging interface; with debugging enabled without sending any trace events (i.e., jumping to the last event); tracing all events (i.e., jumping from event to event until execution's end) without requesting any information about each trace event; and tracing all events and requesting basic information about that event.

Test	*Sort*	*Puzzle*
Debugging disabled	0.258s	0.254s
Debugging enabled	0.259s	0.258s
Trace none	0.343s	0.409s
Trace all	0.782s	12.437s
Trace all and collect	0.977s	18.231s

Table 1: Timings for the external debugger interface

It is clear from the table that obtaining a complete trace of a Mercury program's execution is too time consuming even for simple examples. This conclusion can be reached even before the unavoidable overhead of visualizing such nodes is taken into account. Thus, any tool (visualizer or not) which is based on collecting, storing and manipulating execution nodes must reduce the number of events processed as much as possible. Furthermore, in order to be able to keep memory consumption down to a reasonable level, it must also reduce the amount of information stored for each event. In order to achieve this we have made three implementation decisions which are borrowed from the techniques used by standard tracers.

First, we decided that program execution would proceed step by step in increments indicated by the user, as opposed to off-line tools in which, as mentioned before, the program is first executed to completion and then presented to the user. Our choice, also taken by other visualization tools (e.g., [17, 16, 7]), results in an execution tree which is lazily and incrementally displayed as the user steps through the execution.

Second, we decided to allow the programmer to use spy-points to focus on particular parts of the program. In order to do this, our tool presents the user with a list of predicates/functions from which to select. Call or redo events for procedures not in the selected set will not be watched for or processed; no node is created for call/redo events not in the selected set.

Spy-points will not only reduce the amount of time taken by Mercury's external debugger interface in processing the events, but they will also reduce the size of ViMer's internal data structures representing the tree, the amount of communication between the debugger and the instrumented program (since the external debugger interface will only send events for predicates of interest), and the size of the tree shown to the user.

Note that the spy-points can be set dynamically, i.e., they can be added and removed during execution. In our implementation, changing spy-points does not affect the existing tree: clearly we can't in general add nodes for calls that we didn't previously trace, so keeping all existing nodes (even nodes for procedures removed from the set of spy-points) is the simplest and most consistent behaviour for this set. We have not experimented with the hiding of nodes associated with removed spy-points.

It is surprising to note that previous graphical debuggers have not provided this mechanism. The only visualization we know of which used this idea is that of [8] which illustrated how to collect different graphical views from Mercury programs by using Morphine (essentially a convenient prolog interface to Mercury's external debugger interface).

The final technique borrowed from textual tracers (and perhaps the more controversial) is due to the fact that remembering variable values at each point in the program can be very expensive, especially for large recursive data structures if one records whole values rather than just the changes. For that reason, ViMer only stores variable binding information for the nodes in the currently-live branch, discarding this information as execution proceeds. In other words, the only variable bindings whose value is automatically available are those that are currently live (i.e. variables local to a scope not yet exited), like in traditional procedural debuggers for imperative languages.

ViMer provides two ways of accessing variable bindings for nodes that are no longer on the active branch. The simplest (conceptually) is for the programmer to specify "spy variables", i.e. program variables whose value is to be memorized each time that variable's scope is entered. The more sophisticated is to use the 'retry' facility of Mercury to perform limited re-execution in order to recalculate these values. This facility uses the external debugger interface's 'retry' command to re-execute the smallest possible subtree of the execution tree which will get us to the selected node, i.e. re-executing from the closest common ancestor of the program's current position and the desired point. Because the places of interest tend to be close to the current point of execution, this typically involves much less work than re-executing the whole program. However, the current implementation of 'retry' in the Mercury debugging interface requires running the subtree to completion before restarting it. In the extreme case where the closest common ancestor is the root node ('main'), this can require more work than simply stopping and restarting the program. On the other hand, 'retry' can handle I/O better than simply restarting. For example, it can "table" the results of reads to ensure that the variables have the same values as in the first execution.

