A UNIFIED APPROACH TO CONCURRENT DEBUGGING

APPROVED BY
DISSERTATION COMMITTEE:

To My Father and Mother

Without their sacrifice, continuous encouragement, and

great expectations, I may not have undertaken

this work and persevered to take

it to completion.

A UNIFIED APPROACH TO CONCURRENT DEBUGGING

by

SYED IRFAN HYDER, B.E., M.S (EE), MBA

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment of

the Requirements for

the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1994

Acknowledgments

I am indebted to my wife for her support and understanding. She sustained me through times when I was not making progress and was, thus, often unreasonable, and through times when I was making progress and, thus, vigorously at work away from home. I also owe my children an apology for not having spent enough time with them.

I am full of gratitude for my family. Their continuous support, help and shouldering of the responsibilities made possible the completion of this work.

I also thank my parents for their patience and waiting as I tried my best to finish the work as soon as possible. Their encouragement has been a continuous source of inspiration in my life.

I acknowledge the contribution of my supervisor, Dr. J. C. Browne, and thank him for asking the right questions. It was only the search for the answers to these questions that clarified my confusions and guided me to the correct understanding of the problem.

A UNIFIED APPROACH TO CONCURRENT DEBUGGING

Publication No. _____________

Syed Irfan Hyder, Ph.D.

The University of Texas at Austin, 1993

Supervisor: James. C. Browne

Debugging is a process that involves establishing relationships between several entities: The behavior specified in the program, P, the model/-predicate of the expected behavior, M, and the observed execution behavior, E. The thesis of our approach is that a consistent representation for P, M and E greatly simplifies the problem of concurrent debugging, both from the viewpoint of the programmer attempting to debug a program and from the viewpoint of the implementer of debugging facilities. Provision of such a consistent representation becomes possible when sequential behavior is separated from concurrent or parallel structuring. Given this separation, the program becomes a set of sequential actions and relationships among these actions. The debugging process, then, becomes a matter of specifying and determining relations on the set of program actions. The relations are specified in P, modeled in M and observed in E. This simplifies debugging because it allows the programmer to think in terms of the program which he understands. It also simplifies the development of a unified debugging system because all of the different approaches to concurrennt debugging become instances of the establishment of relationships between the actions.

We define a formal model of concurrent debugging in which the entire debugging process is specified in terms of program actions. This unified model of concurrent debugging places all of the approaches to debugging of parallel programs such as execution replay, race detection, model/predicate checking, execution history displays and animation, which are commonly formulated as disjoint facilities, in a single, uniform framework.

We have also developed a feasibility demonstration prototype of a debugger implementing this unified model of concurrent debugging in the context of the CODE 2.0 parallel programming system. This implementation demonstrates and validates the claims of integration of debugging facilities in a single framework. It is further the case that the unified model of debugging greatly simplifies the construction of a concurrent debugger. All of the capabilities previously regarded as separate for debugging of parallel programs, both in shared memory models of execution and distributed memory models of execution, have been given an implementation in this prototype.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Debugging is a process that establishes a relationship between the program (typically some small segment of a large program) and its execution behavior. The process involves several entities and relationships between those entities. It starts with a program that has been observed to produce invalid final states for one or more initial states. The segment of the program that is suspected of being faulty is selected for monitoring. Expectations about the execution behavior of the suspect program segment are specified in a model or a predicate. The program is, then, run and its actual execution behavior is observed. The actual execution behavior is checked against the model/predicate to reveal any unexpected behavior. Mapping of the unexpected behavior back to the program brings the programmer closer to the bug. This completes one cycle of a process that is repeated until the bug is located.

The entities involved in the debugging process include the program, P, the model/predicate of the expected behavior, M, and the actual execution behavior, E. Ideally M and E should be expressed in a representation consistent with the program P so that the programmer is not forced to understand and manipulate several different notations. Additionally, the facilities provided by a debugger in each part of the debugging process should help in manipulating and establishing relationships between the entities involved in that part of the process.

Debugging of even sequential programs becomes difficult when debuggers for conventional text string languages use unrelated and typically informal representations for M and E. The problem exacerbates for concurrent programs written in pure text forms which often require a different representation for each of the three entities; P, M and E. Such programs often express concurrency by adding synchronization and communication primitives to the sequential text. This produces a complex entanglement of the concurrent considerations of synchronizations and communications with the sequential considerations of flow of control and flow of data. This entanglement gives rise to ambiguities among various parts of the debugging process for concurrent programs by turning each part of the process into a separate problem (Section 2.2). These ambiguities obscure the relationship between the different representations used for the entities. Debugging facilities that establish relationships among P, M and E appear to be incompatible or even orthogonal. Incompatibility of the facilities for different parts of concurrent debugging forces the programmer to either use different facilities for different parts of the cycle or debug without them. Use of multiple representations for P, M and E typically compels the debuggers to either constrain the range of behaviors that can be checked [ReSc94]; or to tolerate the ambiguities in the observed behavior [EGP89], [HMW90], [NM91a]; or to demand extra programming effort [SBN89], [LMF90], [Bat89].

Using separate representations for different portions of the debugging process introduces special problems into the debugging of concurrent or parallel programs. Many approaches to debugging parallel programs appear to be different when approached conventionally. There is a long list of supposedly different debugging facilities for concurrent programs: execution replay facilities [LM86], [MiC89], [Net93], race detection facilities [NM91b], [Sch89], predicate/model checking facilities [Bat89], [HsK90], [WaG91], execution history displays [PaU89], [FLM89], [Ho91] and animation facilities [PaU89]. This multiplicity of different views of concurrent debugging forces the programmer to learn many different representations (Section 2.1.2).

1.1 The Unified Approach

The thesis of our approach is that a consistent representation for all of the different entities (P, M and E) involved in the debugging process greatly simplifies the problem of concurrent debugging, both from the viewpoint of the programmer attempting to debug a program and from the viewpoint of the implementer of debugging facilities. Provision of such a consistent representation becomes possible when sequential behavior is separated from concurrent or parallel structuring. Given this separation in the representation, the program becomes a set of sequential actions and relationships among these sequential actions. The abstractions in this representation, which are defined below, have a natural graphical representation. We will use the graphical representation in all of our discussions, although textual representations capturing the graphical structures are equivalent.

The debugging process for concurrent programs, then, becomes a matter of specifying and determining relations on the set of program actions. These are specified in the program, modeled in the expected behavior and observed (recorded) in the actual execution behavior (Chapter 3). This simplifies the task of debugging because it allows the programmer to think in terms of the program which he understands. It also allows for the automation of the tedious tasks of establishing relationships between the entities. Moreover, the use of actions allows a clean separation between the measurement parts of a debugging tool and the analysis parts of debugging tools. This separation makes the task of the developer of the debugging systems for concurrent programs much more simple as all of the different approaches to debugging of parallel programs become instances of the establishment of relationships between the program's actions.

1.1.1 The Abstraction of Computation Actions

Def. 1-1 An *action* is an operation for which there exists a known input/output relation for a given initial state.

Although concurrent debuggers define execution events to be the execution occurrences of program actions, they often leave the specifications of actions implicit. Unlike other approaches that typically use events, the approach described in this dissertation is formulated in terms of actions. The unified model of concurrent debugging formalized in Chapter 3, debugs the concurrent behavior in terms of relations on the set of *computation actions*:

Def. 1-2 A *computation action* is a piece of program text that starts and/or ends with a synchronization statement.

The abstraction of computation actions and the "causality" of their dependence relations allows the unified approach to disentangle concurrent synchronization and communication from sequential flow of control and flow of data. A computation action can be viewed as consisting of three parts as shown in Figure 1-1. (i) A condition specified on the action's input dependences that determines when the action should start its execution. (ii) A sequential computation that the action will execute. And, (iii) a condition specified on the action's output dependences that determines what follows after its execution. Thus, computation actions interact with each other through their dependences. They start executing their internal sequential computation when their input dependences are satisfied. They end execution by enabling data on their output dependences.

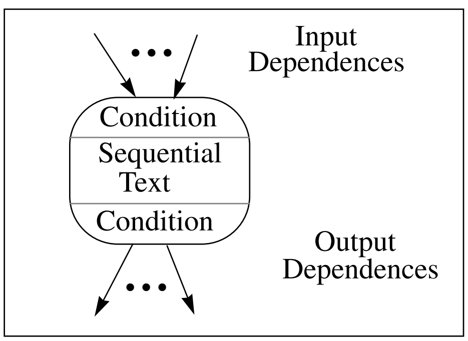


Figure 1-1. Abstraction of a computation action.

The abstraction of a computation action decomposes the concurrent debugging problem into two almost disjoint problems that can be approached at different levels. A programmer debugs the concurrent state (Section 3.2.3) at the upper level, where the only important concerns are the relations on the set of computation actions. Internal states of a computation action are not important at this level. They only become important when the programmer moves to the lower level, inside the action to debug the internal sequential computation of the action.

The use of computation actions provides a graphical representation for the program where nodes are the computation actions and where arcs represent the dependences between the actions. Computation actions are naturally available in graphical visual programming languages like CODE 2 [New93], the language for which we have implemented the unified debugger (Chapter 4). Computation actions can also be obtained from a textual representation of the program as explained in Appendix A.

1.1.2 Program and its Execution

The key concept in providing a consistent representation is that both the structure of a program and its execution behavior have natural representations as graphs when actions are represented at an appropriate level of abstraction. The program, P, is a directed graph (perhaps not defined until runtime, Section 3.1) whose nodes are sites for the execution of computation actions, and arcs are the dependences with which the actions synchronize and communicate.

The execution, E, of a program (Section 3.2) is the traversal of the runtime instantiation of the program graph starting with an assignment to an initial state, until the instantiation of a final state. Traversal of the graph causes execution of actions at the nodes and generates a partially ordered set of execution occurrences of actions or events.

Def. 1-3 *An Event* is an execution of the action at a node of the program graph.

Therefore, each event maps to the action of which it was an execution occurrence. This provides a unique identifier for each event that consists of the id of the action and its execution count. The unique identifiers allow the debugger to record the execution, E, as a partial order on the set of events. As each action is capable of executing multiple number of times, the recorded partial order is a “pomset” [Pra86]; a partial order on the multiset of occurrences of actions as events (Section 3.2.2).

In the recorded partial-order, the orderings indicate much more than a mere temporal order. They indicate the dependences that “cause” the actions to execute at the nodes. A debugger can, then, collect the dependence information from the orderings of different executions of the same action, and deduce the conditions that govern the execution of the action (Section 3.2.2). Therefore, a programmer can describe the expected behavior, M, as some conditions on the expected dependences of selected actions. Then, the debugger can observe the execution orderings of those actions, and deduce the conditions governing their execution. It can raise an exception if they don't match the expected behavior (Section 3.3). This guides the programmer towards the offending action. Thus,

Def. 1-4 *Debugging* is the process of identifying those actions of the program that are responsible for the failure of the program to meet its final state specification.

1.2 The Debugging Process

The use of actions by the unified model of concurrent debugging provides a consistent representation for all the entities (P, M, and E) involved in the debugging process (Chapter 3). This not only simplifies the task of the programmer, but also simplifies the provision of debugging facilities that establish relationships between the entities. The feasibility demonstration prototype of the unified model of concurrent debugging has been implemented in the CODE 2 environment (Chapter 4). It covers all the different parts of the debugging process and provides facilities that:

1. Record and display the actual execution behavior of the program.

2. Restrict the recorded execution behavior to selected actions of the program.

3. Allow the user to specify a model/predicate of the expected behavior.

4. Automate the checking of the expected behavior.

5. Display any unexpected behavior that is detected during checking. This display allows the user to map the unexpected behavior to the program.

6. Provide post-restriction of the recorded information to selected actions.

7. Make cyclical debugging possible by providing a replay capability.

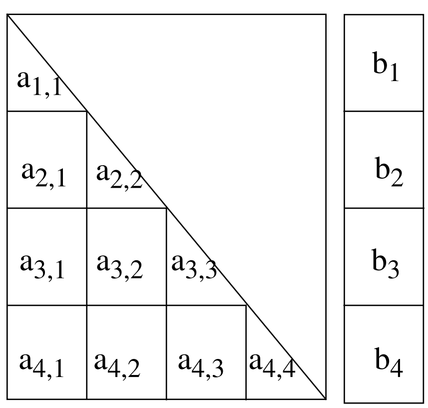
8. Provide support for interactive debugging.

Note that the provision of all of the above facilities is made possible by the same recorded information; namely, the "causal" orderings among the executions of actions (Chapter 3).

1.2.1 Block Triangular Solver Example

A parallel program for a Block Triangular Solver algorithm is used to illustrate the debugging process. The problem is to solve the system **Ax** = **b** for a dense lower triangular matrix **A**. The algorithm is quite simple and involves dividing the matrix and the vector into blocks as shown in Figure 1-2(a). Each "a" in the figure represents a sub-matrix of **A** and each "b" represents a sub-vector of **b**. Let the number of sub-blocks be **N**.

The algorithm replaces **b** with the solution vector **x**. The case for **N** = 4 is shown in Figure 1-2(b). Notice that once b_j has been computed, the operations $b_i = b_i - a_{i,j}b_j$ can be performed in parallel for i = j+1 to N. Thus, the algorithm proceeds iteratively, working on columns of the blocked system one at a time from left to right.



$$b_1 = a_{1,1}^{-1} b_1$$

$$b_2 = a_{2,2}^{-1} (b_2 - a_{2,1} b_1)$$

$$b_3 = a_{3,3}^{-1} (b_3 - a_{3,1} b_1 - a_{3,2} b_2)$$

$$b_4 = a_{4,4}^{-1} (b_4 - a_{4,1} b_1 - a_{4,2} b_2 - a_{4,3} b_3)$$

Figure 1-2. (a) Matrix, **A** and vector, **b**. (b) Replacing **b** with vector **x.**

Let **Solve** be a sequential function that solves this problem (applied to a single block).

```
To process the j-th column do
  Solve(a_{j,j}, b_j);
  for each i from j+1 to N do
     b_i = b_i - a_{i,j} * b_j;
```

Each of the iterations of the `for` loop can be done in parallel. Let **Mult** be a computation that does $\mathbf{b_i = b_i - a_{i,j} * b_j}$. The parallelism in this algorithm stems from the ability to perform the **Mult** computations "beneath" the **Solve** computation for a column in parallel. This is readily seen in the data-flow graph for the algorithm as shown in Figure 1-3. The "**s**" nodes are calls to **Solve** and the "**m**" nodes are calls to **Mult**.

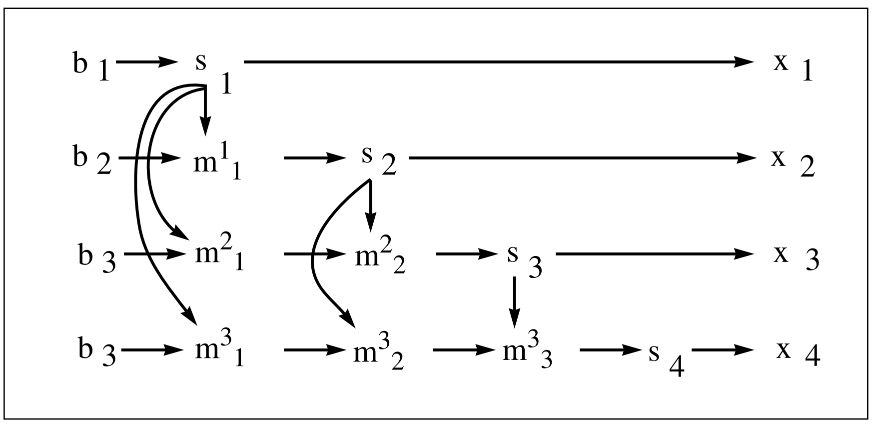


Figure 1-3. Data-flow for Block Triangular Solver.

Figure 1-4 shows an implementation of this program in the CODE 2 graphical/visual parallel programming environment. Node **Dist** sends the appropriate segments of **b** to the nodes that perform the **s** and **m** operations of Figure 1-3. A single instance of node **Solve** performs, one after another, all of the **s** operations, whereas **N** - 1 instances of node **Mult** perform the **m** operations. Node **Gath** collects the segments of **x** from each execution of the **Solve** node and combines them into the single vector **x**. The arc leaving **Mult** implies an iteration. First $\mathbf{s_1}$ is done, and then $\mathbf{m^i{}_1}$ is performed in parallel by **Mult** instances whose indices range from **i** = 1..**N**-1. Next $\mathbf{s_2}$ is done followed by $\mathbf{m^i{}_2}$

performed in parallel by **Mult** instances whose indices range from **i** = 2..**N**-1, and so on.

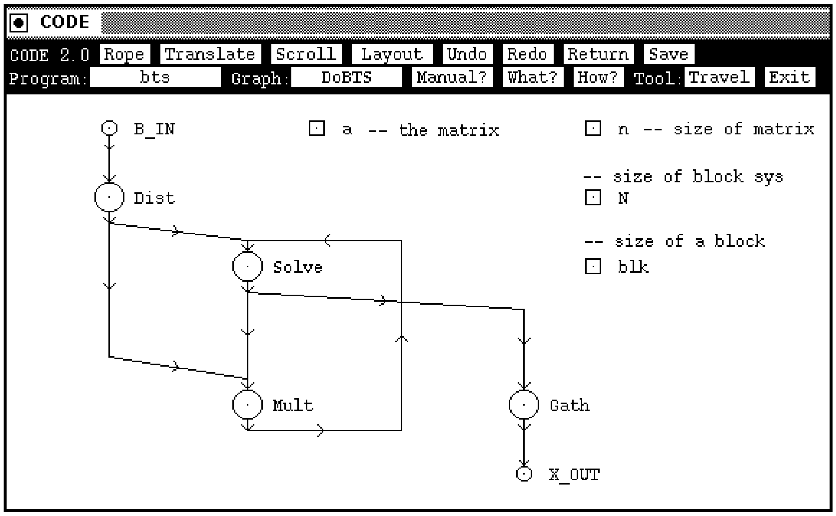


Figure 1-4. CODE 2 Graph for Block Triangular Solver (DoBTS)

Note that the nodes of a program graph are type templates. The number of instances of a node that actually execute is determined at runtime (Section 4.2.1). In the above example, at runtime, there were **N-1** executable instances of node **Mult** and one instance each of nodes **Solve**, **Dist** and **Gath**. Note the distinction between the nodes specified in Figure 1-4 and their executable instances at runtime. The instances of templates are the *executable* computation actions of the unified model of concurrent debugging described in Chapter 3, and are referred to as computation actions or actions.

