Visualization of And/Or-Parallel Execution of Logic Programs[1]

Rick Vaupel, Enrico Pontelli, Gopal Gupta
Laboratory for Logic, Databases, and Advanced Programming
New Mexico State University
Box 30001, Dept. CS
Las Cruces, NM 88003
{rvaupel,gupta,epontell}@cs.nmsu.edu

Abstract

In this paper we consider the problem of designing a graphical tool for visualizing and-or parallel execution. We first present a paradigm for visualizing and-or parallel execution. This paradigm is based on an abstract representation of and-or parallel execution called the recomputation tree. We present a general purpose, interactive, both static and animated, tool called VACE that realizes this paradigm. Interesting features of VACE and their implementation are discussed.

1 Introduction

An important property of logic programming languages is that they are declarative, allowing for parallelism to be exploited *implicitly* during their execution. That is, parallelism can be exploited without requiring any help from the programmer. Three broad types of (implicit) parallelism have been identified in logic programs: *(i) Or-parallelism:* arises when multiple clauses define some predicate and a goal unifies with more than one clause head—the corresponding bodies can then be executed in or-parallel fashion. *(ii) Independent and-parallelism:* arises when arguments of a subgoal are bound to structures that do not share any variables with other subgoals. Such *independent* goals can be executed in parallel. *(iii) Dependent and-parallelism:* arises when two dependent goals cooperate in producing an answer.

Several parallel logic programming systems have been built [2, 11, 16, 18, 17] that implicitly exploit individual forms of above types of parallelism as well as their combinations. In theory, in these systems parallelism can be exploited completely implicitly, i.e., without involving the programmer at all. However, experience reveals that, in general, it is a good idea for the programmer to be aware of the fact that his program is going to be executed in parallel. Also, most parallel Prolog systems give the programmer some degree of control regarding which goals are to be run in parallel and which ones sequentially. One can say, then, that while the programmer's input is not needed in parallelizing a program, *his input is needed in tuning the program's performance.* Given that the programmer, in general, will not be responsible for parallelizing the program, but will be responsible for performance-tuning it, we need some means of informing the programmer how the program has been parallelized and how the parallel execution develops. Providing a tool that enables the programmer to visualize parallel executions is one solution. A programmer can run his program automatically in parallel, and then use this visualization tool to analyze and fine-tune the parallel execution.

Importance of building tools for visualizing (parallel) execution has been emphasized outside of logic programming [13, 12, 15] as well as inside [5, 3, 19, 1, 14, 20, 15]. However, most tools built so far for visualizing parallel execution of logic programs, such as Wamtrace [5], MuseTrace [19], Visandor [3], etc. have been designed to visualize only one kind of parallelism (either or- or and-parallelism). The main reason for this is that these tools were designed for systems that exploited only one kind of parallelism, thus their designers did not feel the need for providing support for both and- and or-parallelism. The Visandor tool is perhaps one exception. The designers of Visandor recognized that the structure of a purely or-parallel computation is similar to that of a purely and-parallel execution. Thus, the same tool can be used for visualizing both purely or- as well as purely and-parallel execution. Visandor can also be used for visualizing execution under systems like Andorra-I [4], where the or-parallelism and and-parallelism are cleanly separated, and and-parallelism is of a restricted variety. It should be noted that Visandor doesn't support arbitrary combination of or-parallel and (non-deterministic) and-parallel execution. Furthermore, these systems offer only one view of the execution: either a static global one or a dynamic and "local" one. Experience shows that both representations are needed. Visandor and MuseTrace, have been combined (ViMust) in an attempt to fulfill this need.

In this paper we present a tool called VACE [21] that bridges these gaps, and that can be used for visualizing (statically or dynamically) parallel execution of logic programs with arbitrary nesting of or- and and-parallelism. To the best of our knowledge VACE is the first tool to have this capability.

The VACE tools works in conjunction with the ACE and-or parallel Prolog system [17]. VACE, like other tools for visualizing parallel execution of logic programs, takes a trace of events as input, and uses the information in the trace to visually represent execution. The event trace is produced during the actual parallel execution; generation of traces is activated by a special user accessible flag. Thus, building a tool for visualizing parallel execution involves not only the design of the graphical visualization system, but also the format of the event-trace. Note that while VACE is designed to work with the ACE system, it can work with any other parallel system with a compatible trace format. In fact, VACE trace format is upwardly compatible with Visandor's [3] and MuseTrace's [19] format.

For visually representing and-or parallel execution, VACE relies on a derivative of the *Composition tree* abstraction, developed [8] for representing and-or parallel execution. We only consider and-or parallel computation with goal recomputation [9], as that produces the most complex execution tree, and is also the evaluation strategy adopted in most systems (ACE, PBA, &-Prolog). In and-or parallel execution with goal recomputation, all goals $G_{i+1} \dots G_n$ to the right of a goal G_i in a parallel conjunction $G_1 \& \dots \& G_n$ are executed in their entirety for every solution produced for G_i [11, 9]. In contrast, in and-or parallel execution with goal reuse, solutions to goals in the parallel conjunction are found, and their join taken, to avoid redundant execution [10]. And-or parallel execution with goal reuse can be abstractly represented as an *extended and-or tree* [8, 9]. An extended and-or tree is a two dimensional structure, so representing it visually is not very difficult. However, representing Composition trees (subsequntly referred to as a *C-tree*) visually is a much harder problem as it is a 3-d structure.