The algorithm used for adjusting and reconstructing the execution tree when the retry command is invoked is relatively straightforward: delete all children of the node, reset the node status to in-progress (i.e. change its colour on screen), and remove the redo layers that hadn't yet occurred at the initial call event of the selected node. In other words, ensure that the only nodes and redo layers left in the tree are those that were present at the initial call event of the selected node.

Note that, whole slabs of execution can be skipped and then later accessed by using the retry command. For example, one might skip a subtree (corresponding to 'step over' / 'next' in procedural debuggers), or skip until the next spy-point/breakpoint is reached. The point about "skipping" this execution is not just about not initially showing it to the user, but also about being noticeably faster and using less memory: the underlying debugging machinery needn't send trace events, and the graphical debugger needn't process or remember them, let alone manipulate tree layouts or whatever information about an execution that the debugger usually provides the user.

5. TREE DRAWING

Displayed trees are continually changing shape due to their incremental display, the exploration of different redo layers, and the use of re-execution. Thus, we need a tree display mechanism that will not only allow relatively fast drawing of the tree, but also efficient updating. Furthermore, we would like the layout to remain reasonably stationary, i.e., parts of the tree structure common between increments should not unnecessarily move in position.

In order to do this we made use of the constraint-solving toolkit QOCA, which was specifically designed for interactive graphical applications [11]. Importantly, QOCA's solutions are differentially updated: finding a new solution after adding or removing constraints involves less work than finding a new solution from scratch.

The rules currently used for determining the layout of the tree are as follows. First, the vertical position of each node is fully determined by the node's depth in the tree since there is a fixed vertical distance between a parent and its children. Second, the set of constraints on the x coordinates of nodes are (from strongest to weakest):

G: (Gap) Neighbouring nodes must be no closer than a certain distance.
L,R: (Left/Right) Parent nodes must lie between their left-most and right-most children.
S: (Siblings) The distance between the left-most and right-most children of a given parent is minimized.
H: (Half-way) Minimize the distance from a parent to half-way between its left-most and right-most children.

Constraint G ensures that nodes remain well spaced apart, and do not overlap. Constraint L,R forces a parent node to lie horizontally between its children. Optimization function S groups siblings as closely together as possible. Finally, optimization function H places a parent node as close to the middle of its children as possible, along the x-axis. A combination of these constraints is applied to each node in the tree, producing an aesthetically pleasing and clear tree layout. QOCA is then used to obtain a suitable solution and determine co-ordinate values for each tree node. As nodes are added to the tree, made visible, or hidden from view, the constraints are added or removed from QOCA.

Notice that we are *not* using the standard tree drawing convention for layered ordered trees which is to require that each parent node is centered between its children [4]. Our drawing convention has the advantage that the layout can be narrower than that obtained with the standard convention since we ultimately allow the parent to be placed anywhere between its children. On the other hand computing the layout with our convention does seem to require a linear programming approach rather than the use of a specialized linear time algorithm such as those developed for the standard convention. One of the advantages of using a generic linear constraint solving approach is that we can easily experiment with different tree drawing conventions while with standard tree drawing algorithms the convention is hard-wired into the algorithm.

Although a complete description of the algorithms used is out of the scope of this paper, let us give a brief overview of the procedure for making a node visible. Consider the example tree and constraints already applied to it, as shown in Figure 8, Tree A. Each constraint has been labelled using the above letters (G, L, R, S, H). Let us assume we wish to add and make visible a new node, F, as a child to node B and to the right of its sibling A. Tree B illustrates the constraints present after the new node is added. Note that constraints G and R relating to node A are removed, and six new constraints are added.