1.2.2 Different Parts of the Debugging Process

Suppose that a deliberately introduced bug in the above example causes a sequencing error between **Solve** and **Mult** actions. The execution of this bugged version starts with an initial state where **N**=4, and terminates with a segmentation fault. We now follow the different steps of the debugging process:

1. Identify and select the portions of the program whose behavior is to be monitored.

This is a set of "suspect" nodes or subgraphs. Note that it is typically impossible to monitor the entire execution behavior of the large complex programs which are actually the ones that need debugging. The visual/graphical representation of P makes the selection of suspect portions of the program easy. In our example, we can click on the **Solve** and **Mult** nodes of the graph of Figure 1-4 to inform the debugger that they need to be monitored. The debugger makes additional preparations to filter out event executions of other nodes like **Dist** and **Gath** (Section 3.3.2). This greatly helps in later steps as much of the irrelevant information is filtered out.

2. Specify the expected execution behavior of the set of nodes that are to be monitored.

The natural mode of representation of execution behavior for graphical programs is the partially ordered set of events expected to be generated by the execution of the actions at the nodes of the suspect subgraphs. Let us call this representation, M, for Model of the expected execution behavior. M is given as a partially ordered set of events. We can either construct this set of events directly, or construct a graph of actions whose execution will generate the desired partially ordered sets of events (Section 3.3). In this case, we specify M by drawing a graph of **Mult** and **Solve**. See Figure 1-5(a). In the data-flow description of Figure 1-3, note that $\mathbf{s_i}$ indicates that **Solve** works on the $\boldsymbol{i}$-th sub-block when it executes for $\boldsymbol{i}$-th time. And $\boldsymbol{m^j{}_i}$ indicates that the $\boldsymbol{j}$-th instance of **Mult** works on the $\boldsymbol{i}$-th sub-block when it executes for the **i**-th time. The specification of the expected behavior can, then, state that if an execution of **Solve** is preceded by an execution of a **Mult** action, then the index of the **Mult** action is equal to the block number on which it worked. Note in Figure 1-3, that $\boldsymbol{m^1{}_1}$ precedes $\boldsymbol{s_2}$, $\boldsymbol{m^2{}_2}$ precedes $\mathbf{s_3}$ and $\boldsymbol{m^3{}_3}$ precedes $\boldsymbol{s_4}$.

3. Capture the execution behavior of the selected portions of the program.

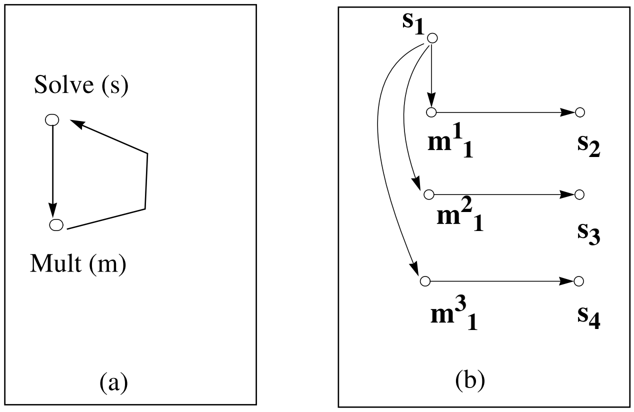


Figure 1-5. (a) Graph of M, (b) Graph of E, (c) Elaborated graph of M.

This is a partially ordered sequence of events that actually occurred in the execution (Section 3.2). Let us call this partially ordered set of events E. This is obtained by annotating the program graph with specifications to record only the events and the orderings resulting from the execution of the suspect nodes or subgraphs. The selection of **Mult** and **Solve** nodes in step 1 produced such an annotation. As a result, the actual execution behavior observed by the debugger as shown in Figure 1-5(b), contains event executions of only the selected nodes (Section 3.3.2). In the figure s_i indicates the i-th execution of **Solve** and event $m^j{}_i$ indicates the i-th execution of that executable instance of **Mult** whose node index is j.

4. Map E to M to determine the locations where the actual and expected events first diverge.

The mapping of E to M can be done automatically since they are specified from the same representation. The result is identification of event sequences in E that do not correspond to the allowed set defined in M. In Figure 1-5(b), we note that events $m^1{}_1$, $m^2{}_1$ and $m^3{}_1$ precede event s_2, s_3 and s_4, respectively. We, however, expected event $m^j{}_j$ to precede event s_i, where $j = i-1$. This detects the occurrence of the unexpected behavior; event $m^2{}_1$ preceding s_3, and event $m^3{}_1$ preceding s_4. The events map to actions m^2, m^3 and s as shown in Figure 1-5(c). Apparently, actions m^2, and m^3 sent data to the action s in their wrong executions. ($\boldsymbol{m^j}$ should have sent data when its execution counts was equal to $\boldsymbol{j}$, i.e. during event $\boldsymbol{m^j}_j$.)

Note that mapping of E to M gives an elaborated graph of M as shown in Figure 1-5(c). This is a run-time structure that shows dynamically created instances of node templates (Section 3.1.1). The elaborated graph is obtained from the partial order graph of E in Figure 1-5(b) by folding back subsequent executions of a node, to its first execution.

5. Map the elaborated graph of M back to P to define corrective action.

Since the elaborated graph of M contains instances of the node templates of P, the mapping is automatic, and guides us towards the offending action in P. The mapping from Figure 1-5(c) to Figure 1-4, helps in identifying the offending action. Note that we ascertained above that data is sent by actions of **Mult** in the wrong executions to ***Solve***. By looking at the specification of the output rule of **Mult**, we found that the data was being sent out to **Solve** without checking that the index of the action was equal to its execution count (or the count of the sub-block which it was solving).

1.3 Overview of the Unified Debugger

The use of actions allows us to separate the analysis and presentation concerns of the debugger from its measurement and recording concerns (Chapter 4). Instrumentation inserted in the actions is only responsible for controlling their executions and generating the event information (Chapter 5). The analysis and presentation of this information is separately carried out by the debugging facilities (Chapter 6). The interactive facility allows the user to control the execution of actions and query their state at runtime (Chapter 7). See Figure 1-5. The event information generated by the instrumentation during the recording run, replay run, or the restricting run of the actions, can either be used by the facilities on-the-fly, or in a postmortem fashion. The event information is first topologically sorted to ensure a causal arrival of events (Section 6.3). Then, depending upon the options selected by the user, it is given to one or more of the following facilities that provide:

1. Animation and execution history displays (Section 6.4).

2. Checking of the expected behavior (Section 6.5).

3. Post-restriction to interesting events (Section 6.1).

4. Filing for postmortem display or checking (Section 6.1.2).

Note that the trace event records from an earlier execution are used for replay.

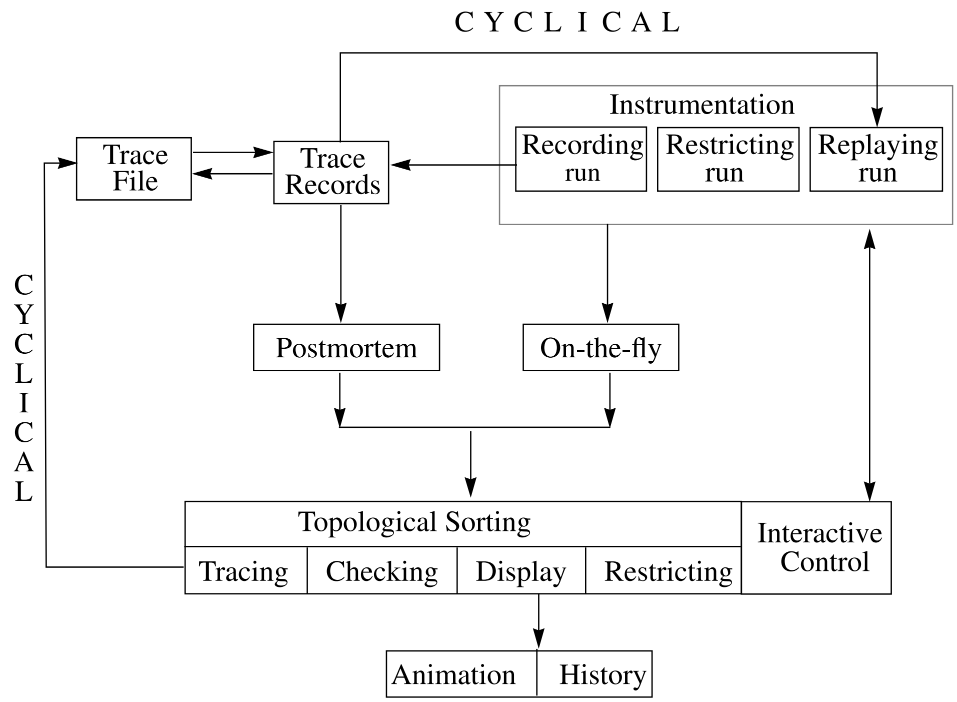


Figure 1-6. Available facilities

1.3.1 The Actual Execution Behavior

An event is identified by the id of the action and its execution count. Instrumentation inserted in each action is only responsible for recording its execution and informing its successors about its event id. The execution event record of each action contains the ids of its predecessors. This information is used to construct a partial order representation of E that provides a definition of the concurrent state (Section 3.2). The predecessors and successors of an event in this partial order, indicate the conditions that triggered and followed each event.

Execution history display is a pictorial view of the concurrent state of E. Animation is simply a display of the progress of execution as E is mapped to M (Section 3.2.4). During animation, the elaborated graph acts as an underlying structure whose nodes and arcs are highlighted in the topological sort order as each event and its orderings are mapped to the nodes and the arcs of the elaborated graph.

1.3.2 Restricting Execution to Selected Actions

The restriction facility records the executions of actions selected for monitoring, and filters out the executions of remaining actions. An action selected for monitoring records its execution and forwards the id of each of its execution event to its successors (Section 3.3). An action whose execution is not to be recorded simply forwards its predecessor list to its successors. The forwarded list eventually reaches a selected action that records it. Execution trace is, thus, restricted to contain only the execution events of selected actions and their orderings. This filtering greatly simplifies the checking of the model of the expected behavior. Note that the restriction described above is done by the instrumentation inserted in the actions at runtime. The unified debugger also provides facilities for post-restricting the event trace generated by the instrumentation. The event trace can be post-restricted to actions selected for display, and actions selected for checking (Section 6.1).

1.3.3 Predicate/Model of the Expected Behavior

A model/predicate of the expected behavior may specify immediate orderings, or transitive orderings between execution events of the selected actions (Section 3.3). It may also specify absence of orderings. A race condition, for example, is a model of expected behavior that specifies absence of orderings between executions of actions whose data accesses may conflict (Section 3.6).

The model/predicate of the expected behavior may be formally or informally specified by the programmer. The checker facility provided by the debugger automatically checks for the formally specified behavior (Section 6.4). The programmer must visually check the (informally specified) behavior against the displays provided by the debugger (Section 6.5). The ability of our approach to restrict the traces to only the selected actions helps both the programmer and the checker facility.

1.3.4 Automatic Checking of the Expected Behavior

Expected behavior specifying immediate orderings can be checked easily as these orderings are available in the execution event records of actions. However, checking of transitive orderings or absence of orderings between execution events is much more involved.

The checker has to collect the predecessor information from the event records of each action to establish transitive orderings. Our checker establishes these orderings with the help of vector clocks (Section 6.4). The size of the clock vector depends upon the number of actions whose relationships are being checked. The checker updates and maintains the vector clock during the topological sorting of the event records.

1.3.5 Display of the Unexpected Behavior

It is not enough for the checker facility of a debugger to come back to the user with a terse statement saying that the expected behavior did not occur. It should additionally be capable of displaying to the user the unexpected behavior that actually happened. Thus, event records need to be sorted and saved during checking as the user may later request their display (Section 6.1). We, therefore, postpone the updating of vector clocks until the topological sorting of the execution event records. This avoids any extra overhead that may be incurred by the actions in maintaining the vector clocks during their execution.

1.3.6 Cyclical Debugging

A programmer often cycles through the debugging steps a number of times before identifying a bug. Execution replay (Section 3.5) allows the user to exactly replay an execution with the exact ordering of events as in the initial execution. The program is first run to record the non-deterministic choices of dependences with which each action executes. It is, then, replayed by forcing the actions to make the choices recorded in the earlier run (Section 5.4).

1.3.7 Support for Interactive Debugging

The use of actions simplifies interactive control over the execution (Chapter 7). The elaborated graph provides mapping between the run-time objects and the symbols defined in the program. These mappings are helpful in controlling the execution of actions, and querying their state at run-time. The debugger provides the usual breakpoint facilities.

Chapter 2. State of the Art and the Related Work

Debugging of parallel and distributed programs has been a subject of much research in recent years. There are over 600 citations contained in the two bibliographies published in 1989 [UtP89] and in 1993 [PaN93]. The work surveyed in these bibliographies describes many different approaches for concurrent debugging and reports on the implementation of many different facilities. Bates and LeBlanc note in [LeM89] that the previous work provides no framework or model for the total process of concurrent debugging. There is no definition of concurrent debugging that says why a particular facility or feature is provided and how it relates to the process of debugging. The result is that most of the facilities or tools which have been proposed or implemented are restricted to some subpart of the debugging process and that the tools are generally incompatible. In this circumstance, the programmer must learn and use many different methods and tools. The programmer typically has to compose his own process for debugging of parallel programs and use different concepts and different facilities for different steps in debugging.

Chapter 1 gave a specification of the process of debugging which involves relationships between the program, P, the expected behavior of the program, M, and the observed actual execution behavior, E. Debugging consists of a series of mappings between these entities. In particular, the mappings which are important are $P \rightarrow M \rightarrow E \rightarrow M \rightarrow P$. This process definition is used as a framework for defining previous work. The analysis focuses on the restriction in concepts that have tended to partition the different subparts of the debugging process making each part a separate problem area. It is clearly impractical to attempt to give a detailed discussion of all of the previous work. Rather, this chapter relates what this approach considers to be the most significant previous work in the context of the debugging process.

2.1 Sequential and Concurrent Debugging

Table 2-1 shows how major concurrent debugging facilities relate to different parts of the debugging process.

Table 2-1. Coverage of various parts of the debugging cycle.

Various Parts of the Debugging Cycle	Sequential Debuggers	Concurrent Debugging Facilities				
		Static Analysis	Execution Replay	Race Detection	Displays/ Animation	Model Checkers
Uses structural information of P	X[a]	X				
Presents to user a representation of E	X				X	
Records causal orderings; P → E	X[b]		X			X
Restricts E to selected events	X					X
Provides mapping to program; E → P	X				X[c]	
Allows user to represent M	X					X
Checks expected behavior; M → E	X	X[d]		X		X
Presents unexpected behavior; E → M	X	X				
Enables cyclical debugging	X[e]		X[f]			
Allows interactive debugging	X		X			

a. In the form of static call graph
b. Available by default in a sequential execution.
c. Typically, animation facilities provide mapping to a process structure, not P.
d. Can check only pre-specified behaviors.
e. Available by default: In the absence of non-determinism in sequential executions.
f. Some replay facilities also support race detection, and use structural information of P.

2.1.1 Sequential Debugging

A source level sequential debugger like *dbx* covers various parts of the cycle (Table 2-1). It represents the program, P, as a sequence of lines in the source code of the program to be debugged.

The user interacts with the debugger with commands that refer to program actions. The actions are source lines of the program text or function/procedure calls. The debugger presents the execution, E, as a sequential trace of events. Each event is an execution occurrence of an action, i.e. a source line or a function/procedure call. As both the execution, E, and the program text, P, are sequences of lines, there is a straightforward mapping between the two. Note that the compiler provides the debugger with mapping of the physical addresses to the lines of the source code.

As it is difficult to debug the entire execution behavior of a program at the same time, the user focuses, in each debugging cycle, on the execution behavior of some small segment of the program. A sequential debugger helps the user in comparing the expected execution behavior of the segment against its actual execution behavior by restricting the execution trace to occurrences of interesting actions. It provides commands like:

`trace <sourceline> [ if <condition>]`, and

`trace <function> [if <condition>]`.

In these commands, the user represents the expected behavior, M, as conditions on the program state. The debugger can easily check such a condition because it is associated with the specified action (source line, or call). The identity of the action allows the debugger to monitor the breakpoint without much overhead. The debugger inserts a trap at the address corresponding to the action. The execution breaks at this point and the condition is tested. However, checking of a condition that can not be pin-pointed to happening at a particular action is much more expensive to monitor. For instance, the command **`trace if <condition>`** asks the debugger to monitor the condition after the execution of every source line. This monitoring is very expensive and such commands are often avoided by the programmer.

The debugger provides interactive support using static and dynamic structure of the program. Static call graph is available in the symbol table information. At runtime, the debugger maintains the current stack of active call frames. Note that each instance of a call (or a block of code) is represented in memory with a frame. This provides the debugger with the dynamic structure of the computation. In addition, the debugger maintains mappings which help in providing breakpoint control of the execution and querying the runtime state of objects. Following mappings are maintained:

1. Mappings between source lines and the addresses of their generated code.

2. Mappings between calls and the addresses of their executable code.

3. Mappings between active blocks (scopes) and their frame addresses.

2.1.2 Concurrent Debugging Facilities

A concurrent execution introduces problems arising from multiple threads of control, synchronizations among these threads of control, probe effect, race conditions and non-determinism [McH89]. This makes concurrent debugging much more difficult than sequential debugging. Therefore, concurrent debugging facilities often tend to develop around one of these problems. See Figure 2-1[1].