VACE is an interactive tool. It provides both static and animated views of the C-tree representing the parallel execution.

The rest of the paper is organized as follows. We first discuss ways of abstractly representing and-or parallel execution with goal recomputation. This abstract representation serves as the vehicle for visualization of and-or parallel execution. Next, we present the features of the VACE tool, along with the design of the interface between the ACE parallel Prolog system and VACE. The implementation issues involved in development of VACE are discussed. Finally, we discuss uses of VACE and present our conclusions.

2 Abstract Representation of And-Or Parallel Execution

2.1 The Recomputation Tree

A programmer (as well as an implementor) must have an abstract view of and-or parallel execution in order to understand how an and-or parallel system is parallelizing his program. For example, the SLD tree is a good abstraction for purely or-parallel execution, and the and-tree is a good abstraction for purely and-parallel execution. The tools designed for purely or-parallel and purely and-parallel execution indeed use the visual representations of these abstractions. For and-or parallel execution, the structure that naturally comes to mind for abstractly representing it is the classical and-or tree [8]. However, the classical and-or tree turns out to be a bad choice. In and-or parallel systems, typically, only some of the conjunctions are annotated (via compile-time analysis) for execution in and-parallel, all other conjunctions are processed sequentially from left to right. This is because not all conjunctive goals can be fruitfully executed in parallel to reduce total execution time. The classical and-or tree is organized assuming that all goals may be executed in parallel; there is no way to depict a conjunction that is sequentially executed. Furthermore, it is extremely difficult to capture in the and-or tree certain "execution" properties, like recomputation of subgoals. Another problem with the and-or tree is that goals belonging to a single computation (that produce one solution) may be spread all over the and-or parallel tree. Thus, environments corresponding to distinct solutions are not easily identifiable.

Two alternative representations for abstractly representing and-or parallel execution have been proposed by us: the *extended and-or tree* [10, 8] and the *composition tree (C-tree)* [9, 8]. The extended and-or tree is designed for abstractly representing and-or parallel execution with goal reuse, while the C-tree has been designed to represent parallel executions with goal recomputation. We describe next an intuitive way of representing and-or parallel execution. We term this representation the *recomputation tree*. The C-tree, described later, is a slight variant of the recomputation tree. The execution tree displayed by VACE (called the VACE tree) is itself a slight variation of the C-tree, and is described subsequently.

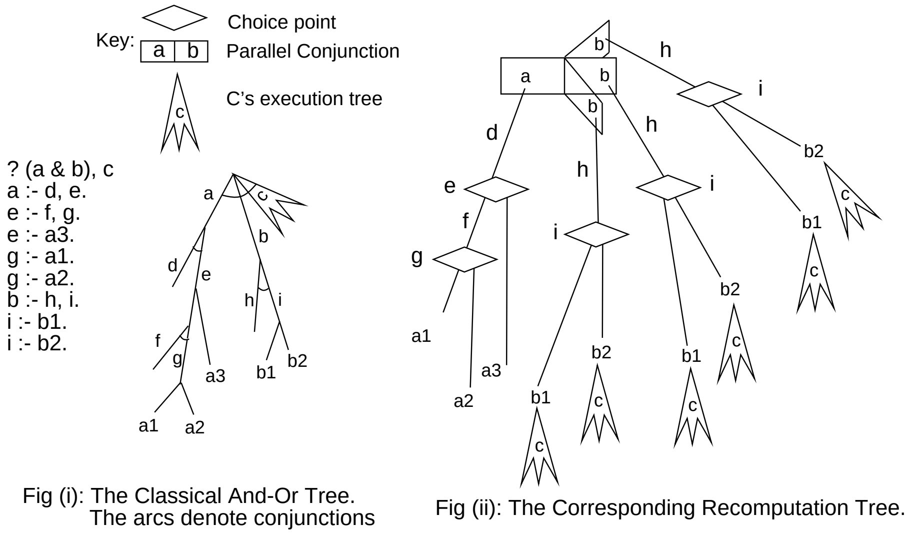


Figure 1: And-Or Parallel Computation w/ Goal Recomputation

To illustrate the structure of *Recomputation Trees* consider the goal `(a & b), c`, where '&' denotes parallel conjunction, and `','` sequential conjunction. The classical and-or tree for this goal is shown in Figure 1(i). With recomputation, for every alternative of `a`, the goal `b` is computed in its entirety. Suppose all three solutions for `a` are found in or-parallel. Intuitively, the structure we want to build is shown in Figure 1(ii). The three instances of `b` created correspond to each of the three solutions for `a`. These three instances of `b` may exist simultaneously as the solutions to `a` are computed in or-parallel. Also, the three instances of `b` share the information for the goal `a`, thus they are shown as a *book* in Figure 1(ii). The book-like structure holds control information about the and-parallel execution of subgoals `a` and `b`. A branch of the subgoal `a` together with one of the 3 instances of `b` constitutes a distinct parallel conjunction. There are 3 parallel conjunctions corresponding to the three solution for `a`. In Figure 1(ii) we assume that each incarnation of `b` is executed sequentially; if all incarnations of `b` are also executed in or-parallel we will have 6 parallel conjunctions (as now each of the solutions of `b` will form a parallel conjunction with one of the solutions of `a`). In other words, each separate combination of `a` and `b` that is executed in parallel represents a parallel conjunction. Again, referring back to figure 1(ii), each parallel conjunction consists of the *page* corresponding to `a` and one of the three *pages* corresponding to `b` in the *book* like node. These two pages will hold control information about and-parallel execution of that parallel conjunction. The choice points (diamond shaped nodes), on the other hand, hold control information regarding or-parallel execution. The three trees for different invocations of `b` in Figure 1(ii) share parts of the tree for `a`. All three share the execution of `a` up to its first choice point labeled `e`. The left two invocations of `b` share `a`'s execution up to its second choice-point (labeled `i`). The execution trees of the continuation of `(a & b)`, i.e., `c`, shown together with `a` and `b` in the and-or tree, hang from the corresponding solutions of `b`.