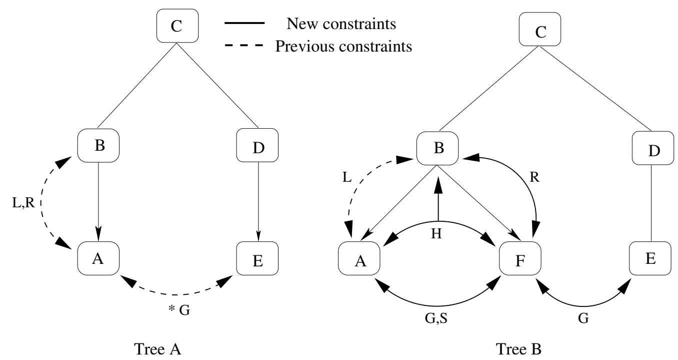


Figure 8: Constraints used to add a node

The result is an efficient adjustment of graphical constraints as the displayed tree changes shape and size. Furthermore, QOCA will retain the basic shape and structure of the tree during updates, preventing sudden changes in layout when possible. There is, however, still room for improvement: ViMer sometimes processes tree changes one change at a time, whereas some calculations can benefit from being delayed as late as possible: in effect performing the calculation once per batch of updates instead of once per update.

6. CURRENT STATUS

As mentioned before, the current implementation consists of approximately 12000 lines of newly-written and 1500 lines of modified Mercury code, and provides other features which are commonly supported by other visual debugging tools such selective hiding/expansion of subtrees, display of variable values, and source code display. The first feature allows the user to select a node, and hide (or collapse) the entire sub-tree beneath it, thus not displaying any events resulting from this predicate call. The user may also select a collapsed node and expand either a single layer, or the entire subtree below.

The second feature allows the user to access the value of variables in the current node by right-clicking on it. The user is then presented with the list of variables associated to that node and can choose to view a textual or tree representation. The former is a standard textual display. The latter uses a tree structure with functors as node names. List are treated specially, with each list element appearing as a child of a special `<<list>>` node. Figure 9 shows the same value displayed using the textual and tree formats. Note that the tree is drawn using the same algorithms used for drawing the execution tree. Also note that the user is allowed to expand and collapse components of the displayed term tree structure. We would like to extend this system to be able to use type specific representation of values. We could then incorporate tailored visualizations like those used in tools such as Grace [12]. Unfortunately, no information about