Execution history displays help user in keeping track of the numerous synchronizations that have taken place among various threads of execution [LMF90], [FLM89], [PaU89]. The focus of such displays is, thus, limited to E.

Animation facilities help in following the progress of execution “instantaneously” in each thread of control [HoC87], [PaU89], [HoC90], [SBN89], [ZeR91], [Ho91]. The focus of animation is mapping E to P.

[1] Dotted lines show that some approaches may address more than one problem area.

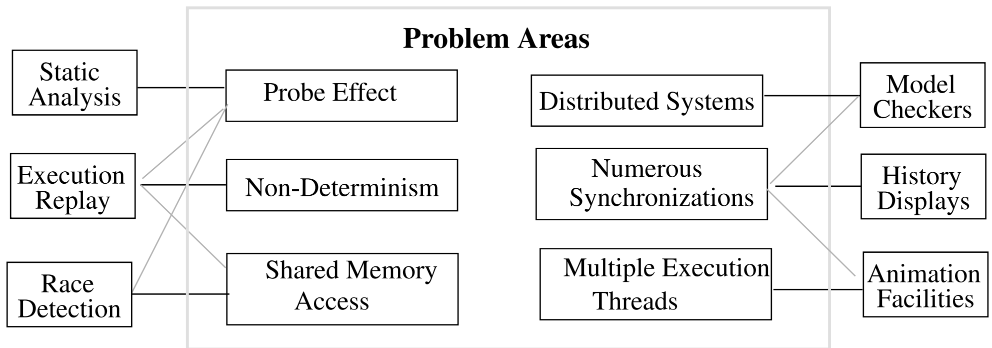


Figure 2-1. Debugging facilities target individual problem areas

Race detection facilities detect simultaneous accesses to shared data (R-W, W-W) [NM90a], [EGP89], [HMW90], [Sch89]. Their focus is *race* behavior in the P $\rightarrow$ E part of the debugging cycle.

Predicate/model checkers automate the process of verifying the expected behavior of events happening at various sites in a distributed system [HHK85], [BFM83], [BH83], [HsK90], [WaG91], [Bat89], [GaW92], [ReSc94]. They allow a user to specify M, but, are limited to M $\rightarrow$ E part of the debugging cycle.

Execution replay facilities overcome the non-determinism of a concurrent execution [LM86], [For89], [MiC89], [Net93]. They make cyclical debugging possible, but often ignore individual parts of a cycle.

Static analysis extracts more information from P when "probe-effect" [Sto88], a principle similar to Hiesenberg's uncertainty principle, limits further instrumentation [Tay83], [TaO80], [BBC88], [CaS89], [McD89]. They analyze P and are limited to the P $\rightarrow$ M part of the debugging cycle.

2.2 Problems in Various Parts of the Cycle

Fig. 2-2 shows the problems and ambiguities often encountered during the mapping between different representations in various parts of the debugging cycle.

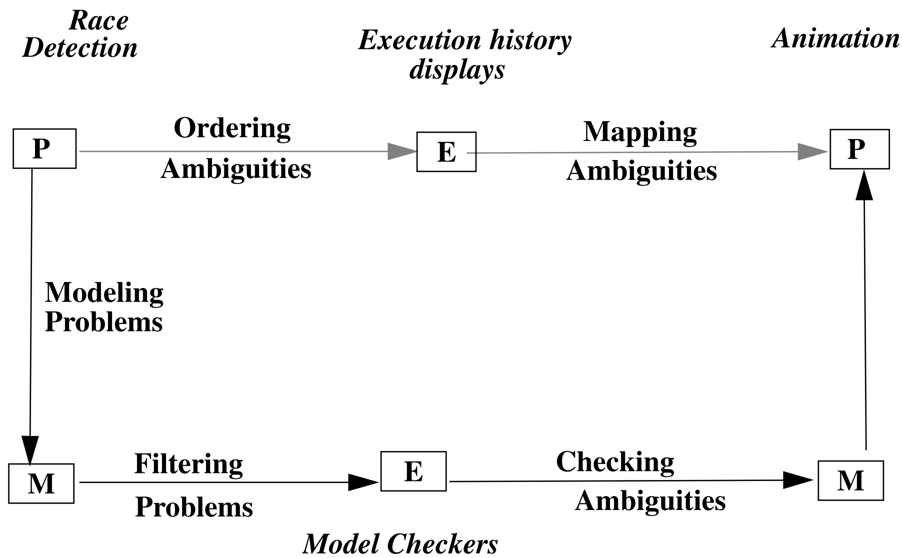


Figure 2-2. Ambiguities in various parts of the cycle

Parallel and distributed execution environments typically provide processes or threads as executable units to run the program text. There is often a complex multiplexing of the program text among instances of the process/thread structure during execution due to resource limitations, scheduler policies, and other constraints. A concurrent debugger must resolve the mapping and timing ambiguities resulting from such multiplexing. A visualization of execution history [PaU89], [McH89] can help in resolving some of these ambiguities by displaying various threads of control and the synchronizations among them. This is typically a time-process graph representation of the execution [LMF90], [FLM89]. Events in such a representation are defined in the context of the process/threads of the execution environment and not as occurrences of program actions. This creates ambiguities in mapping the observed execution behavior to the program (Section 2.2.1). An animation facility can help in resolving some of these ambiguities in the E $\rightarrow$ P part of the cycle. Animation facilities [HoC87], [PaU89], [McH89], [Ho91] provide an instantaneous view of the mapping of events of a time-process graph to a graphical structure. This, however, translates into extra effort for the user who must then develop a graphical structure that can support animation [HoC90], [SBN89], [Ho91], [ZeR91]. Such structures are, however, unable to support the visualization of the abstractions defined by the user. This has forced some of the approaches to abandon the use of visualizations [CFH93].

Execution of a parallel program typically generates so much trace information that it is difficult to debug the entire program at once. Concurrent debuggers provide two types of facilities for dealing with this explosion of event information:

1. Checker facilities that automatically check a formally specified model/predicate of the expected behavior against the actual execution behavior.

2. Visualization facilities that allow the user to visually check the expected behavior against a display of the actual execution behavior.

Visualization facilities typically do not accept a formal specification of M for automatic checking. The user has to visually check an informal representation of M against the display. That is, the visualization facilities does not allow the user to select program segments whose execution behavior would be displayed. Consequently, the displays often contain a lot of irrelevant information, making visual checking tedious. These factors often cast skepticism on the utility of visualization facilities [Mi92].

The use of a textual representation of P in which actions are not determined makes it difficult for a user to formally model the expected behavior in $P \rightarrow M$ part of the cycle. The text does not allow the user to readily represent conditions about the concurrent state that involve event orderings. Hence, predicate/model checkers use execution and *problem* oriented [HsK90] approaches for representing the user expectations (Section 2.2.3). To accomplish this, they may demand extra programming effort from the user [Bat89]. Furthermore, their use of events, instead of actions, for defining M creates additional problems in checking the expected behavior in $M \rightarrow E$ part of the cycle.

These problems can, in turn, constrain the range of expected behaviors that a checker will allow a user to represent in M [Ho91], [WaG91], [ReSc94].

In the P $\rightarrow$ E part of the cycle, debuggers for shared memory and distributed systems approach the problem of recording causal orderings differently. Shared memory debuggers typically do not record the inter-process orderings. They only approximate the orderings from the sequential traces obtained for each process [HMW90], [NM91a], [Sch89], [EGP89]. This restricts their model checking ability to only one type of expected behavior; the *race* behavior resulting from the non-deterministic access of shared data. Thus, they only provide a facility for *race detection*. On the other hand, distributed debuggers like [GaW92] provide more generalized predicate/model checking facilities. They use specialized clocks [Mat89], [Fid89] for time stamping events and recording the causal orderings. This allows them a greater flexibility in checking a variety of predicates/models of expected behaviors. However, this creates the overhead of maintaining these vector clocks [ReSc94].

In M $\rightarrow$ P part of the cycle, the user maps any unexpected behavior in M to the program in order to get closer to the bug. A terse message from the checker stating that the check for expected behavior has failed, is not enough to the user. The checker must help the user by presenting the unexpected behavior that caused the check to fail [Bat89]. However, existing checkers seldom keep enough information during measurement that would allow them to present the unexpected behavior to the user. This forces the user to exert extra effort in repeatedly querying the debugger for the same information.

2.2.1 Mapping Ambiguities

In E $\rightarrow$ P part of the cycle, a debugger has to map the events defined in the context of processes/threads of the execution environment on to the program text. However, ambiguities arise in mapping intra-process arcs of a time-process graph to their corresponding sequential text in the program. For instance, in Fig. 2-3(a) there is an ambiguity about the intra-process arc $\boldsymbol{x}$ of Process 3. $\boldsymbol{x}$ can either map to the sequential text S or to the sequential text T of

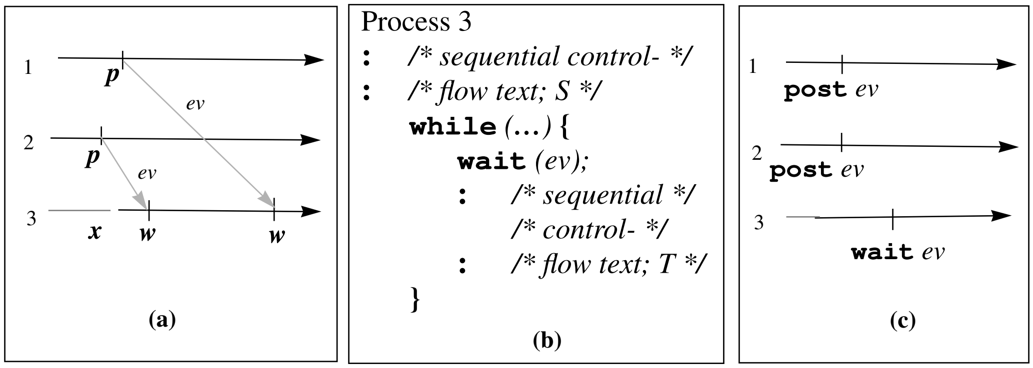


Figure 2-3. (a) Time-process graph (b) Program text (c) Event traces.

Fig. 2-3(b). Event ***w*** that immediately follows ***x***, and maps to the synchronization statement **`wait`** *(ev)* is not of much help. **`wait`** *(ev)* is neither associated with S nor with T. The synchronization event simply sits at the boundary where a piece of text ends and another one starts. Note that an event's association in a time process graph is with a process (#3 in this case), not with an action of the program text.

Instead of letting a synchronization statement sit ambiguously on the border of two sequential text segments, we propose an abstraction that permanently associates the synchronization statements of a program with its sequential text segments. The abstractions resulting from this association is that of a *computation action* as given in Section 1.1.1. The concept of computation action disentangles the sequential control-flow considerations from the synchronization considerations. The programmer is, then, able to concentrate on dependences between actions instead of scheduling entities such as a process.

Animation facilities [McH89], [PaU89] typically require an underlying process structure for supporting their visualizations of the E $\rightarrow$ P mapping. In this structure, nodes are processes and arcs are synchronization/communication links between processes. Current animation facilities often demand extra user effort to develop an alternate structure for supporting the visualizations of the E $\rightarrow$ P mapping. A textual representation of P can not directly support this visualization because it conceals the synchronization dependences to which inter-process arcs of a time-process graph map. These dependences are concealed in the semantics of the synchronization constructs. For e.g., the dependences that order **p** and **w** in Figure 2-3(a), are concealed in the semantics of **`wait`** *(ev)* in Figure 2-3(b) that shows the text of process 3.

We, therefore, use a graphical representation of P [New93] whose nodes are the computation actions and whose arcs are their dependences (Section 3.1). Occurrence instances of computation actions are partially ordered. They provide a "pomset" [Pra86] representation of E that allows us to automatically generate the animation structure (Section 3.2).

2.2.2 Ordering Ambiguities

In the P $\rightarrow$ E part of the cycle, a debugger should record the events and their orderings. Distributed systems often record the event orderings by exploiting the data dependences introduced by the send/receive of messages with the help of unique time-stamps (or identifiers) [Mat89], [Fid89], [WG92]. Shared memory debuggers that detect races [NM91a], [HMW90], however, ignore the data dependences introduced by the accesses to the shared synchronization variables. They may, also, ignore the unique identifiers or time-stamps for each event. Their recorded event traces often contain ambiguities about the inter-process orderings as shown in Figure 2-3(c). There is ambiguity as to whether the **`wait`** *(ev)* of process 3 was fulfilled by the **`post`** *(ev)* of process 1 or 2. This necessitates the use of approximations [HMW90], and leads to intractability [NM90b]. Inability to record the order of accesses to shared objects further complicates the detection of *races* (simultaneous access to shared objects with at least one write), and affects the accuracy of detected races [NM91a], [NM91b].

Note, however, that if an event that write-accesses a shared object, appends its unique identifier to the object, then a later event access to that object can identify its "causal" predecessor (Section 3.2.1). This observation allows us to support execution replay, and to support race detection with little extra overhead (Section 3.4).

2.2.3 Modeling Problems

A sequential *dbx* debugger is user-friendly. It allows the user to associate expected conditions with a program action. For instance, in a *dbx* command such as "**`when at <stmt> if <condition>`**", the user associates a condition about the sequential state of the executing program with a statement of interest. Later, the debugger allows the user to interactively follow the conditional progress of the execution that has been restricted to the interesting actions. Such a *program oriented* approach [HsK90] is not possible with a "textual" representation of a concurrent program. Unlike *dbx* conditions, conditions about the concurrent state involve event orderings whose corresponding dependences are not visible in the textual representation. For instance, event orderings in an expected behavior like "($\boldsymbol{w} \wedge \boldsymbol{p}$) <u>precede</u> ***w***" for Figure 2-3(a), correspond to dependences that are not visible in the textual representation of Figure 2-3(b). Hence, assertion/model checkers [HsK90], [McH89] adopt execution oriented approaches that use models like temporal logic, interleaving, partial order or automatas [Ho91]. In P $\rightarrow$ M part of the cycle, therefore, a user has to exert extra effort to learn a new language, specify the expected behavior and, then, debug it for *user errors* [Bat89] before debugging the original program.

2.2.4 Filtering Ambiguities

In M $\rightarrow$ E part of the cycle, *assignment* and *resolution* problems [Bat89] are typical of the ambiguities that arise during filtering and recognition of the expected behavior. As most existing checkers are execution oriented, they use events in their representation of the expected behavior, and leave the actions implicit. This conceals the information that (i) events are actually multiple occurrences of actions, and (ii) the observed event orderings are the unrolling of the communication/synchronization structure of the program actions. For instance, a behavior like "**p** <u>precedes</u> **w**" gives no information to the debugger about the actions that correspond to events **p** and **w**. Ambiguities can, then, arise whenever more than one observed behavior fits the expected behavior. In Fig. 2-3(a), "**p** precedes **w**" can fit several behaviors; **p** of process 2 precedes the first **w** of process 1, **p** of process 1 precedes the second **w** of process 2, or **p** of process 2 precedes the second **w** of process 3. Such ambiguities restrict the range of checkable behaviors. This, in turn, restricts the range of behaviors that can be represented in M [Ho91], [Bat89].

Information defining actions such as their statement line numbers, could have resolved these ambiguities. We, therefore, use such information about the actions to simplify the filtering and recognition of the expected behavior (Section 3.3).

Chapter 3. The Unified Model of Concurrent Debugging

The unified model presented in this chapter decomposes the problem of debugging a concurrent program into two levels. A programmer debugs the concurrent state at the upper level, where the only important concerns are the relations on the set of computation actions; specified in the program, P, modeled in the expected behavior, M, and observed in the execution behavior, E. Internal states of a computation action are not important at this level because the execution occurrences of the computation action are considered atomic. These states only become important when the programmer moves to the lower level, inside the computation action, to debug its internal sequential text.

3.1 The Specified Behavior

Notation 3-1 Σ_P is the set of computation actions specified in the program.

Data dependences that force the computation actions to execute in a particular order are represented by ordered pairs:

Def. 3-1 The set of *data-flow dependences* is $F_P \subseteq \Sigma_P \times \Sigma_P$.

For instance, the dependence of a `receive` of a message on its `send`, the dependence of a `P` of a semaphore on its `V`, or the dependence of a `wait` of a synchronization event on its `post`, represent such data-flow dependences. The write-read dependence on a shared synchronization variable, or on a message, forces the actions to execute in a particular order. The motivation for representing the synchronization dependences as data-flow dependences comes from the language independence and machine independence goals of the CODE graphical programming environment [BAS90], [NB92]. The data flow characterization of the synchronization and control-flow dependences in CODE allow the environment to support shared memory, as well as, distributed systems.

The set of ordered pairs F_P gives a graphical representation of the program. The nodes of the program graph (Σ_P, F_P) are the set of computation actions, and its arcs are the data-flow dependences. See Figure 3-1(a).

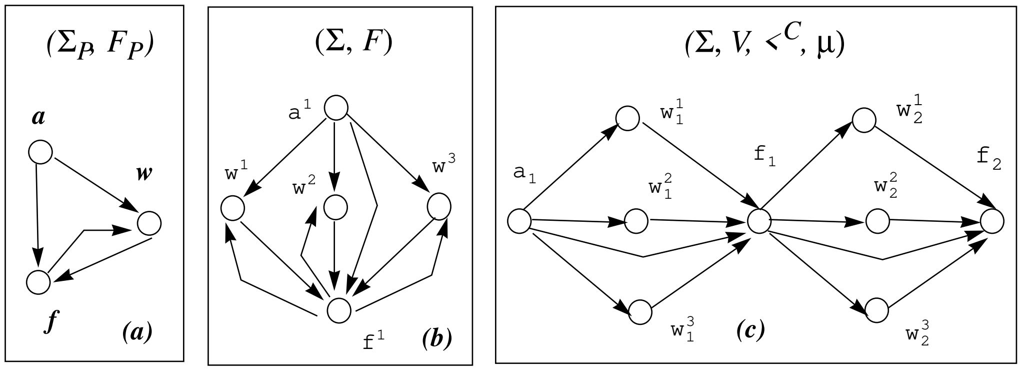


Figure 3-1. (a) Program graph (b) Elaborated graph (c) Pomset execution

3.1.1 The Elaborated Graph

In order to achieve parallelism in a concurrent execution, the same piece of program text is often assigned to multiple executable units (processes or threads) of the execution environment. Such an assignment of actions of Σ_P to the processes/threads of the execution environment is represented by the set of executable computation actions.