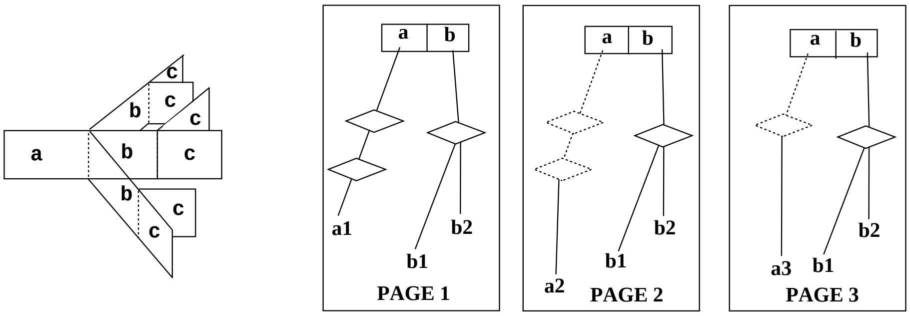


Figure (i): Structure for (a&b&c)   Figure (ii): Recomputation tree projected on pages

Figure 2:

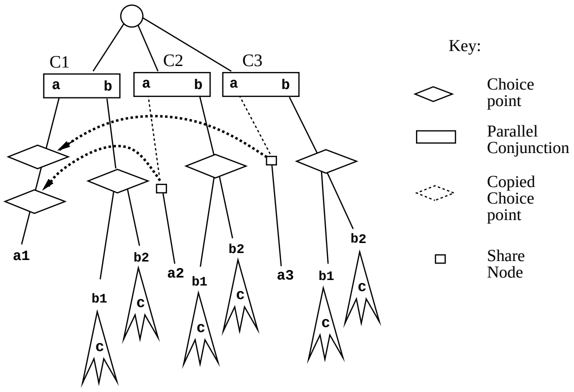
Note that in figure 1(ii), the book like structure, can also be represented as a tree, if we choose to stay within purely graph-theoretic representation.[2] If the parallel conjunction had more than 2 goals, then this book-like structure might further split as shown in Figure 2(i), becoming a book of books. Thus, the recomputation tree can be thought of as a three dimensional figure where, roughly, the horizontal x-axis represents and-parallelism, the vertical y-axis represents sequential execution, and the z-axis or-parallelism. Thus, we can visualize the recomputation tree as a collection of sheets (Fig. 2(ii)) where 2 or more sheets might share some common (or-) branches (in Figure 2(ii) a computation that was performed by a given sheet is shown in solid in that sheet; if that computation is needed in another sheet then it is shown dotted in that other page). The tree on each sheet corresponds to execution of one of the three parallel conjunctions.

2.2 The Composition Tree

In Figure 1(ii), the control structure created for `a` (the page for `a` in the book like node) is shared by the three parallel conjunctions. From an implementation point of view, each parallel conjunction will have to be represented separately. Thus, an implementation based on recomputation trees will indeed have to maintain the view that is shown in Figure 2(ii). If the collection of parallel conjunctions shown as a book earlier in Figure 1(ii), is split up into separate parallel conjunctions (Figure 2(ii)), then we obtain what has been referred to earlier as the *Composition tree* [9]. The Composition tree corresponding to 1(ii) is shown in in Figure 3. The Composition tree (C-tree) can be thought of as a 2-d representation of the 3-d recomputation tree. The various parallel conjunctions (shown as separate rectangles in the C-tree) all have common parent node. To make this explicit, all parallel conjunctions are shown as children of a special node (shown as a circle in Figure 3. The

[2] Note that there is a class of graphs in graph theory called *books* [6].

2-d C-tree representation also shows the sharing of or-branches very clearly through the *share nodes* and side-ways arrows (See figure 3; the side-ways dotted arrows allow access to the shared parts). Note that in the C-tree solid lines represent computation, while dotted lines represent shared computation. The 2-d C-tree forms the basis of the execution tree displayed by the VACE tool.