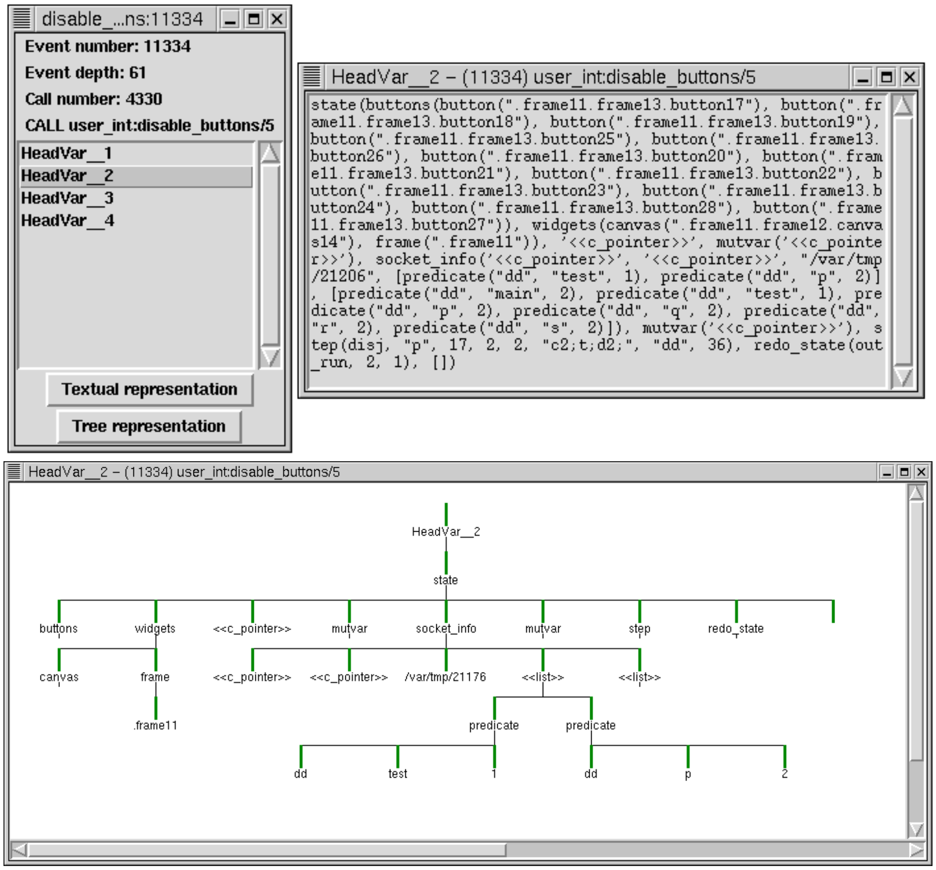


Figure 9: Variable drawing

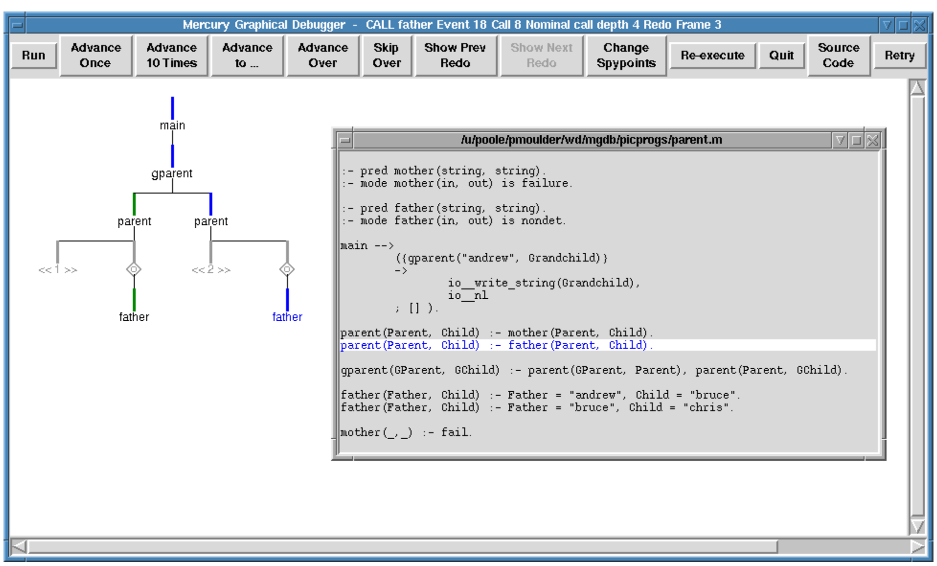


Figure 10: Source code display

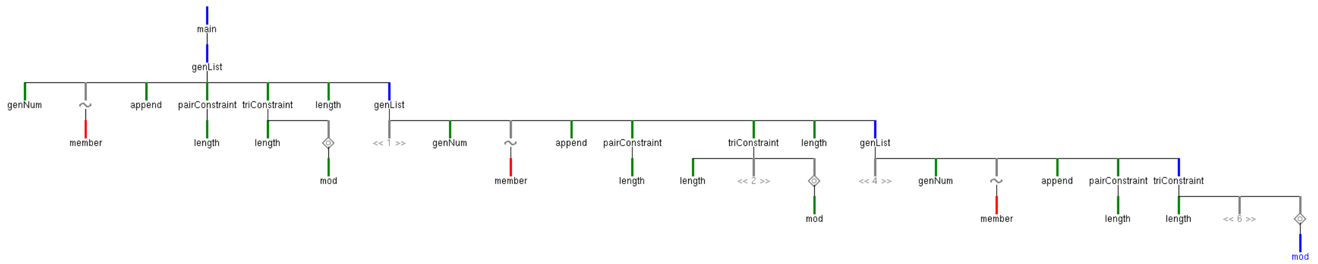


Figure 11: Part of the layered AND-OR tree for the second example

the type of the variable (other than the type's name) can be obtained via the external debugger interface yet.

The final feature allows the user to select which modules to display and displays the source code of each selected module in a separate window. The line number and module information associated with each event is used to highlight the relevant line of the code during execution, as illustrated in Figure 10.

This understanding could be further improved by illustrating the connection between the execution flow and the actual procedure generated by the compiler. This would imply having two screens per module, one showing the source code and another showing a high level version of the compiled code. Future work will investigate implementation of such a mechanism.