Def. 3-2 An *executable* computation action is an instance of $\boldsymbol{u} \in \Sigma_P$ which can execute at runtime.

Thus, computation actions in Σ_P are template types. There can be one or more executable instances of a template computation action at runtime. In the block-triangular solver example explained in Section 1.2.1, there were N-1 executable instances of the **Mult** node (template) at runtime.

Notation 3-2 The set of executable computation actions is denoted by Σ.

Let $\Sigma \subseteq \Sigma_P \times N$. Then, u^i, $u^j \in \Sigma$ indicate the instantiations of $\boldsymbol{u} \in \Sigma_P$ with indices $\boldsymbol{i}$ and $\boldsymbol{j}$. The actual identity of an executable unit to which each instance is assigned is not important. The superscripts $\boldsymbol{i}, \boldsymbol{j}$ are only logical ids for distinguishing between the multiple instances.

The elaborated graph (Σ, F) is the runtime structure resulting from the instantiations of the template computation actions of the program graph (Σ_P, F_P). In this structure, the runtime replication of templates of Σ_P is represented by the set of executable actions, Σ. The corresponding replication of arcs of F_P is represented by the data flow dependences, $F \equiv \{(u^i, v^j) \mid (u,v) \in F_P \text{ and } u^i, v^j \in \Sigma\}$. It is called elaborated graph because it shows the "elaboration" (replication) of nodes and arcs of the program graph. Figure 3-1(b) shows such an elaborated graph where action $\boldsymbol{w}$ of Figure 3-1(a) replicates (or elaborates) into three actions; $\boldsymbol{w}^1$, $\boldsymbol{w}^2$, and $\boldsymbol{w}^3$.

In the following sections, the term computation action (or simply action) will be used for a member of the set of "executable" computation actions, Σ. The prefix "template" will be explicitly used to refer to the template computation actions of the set Σ_P. To simplify the notation, the superscripts (indices) of executable instances will not be indicated unless multiple instantiations of a template action are being considered.

3.1.2 Firing and Routing Rules

Intuitively, a computation action acts like a procedure whose input parameters are the input dependences and output parameters are the output dependences. It begins execution by obtaining a set of values from its input dependences. Then, it performs a sequential computation on this data. It ends its execution by putting a set of values on its output dependences.

Input dependences of an action $\boldsymbol{a}$ are given by the incoming arcs; $in(a) \equiv \{(x, a) \mid (x, a) \in F\}$. And, output dependences are given by the out-going arcs; $out(a) \equiv \{(a, x) \mid (a, x) \in F\}$. Conditions specified on the input dependences determine when to initiate the execution of a computation action, and conditions specified on the output dependences determine what follows its execution. The pre-condition that initiates the execution of a computation action is called an *input firing-rule*, and the post condition that follows its execution is called the *output routing-rule* [New93].

Def. 3-3 An *input firing rule* $I(a)$ is a set of subsets of input dependences, i.e. $I(a) \subseteq 2^{in(a)}$. An *output routing rule* $O(a)$ is a set of subsets of output dependences; i.e. $O(a) \subseteq 2^{out(a)}$.

An input firing rule $I_P(a)$ is a condition in the disjunctive normal form (sum of products). Each element of $I_P(a)$ represents a disjunct, and is given by a subset of $in(a)$, input dependences of $\boldsymbol{a}$. The state of a data-flow dependence (x, a) can be represented by a string of values denoted by [x, a]. A computation action is ready for execution if the state of all the dependences of an element $\iota \in I(a)$ are non-empty strings i.e. $\forall\ (x, a) \in \iota :: [x,a] \neq \varepsilon$. Then, input to a computation $\boldsymbol{a}$ is a set of suffix values detached from the state of dependences in ι. On completing its computation, $\boldsymbol{a}$ will catenate a set of output values as prefixes to the state of all the dependences given in some element $o \in O(a)$.

3.2 The Observed Behavior

The execution occurrences of computation actions and their orderings are traced in $P \rightarrow E$ part of the debugging cycle. Figure 3-1(c) shows this information.

Def. 3-4 *A computation event* is an execution occurrence of some *computation action*.

Notation 3-3 The set of computation events is denoted by V.

An action can occur multiple number of times. Subscripts in Figure 3-1(c) denote the multiple occurrences of actions. Set V of events is, thus, a set of multiple occurrences of actions, or a "multiset" of occurrences of actions. The function μ: $V \rightarrow \Sigma$ maps each event of V to that action of Σ, of which it is an occurrence.

We support this mapping by associating an execution counter $u.i$ with each *action* $u \in \Sigma$. The id-count pair; u and $u.i$, provide a unique identifier for each event. We, therefore, denote the i-th occurrence of actions u, v, w ... as events u_i, v_i, w_i, ...

3.2.1 Causality of Data Flow Dependences

In order to record the orderings enforced by F, the debugger appends the unique identifier with the data shared through the data-flow dependences. Whenever an action u puts some data on its output dependence $(u, v) \in F$ in its

i-th occurrence, it appends the identifier u_i to the data. Similarly, whenever an action u begins its i-th execution by removing some data from its input dependence $(v, u) \in F$, it detaches the identifier appended to the data, and puts the detached identifier in a predecessor list denoted by $u.i.P_F$. The list contains such pairs for all the predecessor events that have "caused" the i-th execution occurrence of u. The traces contain records of the execution occurrences of each event. The trace record of an event u_i contains the action id u, execution count $u.i$, and the predecessor list $u.i.P_F$. Also, see Table 1.

Def. 3-5 The orderings enforced by F are $<^F \equiv \{(u_i, v_j) \mid u_i, v_j \in V \wedge u_i \in v_j.P_F\}$.

The transitive closure of $<^F$ results in an irreflexive partial ordering that constitutes the *causal orderings* $<^C$. The orderings $<^F$ simply reflect the "causality" of data-flow dependences.

Lm. 3-4 $u_i <^F v_j \Rightarrow (u, v) \in F$.

Consequently, if $u_i <^F v_j$, then there is some data shared between u_i and v_j, namely, the state of the data-flow dependence $(u, v) \in F$. The representation of the state of a data-flow dependence in Section 3.1 by a string of values (or an infinite FIFO buffer) takes into account this dependence. Moreover, it allows us to model the general cases of the send and receive of messages in distributed systems, and the data-flow dependences of the graphical/visual languages like CODE [BAS90],[NB92]. But, the representation may create problems in modeling the synchronization primitives of shared memory systems. However, as seen in Section 3.2.3, concurrent state does not depend upon the internal representation of the states of the data-flow dependences. It only depends upon the event orderings. Thus, we can represent the state of a synchronization dependence, with a string (or a buffer) of length one. It will denote the data dependence due to the shared synchronization variable (whose only permitted values are set or reset). The debugger records the causal orderings by appending and detaching the identifier to the shared synchronization variable.

3.2.2 Execution History Pomsets

As seen above, actions can occur multiple number of times as events, i.e. the set V of events is related to the set Σ of actions through the function μ: $V \rightarrow \Sigma$. This effectively turns the poset $(V, <^C)$ into a pomset $(\Sigma, V, <^C, \mu)$ [Gai88], [Gis88], [Pra86]. A POMSET is a Partially Ordered MultiSET of occurrences of **actions**, in much the same way as a **string** is a TOMSET; a Totally Ordered MultiSET of occurrences of **alphabets**. The pomset $(\Sigma, V, <^C, \mu)$ is called a causal pomset because $<^C$ are the causal orderings. It is instrumental in unifying our model because its expression of the concurrency properties is independent of the way time or events are modeled in a system [Gai88].

A pictorial representation of the causal pomset is an *execution history display.* We can now explain why execution history displays are so helpful in debugging. They display the causal orderings of events. These orderings allow a programmer to determine the conditions that initiated and followed each execution occurrence of an action. From Lemma 3-4, an immediate predecessor of u_i must map to an input dependence of u; and from Def. 3-3, an element of the input firing rule of u is a subset of input dependences. Hence, immediate predecessors of an event u_i inform the programmer about that element of the input firing-rule that initiated the i-th occurrence of u. Similarly, immediate successors of an event u_i inform the programmer about that element of the output routing-rule that determined the condition following the i-th execution occurrence of u.

Notation 3-5 $\quad {}^\bullet u_i \equiv \{(v, u) \mid v_j <^F u_i\}$ and $u_i^\bullet \equiv \{(u, v) \mid u_i <^F v_j\}$

Def. 3-6 $\quad$ A causal pomset $(\Sigma, V, <^c, \mu)$ is *compatible* with the firing and routing rules iff $\forall u_i \in V :: {}^\bullet u_i \in \mathrm{I}(u) \wedge u_i^\bullet \in \mathrm{O}(u)$.

This shows the compatibility of the immediate orderings of a given event with the rules specified on the immediate dependences of its corresponding action. In Section 3.3.1, we extend this compatibility of immediate orderings with the immediate dependences to the compatibility of transitive orderings with the transitive dependences. This provides a framework for representing and checking the expected behavior.

3.2.3 Concurrent Execution State

There is non-determinism associated with the choices of the elements given in the input firing rules. An action can non-deterministically select different elements of a firing rule. In Figure 2-3, the second occurrence of action ***w*** can non-deterministically select any element from its *input firing rule*. During execution replay, a record of the causal orderings informs our debugger to select the right element of the firing rules for each execution occurrence of an action. Thus, it reconstructs the states of the previous execution. Using $<^c$ and following [Mat89], we find a notion of concurrent state:

Def. 3-7 A *concurrent state* of an execution $(\Sigma, V, <^C, \mu)$ is a consistent cut-set of the poset $(V, <^C)$. A set $C \subseteq V$ is a consistent cut-set iff $e \in C \wedge e' <^C e \Rightarrow e' \in C$.

This definition is independent of the local state of an action. It is, also, independent of the contents of the messages exchanged by the actions. It only depends upon the order of events. Hence, distributed systems often reduce their roll-back and recovery overhead by only recording the event orderings, and not the content of messages or the checkpoints of the local states [JhZw90]. Therefore, our execution replay facility exploits this definition to reduce the recording overhead (Section 3.5).

3.2.4 Animation

Animation provides an instantaneous view of the progress of execution. It is simply the process of displaying Def. 3-6 on the elaborated graph while traversing the execution pomset.

During animation, the debugger traverses the execution pomset. On encountering the ***i***-th event occurrence of an action ***u***, it highlights in the elaborated graph those input dependences of the action that correspond to the immediate predecessors of its ***i***-th event occurrence. It, then, highlights the action ***u***. Then, it highlights those output dependences of the action that correspond to the immediate successors of its ***i***-th event occurrence.

We can automatically generate an elaborated graph from the execution pomset. Note that the structure shown in Figure 3-1(b) is obtained by folding all the subsequent occurrences of actions in Figure 3-1(c) to their first occurrences. This fulfills the requirement of a strong coupling between animation and execution history [McH89].

3.3 The Expected Behavior

In the P $\rightarrow$ M part of the cycle, a user represents the expected behavior by selecting some "interesting" actions Σ_M from Σ and, then, representing the expectations as conditions I_M, O_M on their dependences F_M.

3.3.1 Representing Expected Behavior

A *dbx* debugger is closely coupled with the program because it compels the programmer to use only those objects that already exist in the program; e.g. it would not allow a user to specify a non-existent print variable. Taking cue from *dbx*, we closely couple our checker with the program and only allow the user to work with those objects that already exist in the program. This is in contrast to the existing checkers that can not verify if a user has supplied a non-existent order of events in the expected behavior.

Note that if a user expects that events u_i and v_j will be ordered in the execution, then their actions u and v, must exhibit a (transitive) data-flow dependence in the program. That is, $u_i <^C v_j \Rightarrow (u, v) \in F^*$, where F^* is a transitive closure of F. This also follows from Lemma 3-4. For instance, $m_1 <^C c_1$ in Figure 3-2(b) corresponds to the transitive data-flow dependence between m and c of Figure 3-2(a); both are shown by dotted lines. Thus, any pattern that is expected in an execution, must be the unrolling of a pattern already present in the program structure.

Thus, a user starts specifying expected behavior by selecting a subset Σ_M of interesting actions from the program; $\Sigma_M \subseteq \Sigma$. Figure 3-2(a) shows a se-

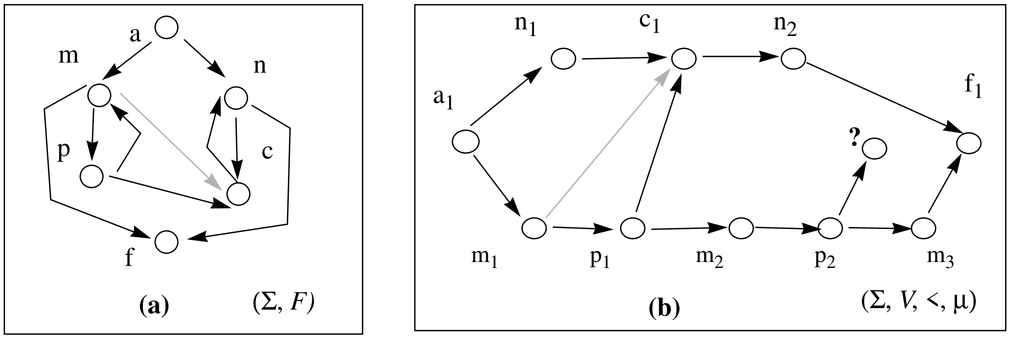


Figure 3-2. (a) Elaborated Graph. (b) Execution.

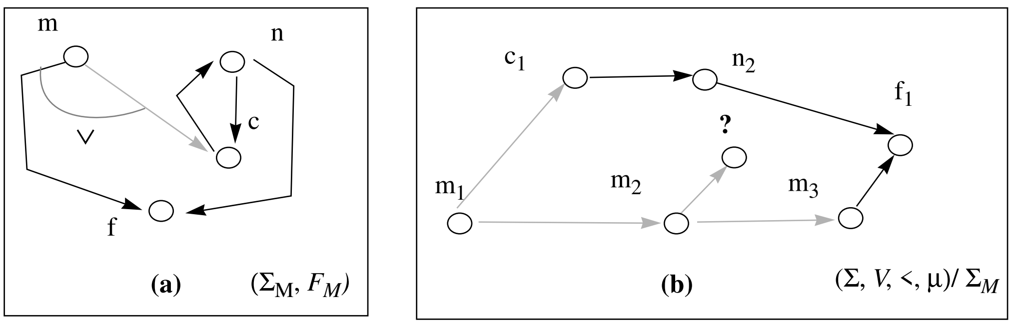


Figure 3-3. (a) Expected Behavior. (b) Restricted execution.

lection of such actions. The user can then specify a dependence between the selected actions only if it corresponds to some dependence of F^*. Some of such selected dependences are shown as F_M in Figure 3-2(a).

Def. 3-8 F_M are the selected dependences from the transitive *data-flow dependences* F^* restricted to interesting actions i.e. $F_M \subseteq F^* / \Sigma_M$, where $\Sigma_M \subseteq \Sigma$.

An observed ordering like $(m_2, ?)$ in Figure 3-2(b), that can not be mapped to a data-flow dependence from F_M is, therefore, symptomatic of a bug!

In Figure 3-2(a), the structure built on Σ_M is the specification of the expected behavior. Akin to the conditions in a *dbx* command like "`when at ... if` <u>*condition*</u>," a user can, then, provide firing rule I_M and routing rule O_M to further restrict the occurrences of "interesting" actions. Note that the rules are conditions about the concurrent state, whereas *dbx* conditions are about the sequential state. Suppose the user specifies an "$\vee$" *output routing rule* for action $\boldsymbol{m}$ in Figure 3-2(a). Then, the checker can filter out occurrences $\boldsymbol{m_1}$ and $\boldsymbol{m_3}$ because they subscribe to the "$\vee$" rule. But, will raise an exception for $\boldsymbol{m_2}$ as it does not subscribe to the "$\vee$" rule.

3.3.2 Recognizing the Expected Behavior

In *dbx*, a directive like "`when at <stmt> ....`" informs the debugger to make necessary preparations for the specified statement. It also informs it to ignore the rest of statements. Similarly, the selection of Σ_M from Σ. informs the debugger to specially prepare for "interesting" actions Σ_M, and to safely ignore the "uninteresting" actions Σ - Σ_M. Then, the debugger can filter out the uninteresting events and can restrict the execution to occurrences of Σ_M.

The restricted pomset $(\Sigma, V, <^C, \mu)/\ \Sigma_M$ of Figure 3-2(b) only contains the interesting events and their mutual orderings. The debugger establishes the orderings among interesting events by exploiting the fact that occurrences of interesting actions can only be ordered if there exists a mutual (transitive) dependence. In Figure 3-2(b), events $\boldsymbol{m_1}$ and $\boldsymbol{c_1}$ are only ordered because of the transitive dependence that exists between actions $\boldsymbol{m}$ and $\boldsymbol{c}$. Note in Figure 3-2(a) that the dependence goes through an intervening uninteresting action $\boldsymbol{p}$. Our debugger, therefore, establishes the orderings between occurrences of $\boldsymbol{m}$ and $\boldsymbol{c}$, by asking the uninteresting action $\boldsymbol{p}$ to relay the causality information that arrived from its predecessors, forward to its successors.