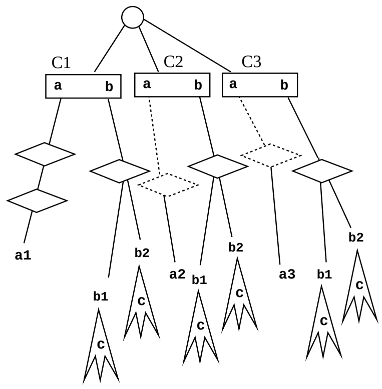


Figure 3: The C-Tree

Figure 4: VACE Tree

2.3 The VACE Tree

The execution tree displayed by VACE (called the *VACE tree*) is based on the C-tree. The side-ways links needed in the C-tree for representing sharing of computation of goals to the left in a parallel conjunction are very difficult to draw using a graph drawing tool. This is because actually the C-tree is not really a tree, it is a directed acyclic graph (due to the presence of side-ways links). To overcome this problem, in the tree diplayed by VACE, parts that are shared in different parallel conjunction are explicitly shown in full in each parallel conjunction. Figure 4 shows the tree corresponding to the C-tree of Figure 3. The distinction between the original computation and its copies is shown by using solid lines and dotted lines respectively (in VACE colors are used to represent this). This fits well with ACE, where shared computations are physically copied.

2.4 An Example

Let us illustrate the construction of a VACE tree with an example. In ACE the shared computations are physically copied, via copying of stacks/heaps [17]. ACE organizes the processors available for parallel execution into *teams*. Each team consists of several processors (*agents*). Or-parallelism is exploited between different teams, while and-parallelism is exploited within each team.

Let us consider the construction of the tree in figure 4. Consider the conjunctive goal `(a & b), c` along with the program of Figure 1. First, the non-determinism of goals `a` and `b` needs to be represented: a circular node is used to collect together the various parallel conjunctions that will be created for `(a & b)`. Let us suppose a team T1, consisting of processors P1 and P2, begins execution of the top level query. T1 first creates the top circular node[3] that collects the multiple parallel conjunctions that will be created by the parallel call `(a & b)`, and places as its first child the parallel conjunction `C1` under which it will execute `a` and `b` in and-parallel. The goal `a` will by executed by P1 and `b` by P2. As executin proceeds new choice-points that lead to alternative solutions of goals `a` and `b` are created. Or-parallelism can be exploited by executing alternatives of these choice-points in parallel. A new team T2, say, consisting of processors P3 and P4, can move in and pick up one of the untried alternatives of the goal `a` (from choice point labeled `e`) from team T1. T2 will copy the stacks/heaps corresponding to computation up to the node `e`, and then its processor P3 will begin computing the solution `a3`. Because we have chosen to follow Prolog semantics, goals to the right need to be recomputed. Hence the goal `b` will be recomputed from scratch by processor P4. The execution of the alternative of `a`, and the recomputation of goal `b` is done under a new parallel conjunction node (node C3 in Figure 4). The copied part of `a` is shown explicitly under `C3`. Another team T3, say, consisting of processors P5 and P6, can pick up work from the choice point labeled `g` created by team T1. T3 will create the parallel conjunction C2, and continue execution similar to team T2.

3 Structure of the VACE Tool

The VACE Tool displays the VACE tree. Different colors are used to color branches and nodes created by different teams/agents. An example VACE tree is displayed in Figure 6. Recall that in the VACE tree, shared computations are shown duplicated in each parallel conjunction. The original computation is shown in the color of the team that created it, the copied computation is shown in dual colors—the two colors belong to the team that was copied from, and the team that was copied to (See Figure 6). It is important to note that the duplication of computations is a property of the *graphical representation*—thus it does not necessarily imply that copying is the method used to support non-determinism in the parallel implementation. VACE can draw the execution tree either statically or dynamically (i.e. as an animation). In dynamic display mode, the various nodes are colored in the order they were created by different processors producing an animation effect.

In the VACE tree shown in figure 6, the diamonds represent parallel choice-points, the wide rectangles represent parallel conjunctions, the tall, thin rectangles represent end of a goal in a parallel call, and the circles (at a leaf node) represent ends of successful computations. When displayed dynamically, the VACE tree changes as execution proceeds. New nodes appear as they are created, while nodes that are backtracked over "disappear". At any given point in the computation it is desirable to know the parts of the computation that were performed and then got backtracked over. Thus, in

[3]Observe that the structures mentioned here are only present in the graphical representation—e.g., no circular node is present in the actual ACE computation.

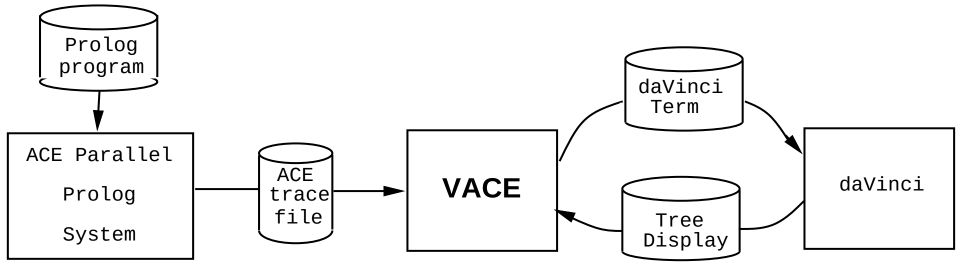


Figure 5: Interaction between VACE and ACE

VACE the backtracked over parts are never made to disappear. Nodes that have been backtracked over are colored in black (during forward execution, a node will be shown in the color of the team that created it). Edges that have been backtracked over preserve their color but are shown as dotted lines.