7. SYSTEM EVALUATION

During the later stages of its development, the debugger was shown to Mercury developers at Melbourne University, who provided feedback about the tool and changes they felt were necessary. Many of these changes were subsequently implemented. We have found the tool to provide a much clearer understanding of the execution flow of Mercury programs than standard text-based traces. The interface is effective and intuitive, and the tool runs efficiently, even when visualizing larger programs.

In order to test the tool's ability to debug real Mercury programs, ViMer was run on several small and large Mercury programs, including those with a wide range of non-determinism and backtracking. The tool was also tested on a Mercury ray-tracer program, which provided an excellent example of a large-sized Mercury program. Different sections of the ray-tracer's execution, and some large data structures created by the program were visualized quite successfully. Furthermore, our visual debugger was used to debug itself, both for testing purposes, and in two cases, to successfully locate bugs in the tool itself.

Regarding efficiency in extracting trace information, and construction and drawing of execution trees, we wish to devise some performance tests that would accurately reflect processing delays experienced by users of the tool. Typically, we would imagine that users would advance execution by as many as several hundred events, then explore some sections of the tree (by expanding collapsed branches). Therefore, we measured (a) the time required to extract the first 500 events and construct the internal tree representation and (b) the time required to expand each parent node and explore the entire tree, for two example programs. Tests were conducted on an AMD 1.2 GHz processor with 1GB of RAM.

The first example program we tested was the logic puzzle example used in Section 4, which is largely non-deterministic. Jumping to the 500th event under the debugger takes 1.3 seconds (elapsed time), and results in a layered AND-OR tree containing 62 nodes. If, instead, one takes 500 steps (one by one), the maximum time taken for a step is 0.06 seconds (the average time taken for a step was 0.007 seconds).

The second example program we tested was the debugger itself. In this example, jumping to the 500th event of its execution took 2.2 seconds, and resulted in a tree containing 229 nodes. If, instead, one proceeds step by step, the maximum time taken for a step is 0.45 seconds. (The average time taken for a step was 0.017 seconds.) Figure 11 shows part of the layered AND-OR tree displayed during the execution of the second example program.

For larger programs, one would usually select some spypoints of interest rather than visualizing the whole execution tree (which would be much too large to fit on screen). If this results in only a small number of nodes on screen (i.e. few communicated trace events and little tree layout work) then the elapsed time will be very close to the time corresponding to the 'Trace none' test in Table 1. In order to test this, we set a single spypoint on a recursive predicate and jumped to event number 50,000. This took 3.9 seconds, creating a tree containing 138 nodes. Considering that for 17,712 events the timings taken from the external debugger interface was 0.409 seconds, and that of those more than 138 events (regular call nodes usually correspond to several events) were actually traced and their information requested (last test of Table 1), the overhead introduced by ViMer seems indeed quite reasonable.

Regarding how long is required to access old variable values using re-execution, it's hard to give a good feel for it. In the worst case, it can be comparable to the time taken to run the whole program. In practice, the nodes one is interested in tend to be close to the current node. As an average case analysis, suppose that the execution tree is an n-ary tree of uniform depth. Clearly the average case depends on the probability distribution of accesses, though it's not clear what the true distribution is. If the current node and the node we wish to access are independently uniformly distributed about the tree, then the smallest subtree containing both those nodes will on average be comparable to the size of the whole tree. At the other extreme, if the probability of a retry requiring an amount W of work is something like e^{-W}, then the average amount of work required is a constant independent of the total tree size.

In our experience using ViMer, the time taken by retry has been usually less than 0.1 seconds, and rarely more than a second.

Note that our tool relies heavily upon reducing the amount of visible nodes by spying a subset of the defined predicates and by collapsing nodes. This not only aids the user in visualizing such large and complex trees, but also reduces processor time required for tree drawing. Our second example program illustrated how display times can increase substantially when more than a few hundred nodes are visible, however we do not expect that users would want to completely expand such a large and complex tree. Note that the delay involved in extracting trace information and constructing the internal tree representation is comparatively small. There is, however, room for improvement in the current algorithm, which only supports incremental update of the tree one node at a time. This creates inefficiencies when adding or hiding several nodes at once between updates of the user display, since some constraints will be added and then removed without being used to obtain coordinate values. We plan to modify the algorithm to remove this inefficiency.