Unlike interesting actions, uninteresting actions do not trace their execution occurrences. Instead of sending the identifier of their current occurrence to their successors, they simply relay forward their predecessor lists. See Table 1. These lists keep getting relayed forward by the intervening uninteresting events until they land in the predecessor lists of the interesting events. Only then they are traced. Predecessor lists of interesting events, therefore, only contain the identifiers of their causally preceding interesting events. Execution is thus filtered to $(\Sigma, V, <^C, \mu) / \Sigma_M$.

Table 3-1. Recording and restricting instrumentation[a]

Monitored Occurrences	Interesting Actions $u \in \Sigma_M$	Uninteresting Actions $u \in \Sigma - \Sigma_M$
u **sends** to *v*	**append**(*msg: u, u.i*);	**append**(*msg:* $u.i.P_F$);
u **receives** *msg*	$u.i.P_F \cup_{\mathbf{e}}$ **detach**(*msg*);	$u.i.P_F \cup_{\mathbf{e}}$ **detach**(*msg*);
u **executed**	**trace** $(u, u.i, u.i.P_F)$; $u.(i+1).P_F = \{\}$; u.i = u.i +1;	$u.(i+1).P_F = u.i.P_F;$ $u.i = u.i + 1;$

a. In the set union $\cup_e$, if there are two events of the same action, then only the most recent event of the two is a member of the set union.

The structural information of M and the fact that the causal pomset is restricted to interesting actions, greatly simplifies recognition of the expected behavior. The debugger traverses the partial order, and tries to check if Lemma 3-4 and Def. 3-6 also hold for M. It checks whether each immediate successor v_j of a given event u_i corresponds to some successor v of the action u in the dependences F_M. Additionally, the debugger checks whether the immediate predecessors and successors of an event u_i satisfy the input firing and output routing rules I_M and O_M for u. If an ordering $u_i <^F v_j$ fails to correspond to some dependence $(u, v) \in F_M$, or the immediate predecessors ${}^\bullet u_i$ or successors $u_i^\bullet$ fail to meet the expected conditions $I_M(u)$ or $O_M(u)$, then an error has been recognized. This happens for the unexpected orderings of m_2 in Figure 3-2(b).

Def. 3-9 Event u_i is in *error* if $u_i^\bullet \notin O_M(u) \vee {}^\bullet u_i \notin \mathrm{I}_M(u)$.

Thus, concurrent debugging is the process of following the unexpected orderings given by the erroneous events, in the direction of causality.

3.4 Shared Data Dependences

Unlike data-flow dependences F that force an ordering on the execution of actions, shared data dependences do not impose any particular orderings. A shared data object is therefore modeled by the set of actions that share the object. The set of those objects is denoted by S.

Def. 3-10 The set of *shared data dependences* is $S \subseteq 2^{\Sigma}$. A *shared data dependence* is a set D of *computation actions*, $D \subseteq \Sigma$ (or $D \in S$).

The actions that participate in a shared data dependence D are classified into disjoint sets of *readers* and *writers*. *Readers* of D are denoted by R_D and *writers* are denoted by W_D; where $R_D \cup W_D \equiv D$, $R_D \cap W_D \equiv \phi$. In Fig. 3-3(a), shared data dependence $D = \{a, r, w\}$. For example, we may have the set of *readers*, $R_D = \{r\}$ and set of writers, $W_D = \{a, w\}$.

The order of accesses to shared objects is recorded with the same mechanism that was used in recording the $<^F$orderings (Section 3.2.1). This, however, requires a protocol for ensuring a valid serialization on the accesses to shared objects like the CREW (concurrent read exclusive write) protocol [LM86]. Note that members u and v of a shared data dependence have a valid serialization if for every occurrence u_i of a write-access and every occurrence v_j of another access, either u_i occurs before v_j, or v_j occurs before u_i. The protocol, therefore, disallows simultaneous write-access to a shared object with other accesses. The unified debugger ensures a deterministic replay by implementing the protocol as in [LM86]. Without the implementation, simultaneous accesses to a shared object can hinder a deterministic replay by corrupting the object and giving unpredictable results. Moreover, as explained later, the protocol is also helpful in detecting race conditions.

Unlike [LM86] that uses versions of shared objects to record the orderings, we use a simpler mechanism for recording the order $<^D$ of accesses to a shared object $D \in S$. Each shared object has the identifier of the last occurrence of its write-access appended to it. Whenever an action accesses a shared object, it reads the identifier appended to the object, and places the identifier in

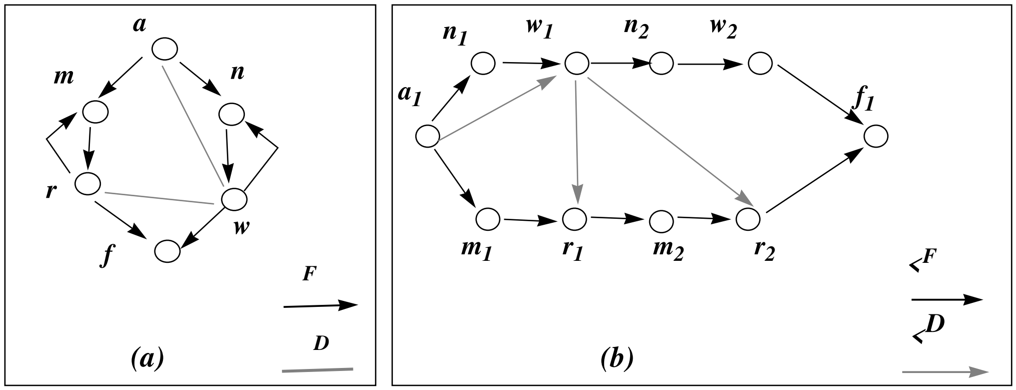


Figure 3-3. (a) Dependences. (b) Orderings.

the predecessor slot reserved for that object in its trace event record. A writer, in addition to the above, replaces the identifier appended to the object with the identifier of its current event occurrence.

In addition to the predecessor list $u_i.P_F$ for the data-flow dependences (Section 3.2.1), the trace record for each event now requires another predecessor list $u_i.P_S$ for shared data dependences. The list has a slot $u_i.P_S[D]$ for each shared data dependence D in which a given action participates. Then, the identifier in the slot for D in the predecessor list of an event record, determines the causal orderings $<^D$.

Def. 3-11 Causal order of accesses to $D \in S$ is $<^D \equiv \{(v_j, u_i) \mid u_i, v_j \in V \wedge u_i.P_S[D] = v_j\}$

There is, thus, an ordering relation $<^D$ for each $D \in S$ generated like $<^F$. These relations augment $<^F$ in the irreflexive partial ordering $<^C$. The ordering $<^C$ is, thus, a transitive closure of the immediate orderings $<^F \bigcup_{D \in S} <^D$.

Note that shared predecessor ordering, $<^D$, explicitly indicates the "causal" precedence of a write event to a given read/write event. However, the "temporal" precedence [Gai88] of a read event to a given write event is indicated implicitly. For example in Figure 3-4, orderings $<^D$ (shown by solid arcs) explicitly indicate the causal precedence of the write event v_j to the read event r_k, and to the write event u_i. However, the precedence of the read event

r_k to the write event u_i (shown by the dotted arc) is implied by the assurance of serialization protocol. The protocol ensures that if a read event and a write event have the same shared predecessor, then the read event must have preceded the write event. That is, r_k precedes u_i if $r_k.P_S[D] = u_i.P_S[D] \wedge r \in R_D \wedge u \in W_D$. As explained below, the precedence of read-events to a given write event is made more explicit by recording the number of readers along with the id of the write event.

$u, v \in W_D$

$r \in R_D$

$\longrightarrow$ $v_j <^D u_i$, $v_j <^D r_k$

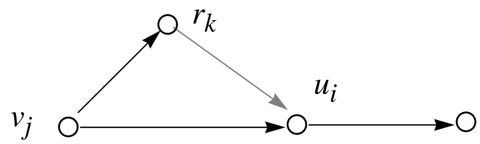


Figure 3-4. Precedence of a write event to a read and a write event.

3.5 Execution Replay

The goal of an execution replay facility is to record enough information about the non-deterministic choices made by the events of an execution so that during replay, the events can be forced to make the previous choices. Thus, replay facility works in two phases: A recording phase in which the execution is run to record ordering information. Then, a replay phase in which the information recorded earlier is used to deterministically replay the execution. In order for the replay to work, it is assumed that the environment provides the same input to the program during both the phases [LM86], [McH89], [MiC89].

There are two types of non-deterministic choices in a concurrent execution: Choices associated with the data-flow dependences and choices associated with the shared data dependences. The unified model separately deals with the choices associated with each type of dependence.

A record of causal orderings $<^F$ is sufficient to overcome the non-determinism associated with the choices of elements in a firing rule. During the checking of the input firing rules, the replay mechanism not only checks if the state of an input dependence is not empty (Section 3.1.2), but also ensures that the event id appended to the value obtained from the dependence is from the same event that fired the earlier execution. Thus, during the $\boldsymbol{i}$-th execution of $\boldsymbol{u}$, a firing rule $\iota \in I(u)$ is only satisfied if $\forall\ (x, u) \in \iota :: [x,u] \neq \varepsilon \wedge appended\_id \in u_i.\boldsymbol{P_F}$, where *appended_id* is the id appended to the value on the given dependence.

The other source of non-determinism is associated with shared data dependences. These dependences do not force the actions to execute in any particular order. The debugger, therefore, records the non-deterministic order of accesses to shared objects so that during the replay phase, the accesses can be forced to occur in the previously recorded order. The replay mechanism ensures that (i) the write access that preceded a given access during the recording phase must also precede during the replay phase, and (ii) all the readers that preceded a given write event in the recording phase must also precede during the replay phase. For this purpose, the recording instrumentation maintains a count of readers in addition to the id of the last write event with each shared object. The debugging information appended to each shared object $D \in S$ is, then, a tuple $(\omega.D, \eta.D)$:

Notation 3-2 $\omega.D$ is the id of the computation event that last wrote D.

Notation 3-3 $\eta.D$ is the current count of the computation events that have read D after it was written by $\omega.D$.

In the recording phase, the replay instrumentation performs the actions shown in Table 3-2 during each execution of a computation action $u \in D$. If $\boldsymbol{u}$ is a reader, the instrumentation saves the id of the last write event in $\boldsymbol{u_i}.P_S[D]$, and increments the reader count, $\eta.D$. If $\boldsymbol{u}$ is a writer, then the instrumentation saves the id of the last write event as well as the current reader count. It initializes the appended reader count to zero, and replaces the id appended to the object with its own id, namely, $\boldsymbol{u_i}$. After $\boldsymbol{u}$ completes its $\boldsymbol{i}$-th execution, the shared predecessor information contained in $\boldsymbol{u_i}.P_S$ is traced in the event record along with the flow predecessor information (Table 3-1).

Table 3-2. Replay instrumentation for D in S.

$u \in D$	$\boldsymbol{u_i}$ obtains access to D	$\boldsymbol{u_i}$ releases access to D
	Recording Phase	
$u \in R_D$	$\mathbf{u_i}.P_S[D]$ **:=** ω.D**;**	$\eta.D := \eta.D + 1$;
$u \in W_D$	$\mathbf{u_i}.P_S[D]$ **:=** (ω.D**,** η.D)	$\eta.D := 0$; $\omega.D := \boldsymbol{u_i}$;
	Replay Phase	
$u \in R_D$	if $\mathbf{u_i}.P_S[D]$ **=** ω.D**;**	$\eta.D := \eta.D + 1$;
$u \in W_D$	if $\mathbf{u_i}.P_S[D]$ **=** (ω.D**,** η.D)	$\eta.D := 0$; $\omega.D := \boldsymbol{u_i}$;

Note that the shared-predecessor information for a reader is simply the id of the event that last wrote the object, whereas the shared-predecessor information for a writer is a tuple containing the id of the previous writer, and the count of the readers. The model assumes that a computation action obtains access to a shared object before the start of its sequential computation and releases the access at the end of the sequential computation. It is also assumed that there is a dummy tuple $(0_0, 0)$ initially appended to each object. The dummy event id 0_0 helps in ensuring the replay of the initial accesses to a shared object.

During the replay phase, the instrumentation ensures that if $u \in R_D$, then $\boldsymbol{u}$ is not granted a read access to D in its $\boldsymbol{i}$-th execution unless the id appended to D is the same as the one recorded in $\boldsymbol{u_i}.P_S[D]$. That is, the instrumentation checks if $\boldsymbol{u_i}.P_S[D] = \omega.D$. If, however, $u \in W_D$, then the instrumentation additionally ensures that the current reader count is the same as the one recorded earlier. That is, the instrumentation does not grant access until the appended id is the same as the recorded id, and the appended reader count is the same as the recorded reader count contained in $\boldsymbol{u_i}.P_S[D]$.

3.6 Race Detection

Simultaneous accesses (with at least one write) to a shared object can race with each other and can corrupt the shared data with unpredictable results. Races are detected by identifying those pairs of events whose accesses to a shared object included at least one write and whose accesses would have been unordered in the absence of the debugger's enforcement of the serialization protocol. For e.g., dotted arcs of Fig. 3-3(b) show the $<^D$ orderings for events $\boldsymbol{w_1}$ and $\boldsymbol{r_1}$ (and $\boldsymbol{w_1}$ and $\boldsymbol{r_2}$) that were forced by the debugger's serialization protocol. The events are otherwise unordered under $<^F$. Without the debugger's protocol, these $<^D$ orderings may not exist, thereby, causing a data-race. Then, $\boldsymbol{w_1}$ and $\boldsymbol{r_1}$ (or $\boldsymbol{w_1}, \boldsymbol{r_2}$) can execute simultaneously with unpredictable results. Thus, all $<^D$ orderings observed under the serialization protocol, should be supported by $<^F$ (i.e. should be causally ordered) as, for instance, the ordering between $\boldsymbol{a_1}$ and $\boldsymbol{w_1}$.

Notation 3-4 $\rho.D$ is the set of computation events that have read D after it was written by $\omega.D$.

Table 3-2 describes the instrumentation needed for race detection. Note that the instrumentation for race detection maintains the set of current readers, $\rho.D$, whereas the replay instrumentation described in Table 3-2 simply maintained the count of readers, $\eta.D$. The orderings are, however, recorded similarly: A reader only saves the id of the previous write event in $\boldsymbol{u_i}.P_S[D]$, whereas a writer saves the count of readers, $|\rho.D|$ in addition to the id of the last write event.

Table 3-3. Instrumentation for data-race detection.

$u \in D$	$\boldsymbol{u_i}$ obtains access to D.		$\boldsymbol{u_i}$ releases access to D
	Race is detected if:	Recording of Orderings	
$u \in R_D$	$\omega.D \not<^C \boldsymbol{u_i}$	$\boldsymbol{u_i}.P_S[D] := \omega.D;$	$\rho.D := \rho.D \cup \{u_i\}$
$u \in W_D$	$\omega.D \not<^C \boldsymbol{u_i} \vee$ $v_j \not<^C \boldsymbol{u_i} \wedge v_j \in \rho.D$	$\boldsymbol{u_i}.P_S[D] := (\omega.D, \lvert\rho.D\rvert)$	$\rho.D := \{\}$; $\omega.D := \boldsymbol{u_i}$;

Whenever, a computation action $\boldsymbol{u}$ obtains a read or a write access in its $\boldsymbol{i}$-th execution, the instrumentation signals a race if the last writer of the object

($\omega.D$) is not a causal predecessor of $\boldsymbol{u_i}$. Recall that $<^C$ is a partial order resulting from the transitive closure of the immediate orderings $<^F \bigcup_{D \in S} <^D$. If $\boldsymbol{u}$ is obtaining a write access, then instrumentation additionally checks the orderings with the current readers of the object (members of $\rho.D$). A race is also signalled if a current reader of the object is not a causal predecessor of $\boldsymbol{u_i}$.

In Figure 3-3(b), race for events r_1 and w_1 (and for events r_2 and w_1) will be detected by checking the condition $\omega.D \not<^{\boldsymbol{C}} \boldsymbol{u_i}$. The race between events w_2 and r_2 will be detected by the condition $\boldsymbol{v_j} \not<^{\boldsymbol{C}} \boldsymbol{u_i} \wedge \boldsymbol{v_j} \in \rho.D$.

Implementing a serialization protocol, and recording the $<^D$ orderings of accesses to a shared object, may seem unnecessary for detecting races. It may appear simpler to report races for pairs of events that are unordered under $<^F$. However, as explained in [NM90a], this can result in reports of spurious races that are infeasible and could never occur. Our debugger's implementation of the serialization protocol is instrumental in eliminating the spurious artifacts that can result from the use of shared objects that were corrupted by an earlier race. Furthermore, the record of $<^D$ helps in improving the accuracy of detected races by identifying other spurious races. Note that race detection in our model does not require any extra overhead. The record of $<^D$ and $<^F$ already exists for supporting execution replay.

Chapter 4.   Implementation Concepts

The unified model of concurrent debugging as described in Chapter 3 has been implemented in the CODE 2.0 visual/graphical parallel programming environment [NB92], [New93]. This chapter describes the capabilities provided by this implementation of the debugger and the extensions and enhancements of the CODE 2.0 programming system which are necessary to support implementation of the unified model of concurrent debugging. It also describes the flow of information among the conceptual elements of the debugging system and the instrumentation of a CODE 2.0 program. An important element of this implementation of the unified model of debugging is that the user provides the implementation with some simple declarative annotations to the parallel program in the CODE 2 environment. The instrumentation and the presentations to the programmer are also presented in the context of the hierarchical dynamic graph structure of the CODE program.

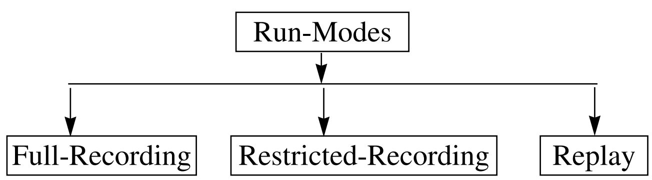


Figure 4-1. Debugging run-modes.