The input to VACE is a trace file that is generated by the ACE system (see Figure 5). The trace file is read in, the corresponding VACE tree is computed, and displayed. VACE's display of the computation is based on a graph display tool called daVinci [7]. Displaying graphs, specially dynamically, is a difficult problem, as numerous decision need to be made during graph layout regarding distances between displayed nodes, routing of edges so that the graph doesn't appear cluttered, etc. Having a ready made tool that displays graphs thus is extremely helpful. In Figure 6 the whole tree is shown at the moment the first solution for the top level query is found. All subtrees generated by all the teams that participated in the execution are shown. In VACE it is possible to de-select some of the teams/agents and draw only the subtree generated by the remaining teams.

The ability of VACE to support both static and dynamic representations of the computation is of extreme importance. Static view of the execution allows us to have a global idea of the properties of the computation (balance of work distribution, amount of parallelism exploited, etc.)—covering the purposes for which Visandor was developed. Dynamic view of the execution is particularly useful to understand the behavior of the execution, e.g. for debugging purposes. This covers the purpose for which tools like MuseTrace were developed. Thus, VACE collects the relevant features of the previously proposed visualization models in a uniform framework.

3.1 The ACE Trace Files

The first step in the development of VACE was the design of the execution trace format. The format of ACE trace definitions is based upon the trace format used for Visandor. In fact, the ACE trace definition is strictly a superset of Visandor trace definition. Thus, any Visandor trace can be visualized in VACE—subject to some limitations, as Visandor traces are not fully geared to take advantage of VACE's facilities. These limitations are mainly related to Visandor's lack of support for backtracking and teams.

A trace file consists of a series of records, where each record represents one event. Visandor assumes that the first record is a code that identifies type

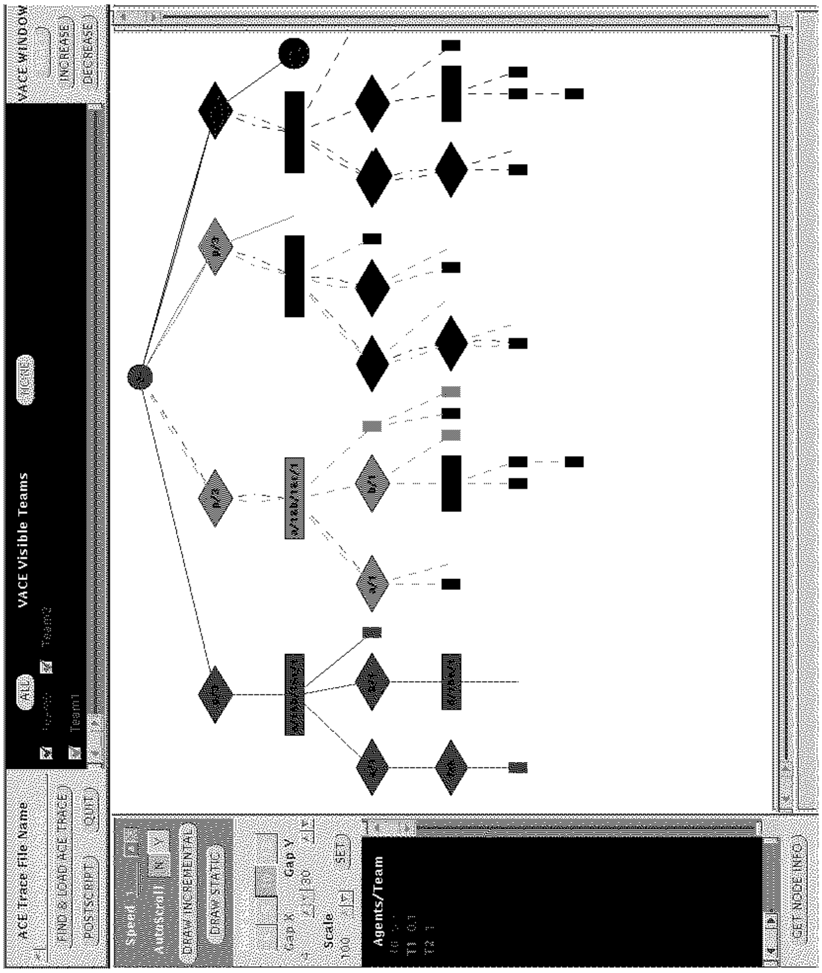


Figure 6: VACE screen dump

of the trace. A 0 value represents and-parallel execution, while 1 represents or-parallel execution. As an extension of Visandor, ACE traces use the value 3 to denote and-or parallel execution. The second record in the trace file identifies the number of teams used in the execution. The rest of the trace contains the events describing the execution (one per line). Table 1 shows the format of a VACE record (which is an extension of Visandor's format).

TimeStamp	event_code	NodeId	Branches	WamId	AgentId	TeamId	Name
unsigned	integer	hex	integer	hex	integer	integer	char[8]

Table 1: Format of a record in the ACE Trace

The various fields have the following meaning: the Timestamp is the time at which the event occurred; Event code identifies the event type; Nodeid is a unique id used to identify each node. The stack address of that node within ACE execution is used as an id. Branches indicates the number of branches (e.g., subgoals in a parallel call). WAMid identifies the copied-to team in the copy events. Agentid and TeamId identify respectively the processor and the team that generated this event. Name is used to indicate the name of a predicate/subgoal. Table 2 describes the events recognize by VACE. The first group is basically the set of events recognized by Visandor (and also by VACE), while the second part of the table contains the events specifically introduced in VACE. Visandor also has events for or-parallelism. However, these turned out to be inadequate for our needs, and thus new events for or-parallelism were included in the trace format.