8. CONCLUSIONS

We have presented the layered AND-OR tree, a tree specifically designed to visualize the execution of programs which, like Mercury's, are mostly deterministic but can contain non-deterministic predicates. We believe this tree provides a better understanding of the execution flow during deep backtracking, with each intermediate layer representing a failed proof tree, and the final layer representing the final proof tree. We have also shown how to use incremental constraint-solving capabilities to efficiently draw and incrementally update the layered tree, obtaining an aesthetically pleasing and clear tree layout.

Finally, our tool borrows several techniques from standard traces to obtain a realistic tradeoff between efficiency and usefulness. In particular, our tool does not require the entire execution to finish for it to work, it allows the use of "spy points" to specify which predicates' events are visualized in the tree, and only allows direct access to variables in nodes appearing in the currently live branch. The effect of the latter decision is softened by allowing the user to set up spy points on variables whose values will then be remembered even if not in the currently live branch, and providing re-execution mechanisms that allow the user to go back to any node already appearing in the tree.

9. ACKNOWLEDGEMENTS

We would like to thank David Jeffery for his involvement in the design of early versions of the tool.

10. REFERENCES

[1] Bouvier, P. Visual tools to debug prolog IV programs. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, pp. 177-190, 2000.

[2] Bratko, I. *Prolog: Programming for Artificial Intelligence*, Addison-Wesley, Singapore, pp. 302-329, 1993.

[3] Deransart, P., Hermenegildo M., and Maluszynski, J. Analysis and Visualization Tools for Constraint Programming: Constraint Debugging. Lecture Notes in Computer Science, 1870, Springer Verlag, 2000.

[4] Di Batista, G., Eades, P., Tamassia R., and Tollis, I.G. Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall, 1999.

[5] Ducassé, M. Opium: An extendable trace analyser for Prolog *Journal of Logic Programming 39*(4), pp. 177-223, December, 1999.

[6] Carro, M. and Hermenegildo, M. The APT tool. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, pp. 237-252, 2000.

[7] 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, December, 1988.

[8] Jahier, E. Collecting Graphical views of a Mercury program. In *2000 International Workshop on Automated Debugging*. http://xxx.lanl.gov/abs/cs.SE/0010038

[9] Jahier, E. and Ducassé Morphine 0.2 User and Reference Manuals. IRISA, Rennes, 1999

[10] LLoyd, J.W. *Foundations of Logic Programming*, Springer-Verlag, New York, 1987.

[11] Marriott, K., Chok, S.S. and Finlay, A. A tableau based constraint solving toolkit for interactive graphical applications. In *Principles and Practice of Constraint Programming - CP '98*, pp. 340-354, 1998.

[12] Meier, M. Debugging constraint programs. In *Principles and Practice of Constraint Programming - CP '95*, pp. 204-221, 1994.

[13] Ousterhout, J. *Tcl and the Tk Toolkit*, Massachusetts: Addison-Wesley, 1994.

[14] Pain, H. and Bundy, A. What stories should we tell novice Prolog programmers. In *Artificial Intelligence Programming Environments*, Wiley, New York, 1987.

[15] Vaupel, R., Pontelli E. and Gupta G. Visualization of And/Or-Parallel Execution of Logic Programs. In L. Naish (Ed.), *Proceedings of the 14th International Conference on Logic Programming*, Cambridge, pp. 271-285. MIT Press, July 8-11, 1997.

[16] Schulte, C. Oz Explorer: A visual constraint programming tool. In L. Naish (Ed.), *Proceedings of the 14th International Conference on Logic Programming*, Cambridge, pp. 286-300. MIT Press, July 8-11, 1997.

[17] Simonis, H. and Aggoun, A. Search-Tree Visualisation. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, pp. 191-208, 2000.

[18] Somogyi, Z., Henderson, F. and T. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. In *Journal of Logic Programming 29*(1-3), pp. 17-64, 1996.