4.1   Overview

Debugging instrumentation added to the actions (Chapter 5) allows the program to be run in one of the modes shown in Figure 4-1[1]. In each of these modes, interactive support is available along with other debugging facilities. In the full-recording mode, the instrumentation generates execution event records for all executions of every program action. In the restricted-recording mode, recording is restricted to only the selected actions. In the replay mode,

[1] There is also a performance run mode in which debugging instrumentation generates a logical trace of the execution for performance measuring purpose. In addition, there is a debugging-off mode in which all the debugging instrumentation can be turned off. No debugging facilities are available in these modes.

the instrumentation uses the event records from a fully recorded trace of an earlier execution to exactly replay that execution.

The instrumentation can save the event records of the execution of actions in the internal structures of the debugger for a later use. It can also send the event information to any of the facilities requested by the user for immediate analysis and presentation. See Figure 4-2. The facilities provide filing, displaying, checking and/or post-restriction of the event information (Chapter 6). As shown in the figure, events traced in a file or recorded in internal structures are available for postmortem analysis and presentation. Note that the events of a fully recorded trace can also provide a replay of the recorded execution.

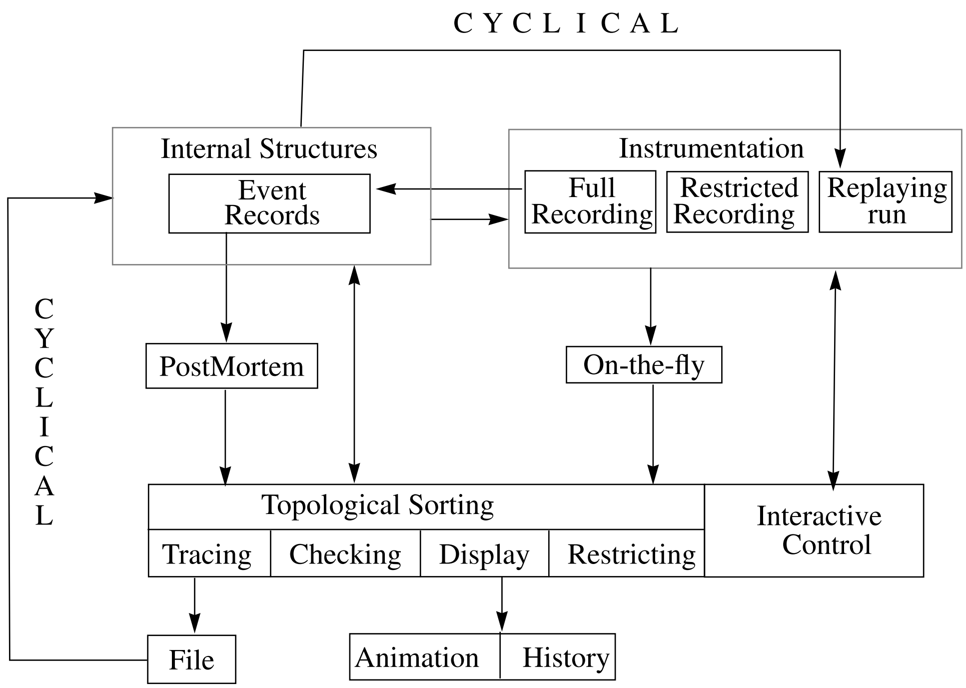


Figure 4-2. Available facilities.

Debugging facilities are available on-the-fly during the execution of actions or in a postmortem fashion after the end of the execution. Events arriving on-the-fly or in a postmortem fashion are topologically sorted before they are utilized by any of the facilities. The sorting delays the utilization of an event by a facility until the arrival of all of its predecessors is confirmed. This ensures a causal delivery of events to the facilities. The sorting process also helps in post-restricting the events according to the individual needs of each facility (Section 6.1).

4.2 CODE 2 Environment

Programming in CODE 2 environment is done by drawing nodes and arcs, and then annotating them. Nodes interact with each other through arcs that bind names in one scope to the names in another scope.

In Figure 1-4, “CODE 2 Graph for Block Triangular Solver (DoBTS),” on page 9, **Dist**, **Mult**, **Gath** and **Solve** nodes are UC (unit of computation) nodes. UC nodes provides the sites where sequential computation takes place. As such, they correspond to the abstraction of the computation actions considered by our model. Annotation of a UC node consists of a list of input ports, list of output ports, list of local variables, a firing rule, a sequential computation and an output firing rule. See Figure 4-3. The annotations also specify whether it is a start node, a terminating node or neither.

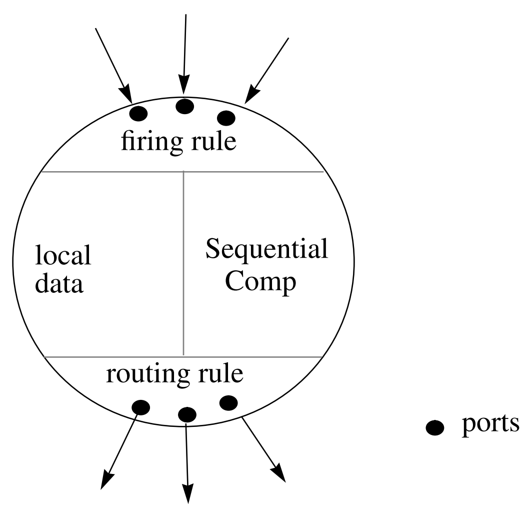


Figure 4-3. CODE 2’s Unit of Computation (UC)

A data-flow arc between the nodes such as the one between **Dist** and **Solve**, binds the output port of **Dist** to the input port of **Solve**. Ports are queues of data that leave or enter a node. Each node uses its own local names for the ports so that nodes can be reused in different contexts.

A firing rule for a UC serves two purposes. It defines condition under which the node is permitted to execute. Second, it describes which local variables would have data placed in them that has been removed from the designated input port. A routing rule for a UC determines the data that will be placed on the output ports.

Figure 4-4 shows the main program graph for the block triangular solver example explained in Section 1.2.1. Nodes **rdsys** and **dobts** in this graph are call nodes. They call other graphs. For example, **dobts** calls the graph DoBTS shown in Fig. 1-4. Thus, CODE programs can be hierarchical. Nodes like creation parameter and interface nodes aid in this hierarchical composition. Creation parameters are visible to all the nodes within a given call graph. Interface nodes of a call graph are like formal parameters to a function call. They are thus aliases for the nodes within the call graph.

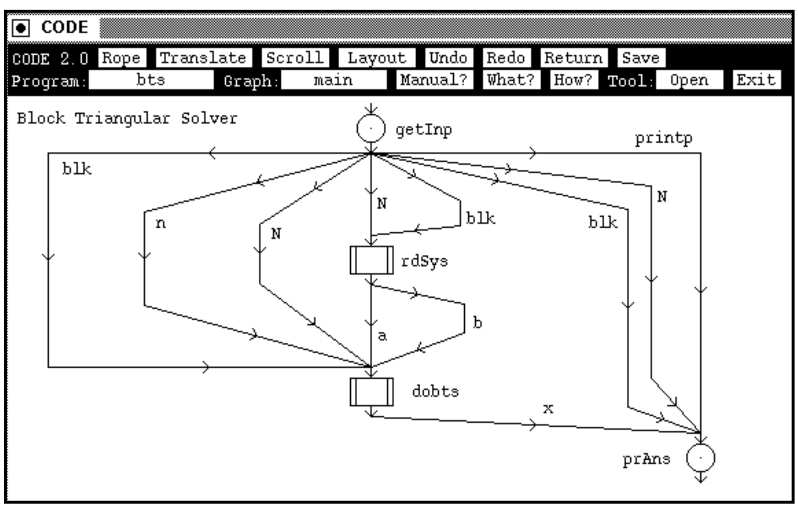


Figure 4-4. Block triangular solver main program graph.

There are also name sharing nodes that contain the objects to be shared by various UCs. A UC declares itself to be a reader or a writer of a shared object of a name sharing node. The UC declares its intention to share with the node through a shared dependence arc. See [New93] for more information about CODE 2 program graphs.

4.2.1 Templates and their Executable Instances

Nodes of a CODE 2 program graph are type templates. It is their instances that exist and execute at run-time, and create other instances by sending data to them. A policy of lazy creation of instances is employed. That is, executable instances are not created until they are needed. For example, an instance for a template like **Solve** in Figure 1-4 will not be created unless data is sent to it. This happens when **Dist** puts data on its output port that is bound to the input port of **Solve**.

A UC template may be instantiated one or more number of times. The number of times it gets instantiated is determined dynamically at run-time. This depends upon the binding on the output arc connected to a port specified in the output rule. For example, the binding on the arc connecting templates **Dist** and **Mult** in Fig. 1-4 specifies that the data placed on the output port **B_TO_M**[i] of **Dist** goes to an input port of the instance $\mathbf{Mult}^{\mathbf{i}}$. **Dist** node can now determine the number of **Mult** instances in existence by sending data on appropriate indices of the output port **B_TO_M**. If it sends data on output port **B_TO_M**[i] where **i=1..N-1**, the binding would cause the data to be received by **N-1** instances of **Mult**. Instances $\mathbf{Mult}^{\mathbf{1}}$, $\mathbf{Mult}^{\mathbf{2}}$, ..., $\mathbf{Mult}^{\mathbf{N-1}}$ would, then, be created if they are not already instantiated.

Instances of a UC and call node templates exist in the CODE 2 run-time environment as structs. The struct for a UC node instance contains its template UID, and its index (see Figure 4-5). The UID is a unique identifier assigned to each template node by the CODE 2 front-end. The index helps in differentiating one instance of a template from another in a given instance of the call node template. Thus, the **Mult** instances considered above are differentiated by their indices. In addition, each instance contains the local state, the nesting context (Section 4.4.2), and some other information needed by the run-time environment. Action-specific debugging information needed by the instrumentation (Section 5.1.2), and the various facilities (Section 6.2) is also kept in the struct for **`UcNodeInst`**.

```
typedef struct UcNodeInst *UcActionId;

struct UcNodeInst {
       TmpltUID            uid;
       Index               index;
       CallNodeInst        *parent; // Nesting context
       :
       :    // Info needed by runtime
       :    // Local state
       :    // Action-specific debugging information
};
```

Figure 4-5. Internal representation of a UC node instance (UC action).

Note on Terminology

As unit of computation nodes (UC) nodes specified in a CODE 2 program graph are template types, the set of UC templates corresponds to the set of template computation actions, Σ_P, of the unified model (Section 3.1). Consequently, the set of UC instances created at runtime corresponds to the set of executable computation actions, Σ (Section 3.1.1). In the following sections, the term "template" will be explicitly used whenever a reference to a UC template or a template computation action is intended. UC instances will be referred to as such or as executable computation actions, or simply as UC actions.

In Figure 4-5, note that the id of a unit of computation action (**`UcActionId`**) is a **`typedef`** for a pointer to the struct **`UcNodeInst`**. The phrase "inside the action" will, therefore, be used to mean "inside the struct for that UC node instance". Similarly, the phrase, "id of the action" will be used to mean "an object of type **`UcActionId`** that points to the struct for that UC node instance". Thus, knowing the id of a UC action implies that all the information stored in the struct of the **`UcNodeInst`** is also available.

4.2.2 UC Execution Event

An event for the CODE 2 implementation of the unified concurrent debugger is an execution occurrence of a unit of computation or UC action. A UC action starts executing by checking the firing rules. See Figure 4-6. If the condition of a rule evaluates to “true”, data is removed from the input ports specified in the rule, and the data values are bound to the local variables. The UC action, then, tries to acquire read and write locks for any shared objects which it has declared that it wants to access. On success, the action performs the specified sequential computation. It then selects a satisfied routing rule, and sends out data on the output ports specified by the rule. This completes one execution (event) of the action, and modifies the state of input ports of those actions to which data was sent. This also readies for execution the actions to which data was sent.

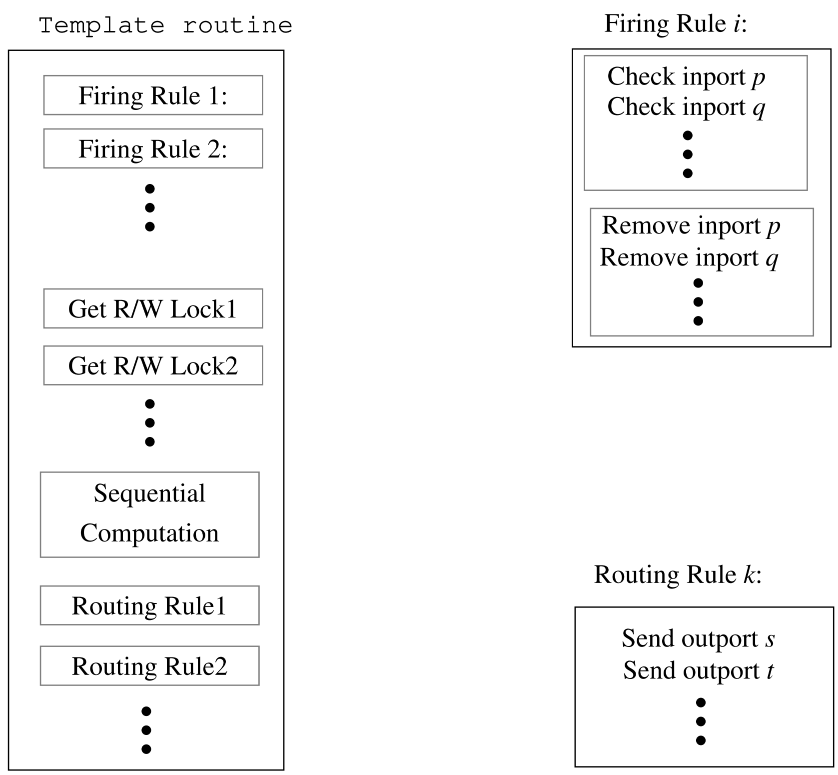


Figure 4-6. Execution of a Template Instance.

4.2.3 CODE 2 Runtime System

The CODE 2 runtime system implements the execution of a UC action by a routine that is generated by the translator for each UC template. The template routine takes as an argument the id of the action whose execution it is going to implement. The id allows the routine to access the local state and nesting context of the UC action. Note that at the time of its creation, each action is provided with a pointer to the template routine that implements its execution.

The CODE 2 execution environment for Sequent shared memory machine consists of a ready queue, a number of worker tasks, and the template routines that implement actions [New93]. The ready queue contains ids of the actions that have received new data on their input ports. The worker tasks are light-weight *FastThread* threads. Each worker loops around looking for a UC action to run from the ready queue. On finding one, it executes the action by running its corresponding template routine. The execution of the action would, in turn, ready for execution the actions to which the data is sent.

CODE 2 is a retargetable, machine independent parallel programming environment [New93]. In addition to the parallel implementation of the runtime system on the Sequent shared memory machine, there is a serial and a distributed implementation. The serial implementation of the runtime system is on a Sun-4 workstation and is quite similar to the parallel Sequent implementation. The only difference is that the serial implementation employs only one worker task and no synchronizations. The distributed implementation of the CODE 2 runtime system is on a network of workstations using PVM message passing primitives [Vok94].

The description of the unified debugger given below is for the parallel Sequent implementation. The serial implementation of the debugger is very similar. Distributed implementation of the unified debugger is discussed in Section 8.3.2.

4.3 Debugging Environment

Figure 4-7 shows the debugging environment for the Sequent shared memory parallel machine. As mentioned above, worker tasks execute the UC actions by running their corresponding template routines. Instrumentation added to these routines provides debugging support for replaying, recording and restricting traces (Chapter 5). The instrumentation sends an on-the-fly stream of information about the execution of these actions to the debugger on the `NewsQ`. The worker tasks and the debugger task can, therefore, run asynchronously. The debugger task is then free to interact with the user, respond to the execution information, provide display and other facilities, and maintain debugging information (Chapter 6).

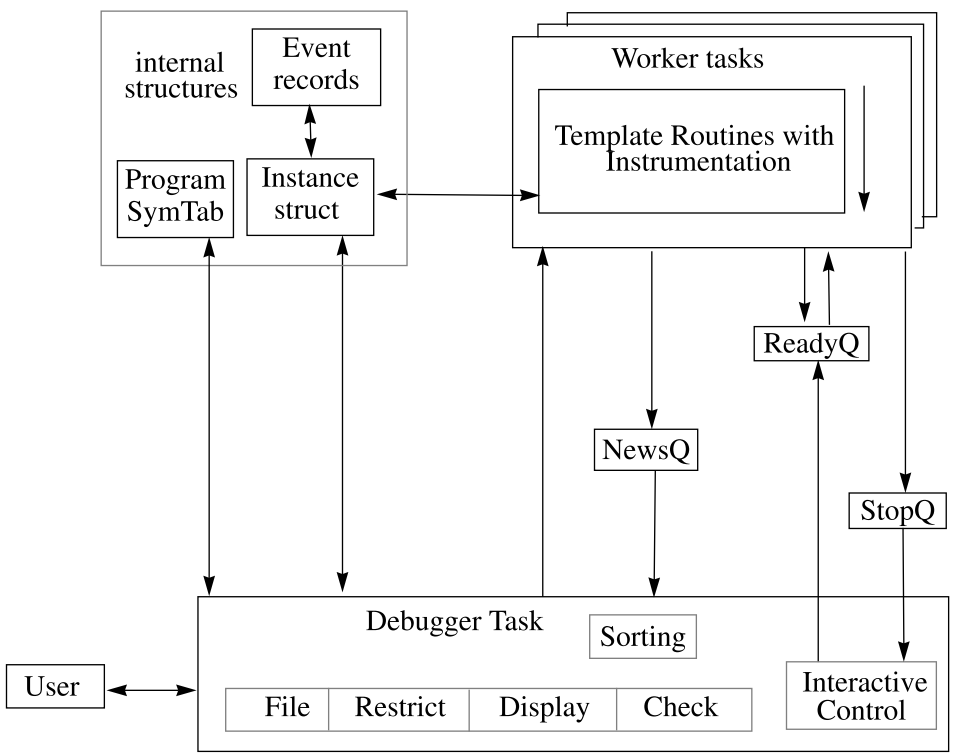


Figure 4-7. Debugging Environment

Information about the symbols and templates of the program, actions, and event occurrences are maintained in various internal structures

(Section 4.4). These structures allow the debugger to map between the templates, actions, and events (Section 4.5).