Event Code	**Event Name**	**Event Description**
1	Fork	creation of and-parallel conjunction
2	Start Goal	beginning of and-parallel subgoal
3	End Goal	termination of and-parallel subgoal
4	Join	completion of and-parallel call
5	Start Time	begin trace
6	End Time	end trace
41	Chp Create	creation of choice point
42	Alternate	selection of new choice point alternative
43	kill	remove computation rooted at a parallel call
44		Backtrack from a node
45		Backtrack to a node
46	Start Copy	begin duplication of Or-parallel computation
47	End Copy	completed duplication of Or-par. computation
48	Copy Chpt	duplicate choice point
49	Copy Pcall	duplicate parallel conjunction
51	Copy Generic	duplicate generic structure
52	Goal End	termination of a subgoal
53	Copy Branch	duplicate complete and-parall. branch
60	Chpt Name	associate predicate name to choice point
61	Pcall Name	associate pred. name to parallel call

Table 2: Events Description

As can be noticed, the trace format allows for the description of copying operations—thus it is geared towards a copying approach for or-parallelism. This choice is justified by two reasons:

- duplication of "shared" computations is required in the representation model adopted in VACE to describe and/or-parallel computations;
- in case of schemes in which common computations are shared during or-parallelism (e.g., Binding Arrays), the trace format includes an additional event (Event 70—*Steal Clause*), which is automatically mapped by VACE on a corresponding sequence of events 44–51.

It is interesting to observe that this set of event is also sufficient to display the execution of various schemes for Dependent And-Parallelism (DAP). This is because from a visualization point of view, there is no difference between dependent and independent and-parallelism. Experiments are in progress to generate and display DAP traces.

Traces are collected automatically during parallel execution. During program execution, each computing entity (processor) collects the events it generates in a local heap, time-stamped with the help of a unique global clock. Only at the end of the execution the content of these heaps is merged and properly sorted, to produce a time-ordered sequence of events. A feature of VACE, whose possibilities have not been completely explored yet, is the ability of specifying *which* events are to be collected during the execution. By default all the mentioned events are collected, as soon as they occur in the execution. The top-level of ACE supports a set of declarations that allow to selectively enable/disable tracing of certain events. This allows, for example, to obtain simpler representations for very complex computations (e.g., those having very complex nesting of and- and or-parallelism). The facility to disable certain events can also be provided at the level of VACE. The addition of this facility to VACE is planned in the near future.

3.2 VACE Features

Figure 6 shows the interface that VACE offers to the user. The various buttons located on the interface allow the user to interact with the visualization process in various ways. The following options are supported: (i) Input/Output Options, (ii) Tree Drawing Options, (iii) Team Display Options, (iv) Tree Resizing Options, (v) Window Options.

Input/Output Options: The commands in this control area handle (i) input, output, and termination of the VACE application, (ii) loading the trace files, (iii) saving the VACE tree as a Postscript file.

Tree Drawing Options: The commands and setting in this control area are used to manipulate the different tree drawing mechanisms available within VACE. The SPEED setting option allows the user to modify the speed at which the VACE Tree is incrementally animated (colored). The AUTO-SCROLL command is used to turn on and off the automatic scrolling of the daVinci canvas during a DRAW INCREMENTAL animation (this is very important since the computation tree can grow quite large and is likely to not fit in the window frame). The DRAW INCREMENTAL command removes color from all the VACE Tree nodes, then incrementally adds team and backtracking colors back to the nodes in an animated fashion. The DRAW STATIC command draws the entire VACE Tree in one step. If a node has previously been selected, this command will only draw those nodes that were created prior to, and including, the selected node. This feature is similar to the *time line* feature of `ViMust`. Each node of the tree can be selected and information regarding that node (failed or active, name of the associated predicate, etc.) obtained using the GET NODE INFO button.

Team Display Options: VACE allows visualization of the complete execution performed by all the agents available. Furthermore, it supports as an added feature the capability of representing teams of processors (as used in the Andorra-I, ACE, and other and/or-parallel system). The side window, called *Agents/Team* maintains a description of the organization of the processors in the system (i.e., which agent is in which team) at every step.

The top window (*VACE Teams*) contains a box for each processor (in case of a single form of parallelism) or for each team (in case of combined exploitation of and/or-parallelism). By selecting or de-selecting these boxes, it is possible to select the part of the computation that we want to visualize. If we de-select certain entries, then the computation produced by the corresponding agents will not be presented on the screen. This allows the user to focus on the behavior of specific computing agents.

In the case of and/or-parallelism, each team is assigned a different color (this assignment is described in the *VACE Teams* window). The part of the computation tree that has been developed by a team is drawn using the color of that team. If we focus on the work of a single team (de-selecting all the other teams), then VACE will automatically recognize this situation and reassign the colors to the agents composing the selected team, so that the computation performed by each agent in the selected team can be clearly seen. This adequately matches the logical structure of the combined and/or parallel computation: in systems like ACE and PBA [9] and-parallelism and or-parallelism are layered (or-parallelism is exploited at the higher layer—between the various teams, while and-parallelism is exploited at a lower layer—within each team). This feature of VACE allows a user to move easily from one layer to the other, with the focus getting automatically switched via reassignment of colors.

4 Examples

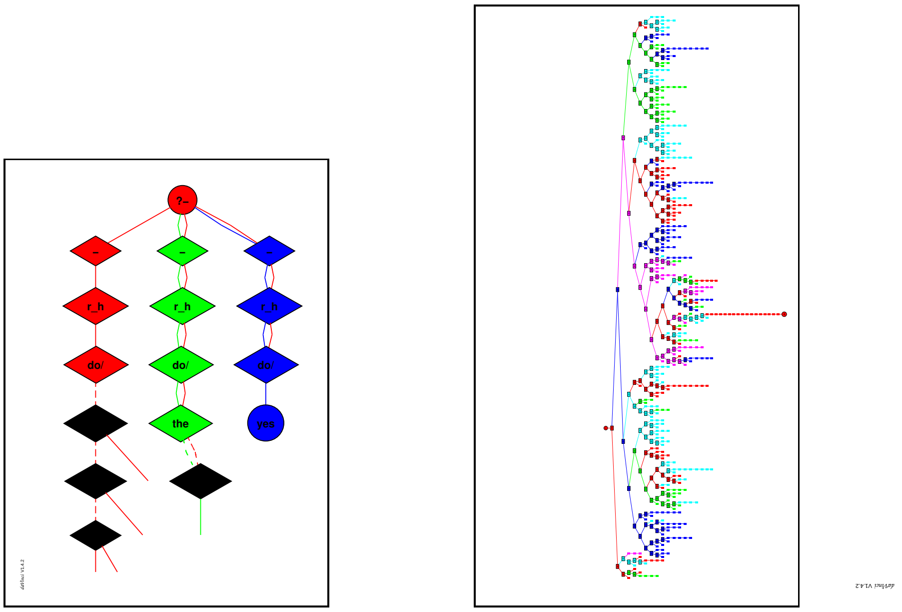


Figure 7: Sample VACE Displays

The picture to the left in Figure 7 illustrates the global tree described (before the final backtracking) for a simple or-parallel computation obtained from a simple theorem prover (3 processors have been used). The picture to the right in Figure 7 is the dump obtained from the and-parallel execution of Quicksort (using 5 processors). In both the cases (as well as in all the examples considered), the dump is relative to the endpoint of the execution, immediately before the final global backtracking. The use of colors to distinguish the actions of the different agents gives a global view of the distribution of work (the tool as been extremely useful in improving the scheduling mechanisms of ACE; situations where the tasks distribution was unbalanced were easy to pinpoint). The picture to the left in Figure 8 is an

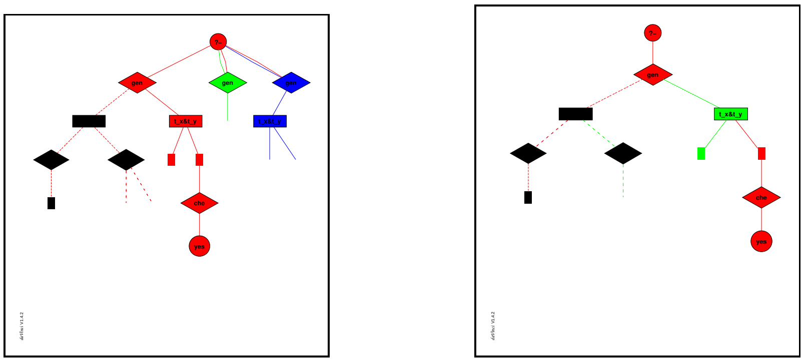


Figure 8: VACE tree for Cryptarithmic Puzzle

and/or-parallel execution obtained from a simple cryptoarithmetic puzzle—implemented as generate&test, where and-parallelism is present in the test phase and a single nondeterministic predicate is used to perform generation. Three teams of two processors each were used. This picture illustrates the finding of the first solution (by the first team). The picture on the right in Figure 8 has been obtained focusing on the computation developed by the first team in solving the cryptoarithmetic puzzle—i.e., by de-selecting from the *VACE Teams* window all the teams but the first. VACE automatically reassigns the colors to the different agents composing the team—which facilitates understanding how the and-parallel work has been distributed inside the team.

5 Generality and Uses of VACE

VACE is quite a general-purpose tool. It can be used for visualizing purely and-parallel execution, purely or-parallel execution, as well as and-or parallel execution. The and-or parallel execution that can be visualized using VACE can be of the most general form, i.e., with arbitrary nesting of and-parallelism and or-parallelism. Of course, VACE can be used to visualize restricted forms of and-or parallel execution such as that found in the Andorra-I system. Thus, the VACE system is the first system to allow visualization of both and- and or-parallelism (both statically or dynamically) simultaneously.

The VACE tool has been quite helpful during the implementation of the ACE system. VACE can also be used to qualitatively estimate how well the load is balanced across various processors. This is because nodes created by each team are shown in different colors. The amount and distribution of different colors in the display is an indication of how evenly work was distributed among different processors. Also, the animation feature is very helpful in showing a user the order of events. A user can select a node and see the order in which nodes/edges were created in an animated fashion until the time the selected node was created. Also, selective inclusion of teams in the display allows the user to focus on the computation of a single team.