4.3.1 News from the Instrumentation

Instrumentation inserted in the template routines sends news about the execution of UC actions to the debugger task. The news are about the event occurrences, state changes and encountering of breakpoints.

News About the Event Occurrences

After each execution of a selected action, the instrumentation sends a news item to the debugger task containing the event record that was generated or used during the execution. Note that during the recording mode, the instrumentation generates a new event record for each execution of a selected action. These records are later used during the replay mode to re-execute the action.

The information contained in these records is utilized by the facilities for model checking, display, filing or for further restriction of the trace. So, the records are sent to the debugger task only if the user has requested one or more facilities.

News about the Stopped/Running State of an Action

The instrumentation can change the execution state of an action when it stops the action by enqueuing it in the **`StopQ`**. The state also changes when the instrumentation resumes the execution of the action from wherever it was stopped. The news about the change in the state is sent to the debugger task. The debugger task then forwards the news to the user. The user can, then, employ the interactive facility for querying the state of the stopped action (Chapter 7). The interactive facility can later be employed for continuing the execution of the action. This is done by enqueuing the id of the action in the **`ReadyQ`**.

News about the Encountering of Breakpoints

A user can set various kinds of breakpoints for an action (Section 7.2). Depending upon their type, the breakpoints are conditionally or unconditionally evaluated at selected points during the execution of the action. The instrumentation sends news about the results of these evaluations to the debugger task.

4.3.2 The Debugger Task

The debugger task continuously loops around looking for the input from the user, and for the execution information coming from the instrumentation on the **`NewsQ`**. See Figure 4-8.

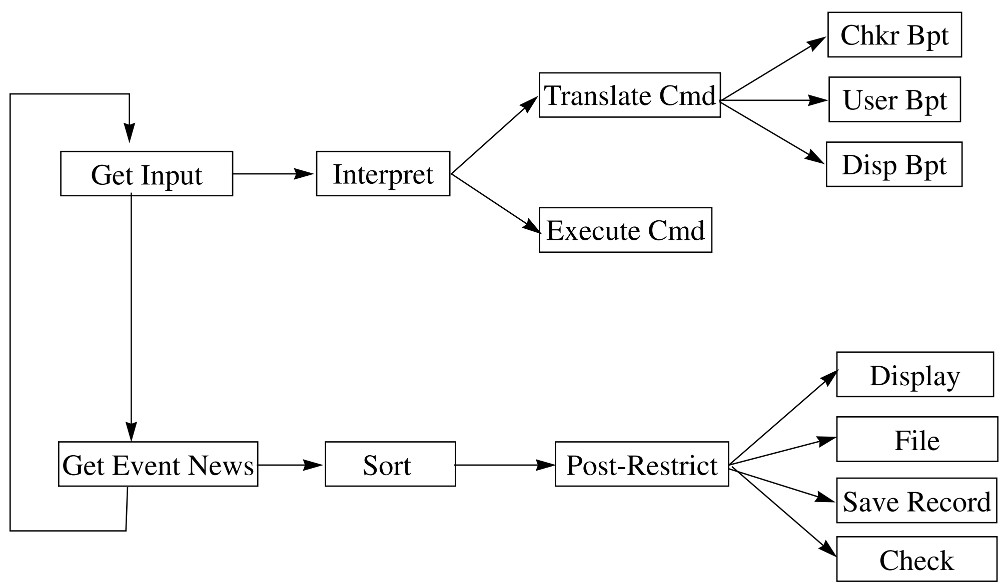


Figure 4-8. The Debugger Task.

On recognizing some input from the user, the debugger interprets the command. If the command is executable, it is immediately executed. Otherwise, it is handed over to one of the facilities for translation. The facility, then, sets breakpoints for the specified action(s) where the translated command will be evaluated. The user-breakpoint (Section 7.2), checker-breakpoint

(Section 6.5.4), and the display-breakpoint (Section 6.4) are respectively set by the interactive, checking and display facilities.

Any execution information that is seen by the debugger task on the **NewsQ** is handed over to the sorting facility. The sorting delays each event until its causal arrival is confirmed. Once an event gets sorted, post restriction takes place (Section 6.1). The sorted event is then given to the facilities selected for displaying, checking and/or filing. The checker facility, if selected, evaluates the translated commands associated with the checker breakpoints and signals any exceptions (Section 6.5). Similarly, the display facility, if selected, presents the event information as specified by the user (Section 6.4). If archiving of the event trace is requested, then the event is filed in the specified format. Once the facilities have utilized the information in the record of the sorted event, the record is deleted if saving of the event trace has not been requested by the user (Section 6.3.2). Note that the user can request to have the trace of event records internally saved for a later replay and/or postmortem utilization.

4.3.3 Communicating with the Instrumentation

The debugger task provides the instrumentation with information (Section 5.1) that allows it to properly execute the actions in the selected mode. This includes:

1. Global information that applies to the execution of all the actions.

This is distributed by means of shared variables.

2. Action-specific information that may be different for different actions.

This is stored inside the selected actions. The instrumentation obtains access to this information when the template routine provides it with the action id.

4.4 Internal Structures

At the top level, the unified debugger maintains three structures: The debugger symbol table, the dynamic instance tree, and the event graph. The symbol table contains an internal representation of the symbols and templates of the program. The dynamic instance tree contains information about the runtime instantiations of templates of the program into instances. Event graph captures the orderings between the execution occurrences of actions.

4.4.1 Debugger Symbol Table

The symbol table allows the debugger to interact with the user. It contains information about user defined types, local variables in different scopes, input ports, output ports, and call graphs. It also contains information about the UC, NS, and call node templates of the CODE 2 program graph. The set of template computation actions, Σ_P (Section 3.1) is, therefore, available through a symbol table lookup for the UC template symbols.

The CODE 2 compiler/translator was modified to add symbol table information to the generated code. The added information is in the form of a routine which is linked with the other code generated by the translator. At start-up, the debugger invokes this routine to enter the information in its symbol table. Figure 4-9 shows a part of the debugger symbol table entry routine for the block triangular solver example. The routine contains a call for each symbol that is to be entered in the debugger's symbol table.

```
void _c2_dbgSymEntry()
{
        _c2_entrPgmSym("bts", 46);
        _c2_entrArrayTypeSym("real","Vector",46);
        _c2_entrArrayTypeSym("Vector", "Matrix", 46);
        _c2_entrGphSym("DoBTS", 26, 46);
        _c2_entrUCSym("Gath",31,26);
        _c2_entrVarSym("b", -1121, "Vector", 31);
:
:
}
```

Figure 4-9. Translator generated symbol table entry routine.

Information about a symbol includes name of the symbol, block containing the symbol, type of the symbol (if any) and the UID of the symbol. The UID refers to the unique id assigned to the symbol by the CODE 2 compiler. In Figure 4-9, entry for the variable "`b`" provides information about its type (`Vector`), its UID (`-1121`), and the UID of the enclosing block (`31`) in which the variable is defined. Note that the UC template symbol "`Gath`" has a UID of `31`, and is thus the enclosing block for "`b`". In CODE 2, names given to UC templates like **Gath** and **Mult** are optional because the graphical frontend can uniquely identify each node from its graphical context. These UIDs are used by the runtime to implement lazy creation of template instances. The debugger uses them for mapping the symbols to their object addresses (Section 7.4.3) and for mapping between the templates and their instances (Section 4.5).

4.4.2 Dynamic Instance Tree

The run-time elaboration of the templates of a program graph into instances introduces two types of relationships: One is captured by the elaborated graph and the other is captured by the dynamic instance tree.

1. The elaborated graph shows the relationships established by the flow of data and the accesses of shared objects between the UC actions.

Figure 4-11 shows the elaboration of the program graph as a result of the flow of data between instances of the UC templates of Figure 1-4. These data-flow and shared access relationships are available in the debugging information stored inside each UC instance. The display facility uses this information to construct the elaborated graph, (Σ, F), as it is presenting the event information to the user (Section 6.4). As such, the primary use of the elaborated graph is to communicate to the user the synchronization structure of the computation resulting from the instantiations of the UC templates at runtime.

2. Dynamic instance tree shows the hierarchical relationship established by the nesting of an instance within the scope of another instance.

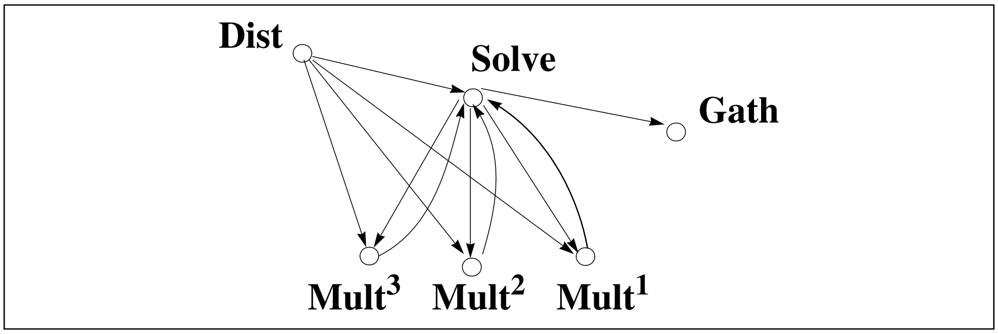


Figure 4-10. Data-flow relationships of the dynamic instance graph.

Creation of a UC action inside a given call graph instance gives rise to such a nesting. The instances are organized in a tree. Each tree node represents an instance of a UC, call graph or an NS (name sharing) template of the program graph. The node contains the type and the address of the node instance it represents. The tree is rooted at the main program graph. Figure 4-11 shows such a tree for the main **bts** program graph. A call instance contains a list of all of its children nodes. Any of these children can itself be a call graph instance. For example, the child node of **bts** in Figure 4-11 is **dobts**, which is itself a call graph instance and thus contains other instances.

Whenever a new instance is created inside a call instance, a new child node is introduced in to the list of children for that call instance. Additionally, the newly created instance is informed about its nesting context by storing a pointer to the parent call instance inside the struct for the new instance.

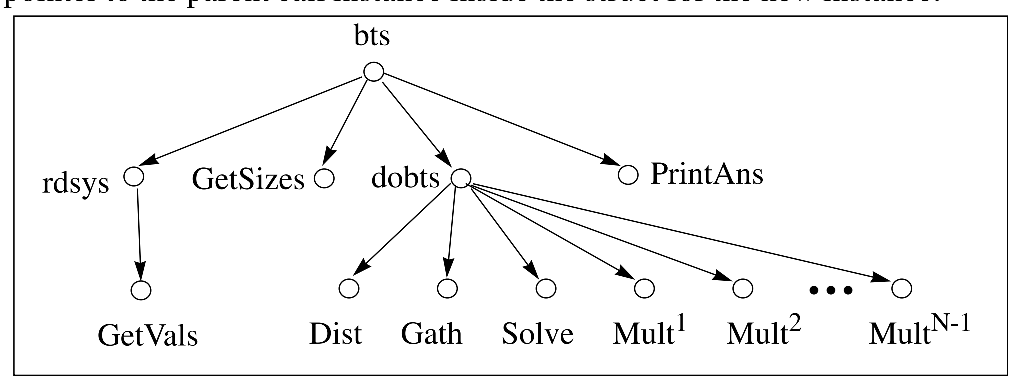


Figure 4-11. Nesting relationship in the dynamic instance tree.

As the execution proceeds, the newly created instances keep getting added to the dynamic instance tree. The tree therefore contains the current information about all the dynamically created instances. It is, therefore, also of use to the run-time for managing the creation of instances. To avoid duplication, our implementation uses the tree maintained by the runtime. It obtains a handle to the graph by storing the identity of the root of the instance tree. Note that the instance of the root program graph is created by the runtime system at start-up.

The dynamic instance tree is mainly used by the debugger for organizing and maintaining the internal debugging information. The root of the tree serves as a representation for the scope of the program. The debugger uses the handle to the root of the dynamic instance tree in implementing functionality that requires iterating on all the instances. For example, determining the membership of the set of executable computation actions, Σ (Section 3.1), selecting a particular scope (Section 4.5.1), determining the effective recording option of all the instances (Section 5.3.1), preparing all the instances for a new run (Section 7.1.3) requires such iteration.

4.4.3 Event Records and Event Graph

An event is an execution occurrence of a UC action. It is therefore identified by the id of the action (a pointer to the struct of its UC node instance) and its execution count. The event record of the execution of a UC action contains information about the ids of the predecessor events and the counts of the successor events. See Figure 4-12. Note that the id of the action in the event record (and the event id) maps the event to its corresponding action and, thus, supports the mapping μ: $V \rightarrow \Sigma$ needed by the unified model (Section 3.2.2).

The list of flow-predecessors contains ids of the events whose data was removed by the action from its input ports. The list of shared predecessors contains ids of the events that last wrote the objects that were shared accessed by the action in this execution. These lists are the implementations of the sets $u_i.P_F$ (Section 3.2), and $u_i.P_S$ (Section 3.4) for an event $\boldsymbol{u_i}$ in V.

```
typedef struct UcEvId *UcEvId;

struct UcEvId {
        UcActionId id;
        int ExecCnt;
};

struct UcEvRec {
        UcActionId id;
        int ExecCnt;
        List fpl;            // list of flow predecessors
        List spl;            // list of shared predecessors
        int fscnt;           // count of flow successors
        int sscnt;           // count of shared successors
};
```

Figure 4-12. Event record and event id.

The number of flow successors are the number of events to which the data was sent via the data-flow arcs. The number of shared successors is the number of readers or writers that accessed the shared object written by the event. Note that the number of shared successors is more than zero only for the actions that write any shared object. This information is needed during topological sorting to delete the information that is no longer needed by any of the facilities (Section 6.3.2).

The display facility (Section 6.4) uses the event record information to construct the event history graph as shown in Figure 4-13. This is a pictorial representation of the pomset $(\Sigma, V, <^{C}, \mu)$ considered in Section 3.2.2. The one-to-many relationship that may exist between an action and its events (Figure 4-14) is maintained by keeping the list of event records inside the action. Note that the set of events, V, is, thus, a union of the lists of events kept inside each action in Σ. Keeping the event records within an action, allows the replay instrumentation to access the event records whenever it has the action id (Section 5.1.2). Moreover, the debugger can present to the user all the event information of a given action (command **`lse`** in Table B-1).

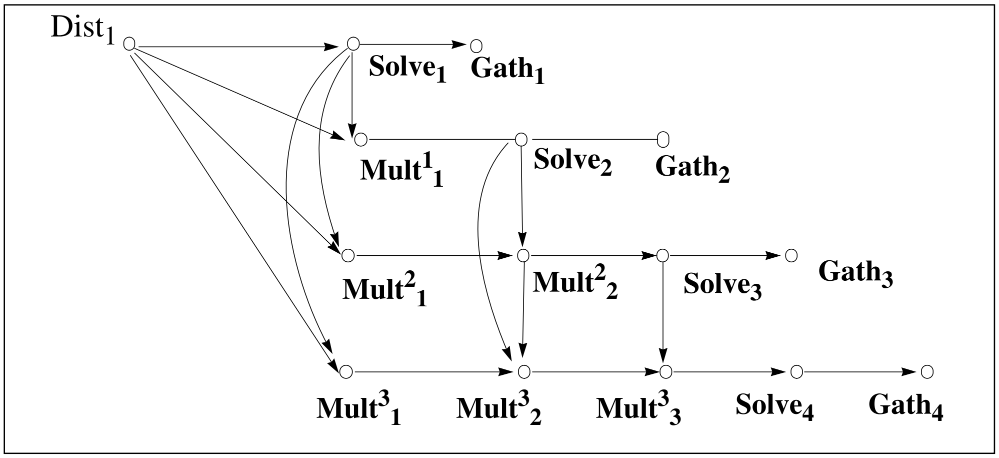


Figure 4-13. Eevent history graph of the DoBTS example for N=4.

4.5 Selecting a Scope

The debugger maintains information about the currently selected scope. The selected scope can be that of the program, a call instance, a UC action, or an event. Note that the context of the currently selected scope is used to interpret context-sensitive commands like **`print <expr>`**, and **`stop`** (Appendix B). The user must select an appropriate context before such commands are issued. For example, the program scope is selected for issuing a global command to stop the execution of all the actions (Chapter 7). Similarly, for querying the state of an action, the action must be selected before issuing (say) a **`print`** or a **`dump`** command. Note that the root of the dynamic instance tree serves as a representation for the scope of the program. Several commands are available to the user for selecting a scope (Appendix B).

The mappings existing between the symbol table, dynamic instance tree and the event graph (Figure 4-14) allow the debugger to select a scope specified by the user. The symbol table contains symbols for UC, call node and NS templates. These symbols are used to construct unique identifiers for the instances as explained below. Given such a unique identifier for an instance, the debugger can traverse the dynamic instance tree, and locate the instance's struct. Once the struct is obtained, the debugging information contained in that struct becomes available (Figure 4-5). Note that the debugging information for a UC action contains the list of its events (Section 5.1.2). If an event of the action is to be located, then this list is searched for the given execution count to obtain the desired event record.