Although VACE is designed with the ACE and-or parallel system in mind, it can be used with other and-or parallel system as well, such as the Paged Binding Array model [9], as long as a compatible event trace file can be generated. The design of the event trace file (Section 3.1) for VACE might appear to be specific to ACE (there are events for stack-copying etc.), however, the same trace format can be used for other and-or parallel models as long as the execution model is based on goal recomputation. It should also be noted, that while the current focus of VACE has been to represent combined or- and independent and-parallel execution, it can easily be used for visualizing combined or- and dependent and-parallelism (and, hence, for combined or-, independent and-, and dependent and-parallelism).

An additional feature that makes VACE even more interesting is the fact that the routines for manipulating the generation of event-trace in ACE are made available at the level of Prolog. ACE supplies to the user a set of builtins which allow him to include or exclude selected events during execution. Originally this facility was added for debugging purposes, but it turned out to be of great utility. It allows the user to generate traces with fewer events, which can be more easily visualized. Also, the user can force generation of an event at specific points of the execution and trace them through VACE. This can be used, for example, to introduce tracing facilities in any meta-interpreter written in Prolog (which can then be used for graphically visualizing the computation performed by the meta-interpreter). This facility can also be used to visualize sequential Prolog execution through display of the SLD tree, by restricting the events that are recorded. (Though, it should be noted that this filtering could also be done from within VACE on the trace file after it has been generated, provided the appropriate buttons to filter the trace file are provided in VACE).

6 Conclusions

VACE is a novel, powerful tool for visualizing and-or parallel execution of Prolog programs. It is based on recomputation tree (C-tree) representation of and-or parallel execution. VACE has a number of features that make it useful for studying parallel execution behavior of a given Prolog program. VACE has been used for performance tuning of benchmark programs for ACE. It has been an indispensable tool during the development of ACE. While there are quite a few tools for visualizing parallel execution of Prolog programs, none of them are designed for visualizing both and- and or-parallelism. In this regard, VACE is the first tool of its kind.

We are currently expanding the capabilities of VACE, with particular attention towards the design of a simple interface to allow the introduction of user-defined events in the trace.

Acknowledgments

We want to thank the developers of daVinci, Michael Frohlich and Mattias Werner, as well as the Visandor team at UPM and MuseTrace team at SICS for their help. Thanks also to the anonymous referees whose comments considerably improved the paper. This work is partially supported by NSF Grants CCR 96-25358, HRD 96-28450, and INT 9415256, and NATO Grant CRG 921318.

References

[1] S. Aikawa et al. A Graphical Tuning Tool for Multiprocessors Systems. In *Int. Conf. on FGCS*. ICOT, 1992.

[2] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *N. American Conf. on Logic Prog.* MIT Press, 1990.

[3] M. Carro et al. Some Event-driven Paradigms for the Visualization of Logic Programs. In *Proc. Int. Conf. on Logic Programming*. MIT Press, 1993.

[4] V. Santos Costa et al. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd ACM SIGPLAN PPoPP*, 1990.

[5] T. Disz and E. Lusk. A Graphical Tool for Observing the Behaviour of Parallel Logic Programs. In *Proc. Symp. on Logic Programming*. IEEE Computer Society, 1987.

[6] R. Duke and F. Harary. Generalized Ramsey Theory VI: Ramsey Numbers for Small Plexes. *Journal of Australian Mathematical Society*, 22(4):400–410, 1976.

[7] M. Frohlich and M. Werner. *DaVinci V1.4 User Manual*. University of Bremen, 1994.

[8] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Press, 1994.

[9] G. Gupta, M. Hermenegildo, and V. Santos Costa. And-or parallel prolog: A recomputation based approach. *New Generation Computing*, 11(3,4):297–322, 1993.

[10] G. Gupta and B. Jayaraman. And-or parallelism on shared memory multiprocessors. *Journal of Logic Programming*, 17(1):59–89, 1993.

[11] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Int'l Conf. on Logic Prog.*, MIT Press, 1990.

[12] V. Herrarte and E. Lusk. Studying Parallel Program Behaviours with Upshot. Tech. report, Argonne National Labs., 1992.

[13] E. Kraemer and J.T. Stasko. The Visualization of Parallel Systems: an Overview. *Journal of Parallel and Distributed Computing*, 18(2), 1993.

[14] A.J. Kusalik and G.M. Oster. Towards a Generalized Graphical Interface for Logic-programming Development. Technical Report 93-2, University of Saskatchawan, 1993.

[15] A.J. Kusalik and S. Prestwich. Visualizing Parallel Logic Program Execution for Performance Tuning. In *Proc. JICSLP'96*. MIT Press, 1996.

[16] E. Lusk and al. The Aurora Or-parallel Prolog System. *NGC*, 7(2,3), '90.

[17] E. Pontelli and G. Gupta. Parallel Symbolic Computation with ACE. *Annals of AI and Mathematics*. (to appear).

[18] K. Shen. Initial Results from the Parallel Implementation of DASWAM. In *Int. Conf. and Symp. on Logic Programming*. MIT Press, 1996.

[19] C. Svensson and J. Sundber. MusTrace: a Graphical Tracer for Or-parallel Prolog. Technical Report T-90/9003, Swedish Institute of Computer Science, 1990.

[20] E. Tick. Visualizing Parallel Logic Programming with VISTA. In *Int. Conf. on Fifth Generation Computer Systems*, 1992.

[21] R. Vaupel. VACE: a Visualization System for ACE. Tech. Rep. NMSU-CSTR-9513, Dept. Computer Science, NMSU, 1995.