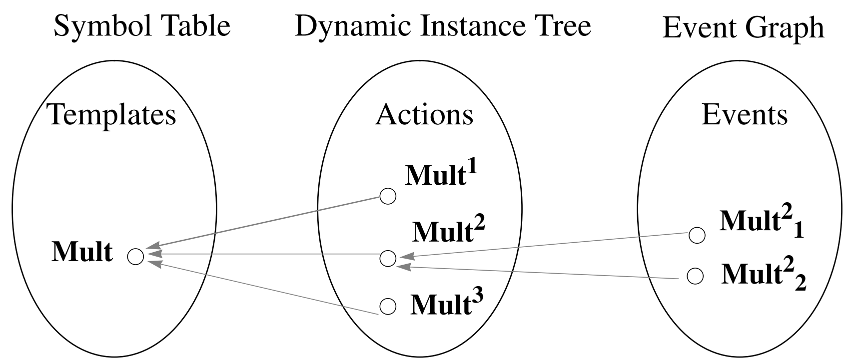


Figure 4-14. Mappings between templates, actions, and events.

Unique Identification of Instances

It is possible to have several instances of the same template with the same index in various call graph instances. Unique identification of an instance, therefore, requires the nesting context of the instance in addition to its template id and the index. Consider for example, that an instance of the main program graph contains two instances of DoBTS call graph; **dobts**$^{\mathbf{1}}$ and **dobts**$^{\mathbf{2}}$. If both contain a UC action **Mult**$^{\mathbf{3}}$, then the two **Mult**$^{\mathbf{3}}$ actions in **dobts**$^{\mathbf{1}}$ and **dobts**$^{\mathbf{2}}$ would have the same template UID and the same index (i.e. 3). The only way of distinguishing between the two actions is their nesting context. This involves giving each instance a path-name similar to the one Unix assigns to a file in a directory. Then, the two UC actions are uniquely identified by **/dobts**$^{\mathbf{1}}$**/Mult**$^{\mathbf{3}}$ and **/dobts**$^{\mathbf{2}}$**/Mult**$^{\mathbf{3}}$. Therefore, commands that refer to UC, call graph and NS instances require the full pathname (Appendix B).

4.5.1 Selecting an Instance with a given Pathname

Searching for an instance with the given pathname is recursive and starts at the root of the dynamic instance tree. The search for (say) **/dobts**$^{\mathbf{2}}$**/-Mult**$^{\mathbf{3}}$, starts with the lookup for the name **dobts** in the symbol table. This yields the corresponding symbol which provides the UID of the **dobts** template. The search then examines each child of the root for an instance with UID of **dobts** and index of 2. As each instance contains the UID of its corresponding template and its index, the search would eventually locate the instance **dobts**^{2}, if it exists. As **dobts** is an instance of a call graph, and the path name still has an extension (**Mult**^{3}), the search continues by descending inside the **dobts**^{2} call instance and examining its children. The lookup for **Mult** in the symbol table yields its UID. The index 3 is given in the pathname. So, the search examines the children of **dobts**^{2} for an instance with the UID and index of **Mult**. If the search fails at any of these steps, the search exits with the error status. The search succeeds if all the instances given in the pathname have been located in their proper context.

4.5.2 Constructing the Pathname for an Instance

Given the UC action id, the full pathname of the action can be constructed. The struct for each instance contains its UID, its instance index and its nesting context. Using the UID, the symbol table lookup yields the symbol for the corresponding template. As UID is unique for a template, there is no ambiguity. The nesting context of an instance is available as a pointer to the parent call instance (Figure 4-5). The UID in the parent instance struct allows us to obtain the symbol for the parent graph. The symbols for each of the parent along the path is, thus, obtained recursively. The recursion ends at the root graph instance which does not have a parent (indicated by a nil pointer).

Chapter 5. Debugging Instrumentation

Debugging instrumentation is inserted in the executable code generated by the CODE 2.0 translator to relieve the debugger task from micro-managing the execution of UC actions. This obviates the need for intrusion from a "centralized" monitor/debugger as the instrumentation takes over the responsibility for recording the execution information, controlling what is recorded, and controlling the execution itself. The debugger task, then, uses the information generated by the instrumentation to implement the facilities provided to the user. The instrumentation is in the form of calls to a library of runtime routines that perform various action-specific debugging activities. The calls are inserted at different points in a template routine where they can record, replay, restrict, and interactively control the execution of an action. See Table 5-2. Before explaining these calls, this chapter explains the information needed by the instrumentation to ensure the execution of actions in the proper debugging mode.

5.1 Information Requirements

The debugger requires both global information and action specific information. Global information is provided through shared variables in the Sequent implementation. Action-specific information is provided by storing it inside each action. The instrumentation obtains the action-specific information when the instrumentation calls are invoked from the template routine with the action id as a parameter. Recall that id of the action makes available the debugging information inside the struct for that action.

5.1.1 Global Information

The information about the currently selected mode for running the program is available in **`CurMode`**. See Table 5-1. During the execution of each action, **`CurMode`** selects appropriate instrumentation for the full-recording, restricted recording and the replay mode. Commands for **`next`**-ing and **`stop`**-ping of actions provided by the interactive facility (Chapter 7) require cooperation from the instrumentation. Information about the current global state of the execution which is available in **`CurState`**, helps the instrumentation in implementing the global **`next`**-ing and **`stop`**-ping of actions. The instrumentation sends information about the execution of actions to the debugger task if any of the debugging facilities described in Chapter 6 have been selected. **`SendNews`** informs the instrumentation whether to send the news about the execution events of actions or not.

Table 5-1. Global Information used by the Instrumentation.

Global Information	Options	Use
CurMode	Recording[a]	Select recording instrumentation
	Replay	Select replay instrumentation
	DebugOff	Turn-off all instrumentation
	PerfTr	Performance tracing instrumentation
	Postmortem[b]	Facilities use pre-recorded trace
CurState	{Stopped,Nexting, Waiting, Stopping, Finished	Provides interactive control over the execution of actions.
SendNews	Boolean	True if the debugger task needs EvRec for any of the facilities.

a. Section 5.3.1 explains how the full-, restricted-, and off-recording modes are selected
b. Postmortem mode is not used by the instrumentation.

5.1.2 Action Specific Information

Table 5-2 describes the use of action-specific debugging information kept in the struct **`UcNodeInst`** (Figure 4-5) for each UC action. Some of these variables are selected by the user, while others are maintained by the instrumentation.

The user controls the amount of recording by selecting appropriate recording options. This determines the **`EffRecOpt`** (Section 5.3.1) which informs the instrumentation what information to record in the restricted recording mode. The user may set a breakpoint, or issue a command to control the execution of an action. **`UsrBpOn`**, **`UCCommand`**, and **`IsStopped`** inform that portion of the instrumentation which implements the interactive facility what actions to take. Instrumentation calls are inserted "at" various points in a template routine (Table 5-2). **`CurAt`** indicates the most recent instrumentation call made by an action. This is helpful in resuming the execution of an action from the point where it was stopped by the instrumentation.

Table 5-2. Action-specific information needed by the instrumentation.

Field	Type	Usage
`ExecCount`	int	Recording, replaying, checking
`CurEvRec`	pointer to **`EvRec`**	Recording, restricting, replaying
`EvRecList`	list of **`EvRec`**	Recording, replaying
`CurFp`	pointer	Replaying
`SelRecOpt`	{**`full`**, **`restrict`**, **`off`**}	Determines **`EffRecOpt`**
`EffRecOpt`	{**`full`**, **`restrict`**, **`off`**}	Recording, restricting
`CurSp`	pointer	Replaying
`CurAt`	**`At`** (see Table 5-2)	Interactive
`UCCommand`	**`{cont, next, stop}`**	Interactive
`IsStopped`	Boolean	Interactive
`UsrBpOn`	Boolean	Interactive

The instrumentation maintains the current count of the number of times an action has executed. **`ExecCount`** is updated after each execution of an action.

Whenever an action begins a new execution, **`CurEvRec`** is made to point to an appropriate record. During replay mode, **`CurEvRec`** points to the event record whose information is being used to replay the current execution.

During full recording mode, **`CurEvRec`** points to a newly created record that will contain the information recorded for the execution. If executions of an action are not being recorded, then for each execution of the action **`CurEvRec`** points to the same record. This record helps in propagating information about the predecessors of the event to its successors.

Event records generated by the recording and restricting instrumentation can be internally saved in a list. The records in **`EvRecList`** may later be used for replay. During replay mode, **`CurFp`**, and **`CurSp`** point to the predecessors that are currently expected by the action to ensure its replay.

5.2 Full-Recording Instrumentation

In the full-recording mode, executions of all UC actions are recorded irrespective of the user selections for each action. Event records are generated for every execution of an action, and they contain flow-predecessor orderings, as well as shared predecessor orderings (if any exist).

5.2.1 Determination of Flow Predecessors

The instrumentation determines the $<^F$ orderings (Section 3.2) by obtaining the id of the predecessor event from the data removed from each input port. In order to allow the successor events to determine their predecessors, current event id is appended to the data sent out on each output port.

In CODE, ports of a UC node are implemented as queues. The run-time uses a container object for carrying values of different types on these queues. We make provisions in the definition of the container object to carry the additional information that the recording instrumentation is going to append.

Data is removed from an input port by dequeuing the container object from the queue representing the input port. At this point, the **`DetachFpAct()`** routine detaches the predecessor information from the object and adds it to the predecessor list in the current event record.

Sending of a value on an output port is more involved. The run-time first locates the destination action using the binding specified for the output port. It creates the action if it is not already in existence. Then, the value to be sent is placed in a container object. At this point, **`AppendFpAct()`** appends the identifier of the current event to the object. The object is then inserted in the queue that represents the input-port of the destination action. Note that each event record additionally contains a count of number of successors. We increment this count whenever the data is sent on an output port.

Table 5-3. Instrumentation inserted in a template routine.

Instrumentation Breakpoint Actions	Point **`At`** which inserted See Figure 4-6.	Usage	Action
`void BeginEvAct()`	Start of template routine	Recording, restricting, replaying, interactive	Get `CurEvRec` for a new UC exec
`void EndEvAct()`	End of template routine	Recording, restricting, replaying	Send and/or save CurEvRec info
`void DetachFpAct()`	After removing from an input port	Recording, restricting, interactive	Remove dbg. info. appended to data
`void AppendFpAct()`	Before sending on an output port	Recording, restricting, replaying, interactive	Append dbg. info. to data sent out
`void ReadSpAct()`	About to acquire a R/W lock	Recording, restricting, replaying	Read appended info to shared object
`void WriteSpAct()`	About to release a write lock	Recording, restricting, replaying	Write to info appended to shared object
`Boolean CheckFpAct()`	Checking/ removing from an inport	Replaying	Check id appended to input data
`Boolean CheckSpAct()`	About to acquire a R/W lock	Replaying	Check id appended to a shared object
`Boolean AftCompAct()`	After the sequential computation	Interactive	Evaluate any user breakpoints here

Table 5-3. Instrumentation inserted in a template routine.

Instrumentation Breakpoint Actions	Point **`At`** which inserted See Figure 4-6.	Usage	Action
`Boolean BefCompAct()`	Before the sequential computation	Interactive	Evaluate any user breakpoints here

5.2.2 Determination of Shared-Predecessors

The instrumentation determines the shared predecessors (Section 3.4) by reading in the event ids that were appended to the shared objects by their last writers. This is done for each shared object accessed by the action in the current execution. In order to enable the successor events to determine their shared predecessors, the instrumentation replaces the event id appended to the shared object that is being written, with the current event id.

In CODE, a shared object resides in a name sharing node action. This is implemented by declaring a shared object as a field in the struct representing the name sharing node. For each shared object inside a name-sharing node, we ask the compiler to declare an additional field in the struct. This new field is used for appending the predecessor information ($\eta.D$ and $\omega.D$ of Section 3.5).

An action obtains access to a shared object by acquiring a read or a write lock through a typical CREW protocol. Instrumentation is added where the locks are being acquired and released. When the action is about to acquire a read- or a write-lock to a shared object, **`ReadSpAct()`** reads the event id appended to the shared object. It adds the event id to the list of shared predecessors kept in the action's current event record. Similarly, when an action releases a write lock, **`WriteSpAct()`** replaces the appended id with its current event id. This allows the events that later access the object to determine their predecessor.

```
struct SharedPred {
            EvId evid;
           int rdrcnt;        // reader count; significant
     };                       // only in full recording mode
                              // and replay mode.
```

The predecessor information that is written to or read from the extension to to the shared object definition contains a reader-count in addition to the event id. This is helpful during replay when a writer must wait until all the readers of the object written by its predecessor have come and read the object (see also Table 3-2).

In **ReadSpAct()**, an action acquiring access to an object reads the event id appended to the object as explained above. In addition, however, if the action is acquiring read-access, then it increments the reader-count. But, if the action is acquiring a write-access, then it records the appended reader-count. In **WriteSpAct()**, the action releasing the write-lock, appends its id to the object, and initializes the reader-count to zero.

5.3 Restricting Instrumentation

In the restricted-recording mode, the user has several options to control the amount of recorded information:

1. Option to turn off or on, the recording of events of selected actions.
2. Option to record flow predecessor orderings with the shared predecessor orderings or without them.
3. Option to override the selected options for all actions in a given scope.

5.3.1 Determining the Effective Recording Option

The amount of recording is controlled by the "effective" recording option. **EffRecOpt** may be different from the option "selected" by the user.

SelRecOpt can be overridden by the parent scope's option. Table 5-4 describes the possible recording actions for UC and call instances.

The effective recording option of an instance depends upon (i) the instance's selected option, and (ii) its parent's effective option. The selected option is effective only if the parent's **EffRecOpt** is **Restrict**. Otherwise, the action's effective option is the one inherited from its parent action. That is,

```
if my parent's EffRecOpt is Restrict
            then
                    my EffRecOpt = my SelRecOpt;
            else /* Full or Off */
                    my EffRecOpt = my parent's EffRecOpt
```

This rule is applied recursively to all the instances of the dynamic instance tree starting from its root. The root's **EffRecOpt** is always equal to **SelRecOpt**. If a new option is selected for a call instance, then the rule is recursively applied to all the instances inside the scope of that instance.

Table 5-4. Effective recording options for UC and call instances.

Instance Type	EffRecOpt	Recording Action
UC	Full	Record flow and shared predecessors.
	Restrict	Record only flow predecessors.
	Off	Do not generate any event records.
Call	Full	Turn to full the recording of all internal instances.
	Restrict	Defer to individual selections for internal instances.
	Off	Turn off the recording of all internal instances.

Thus, full recording mode of the execution is chosen by selecting a **Full** option for the root of the dynamic instance tree, i.e. the program scope. This overrides the selections of all internal instances and forces a **Full** effective option for all the instances of the tree. Similarly, the restricted recording mode is chosen by selecting the **Restrict** option for the root. Similarly, the off recording mode is chosen by selecting the **Off** option for the root that turns off the recording of all the instances by overriding their selected recording options.

5.3.2 Turning Off the Recording of a UC Action

Each action has to maintain the list of its predecessors; even those actions not selected for full recording. The list of predecessors maintained by the actions that were not selected for recording, helps in establishing orderings between event occurrences of the selected actions (Table 3-1). Instrumentation for an action whose recording is turned off, forwards the list of the predecessor events to the successor events.

This requires that the provision for appending to the data sent on the queues should be capable of holding more than one event id; it must be a list of event ids. A similar provision is needed in the extension to the definition of a shared object. An event record is used for temporarily accumulating the predecessors. It is pointed to by the **CurEvRec**. During each execution, the predecessors are accumulated in this event record. By the end of the execution, these predecessors have been forwarded to the successor events.

DetachFpAct(), and **ReadSpAct**() are responsible for detaching/reading a list of event ids, and then concatenating the list to the list of predecessors kept in the current event record. Note that the list may contain one or more event ids.

Forwarding of the flow-predecessors information is done in **AppendFpAct()** where a copy of the list of flow-predecessors is appended to the data sent on the output port. Similarly, in **WriteSpAct()**, extension to the shared object definition is replaced with a copy of the list of shared predecessors.

5.3.3 Restricting the Recording of a UC Action

With the **`Restrict`** recording option, instrumentation only records flow-predecessors. It does not record shared predecessors. Therefore, it employs some of the instrumentation for the **`full`** option recording, and some of the instrumentation for the **`off`** option.

A new event record is generated for each execution. In this event record, **`DetachFpAct`**() and **`ReadSpAct()`** accumulate the flow and the shared predecessors as described above. As the list of flow-predecessors so accumulated is to be retained, **`AppendFpAct()`** will only forward the current event-id to its successors. However, the list of shared predecessors is not to be retained. So, **`WriteSpAct()`** will forward the entire list of shared predecessors to its successors as described above.

5.4 Replay Instrumentation

Replay instrumentation is responsible for enforcing that an action executes with the same orderings that were recorded in an earlier execution (Section 3.4). The record of orderings is available in the list of event records kept with each action. If the action is starting its ***n***-th execution, then **`BeginEvAct()`** would fetch the event record whose execution count is ***n*** from this list.

During the execution, the replay mechanism uses the recorded orderings for overcoming the two sources of non-determinism: One source of non-determinism is associated with the data-flow orderings and appears as choices in the firing rules of a UC. The other source of non-determinism is associated with the choices of ordering with which concurrent actions could obtain access to shared objects. Running the execution in replay mode, therefore, requires that the event records contain both the list of flow-predecessors and the list of shared-predecessors. This means that the available trace should have been recorded at the **full** option.

At the end of the execution, **`EndEvAct()`** sends information to the debugger task that it has successfully replayed the execution event.

5.4.1 Enforcing Previously Recorded Flow Orderings

It may appear that we can force an action to fire with the previously recorded orderings if we only ensure that the rule firing the current execution is the same as the one that fired the previous execution. This could have been the case if the input ports were not capable of receiving data from more than one action. However in CODE 2, input ports can receive data from different actions. The data gets merged in the port queue in some nondeterministic order. Then, simply ensuring that the same rule fires is not enough. We must ensure that (i) the same rule fires, (ii) the data that was checked on a port, was from the same predecessor, and (iii) the data that was removed from the port was also from the same predecessor.

Consider for example firing rule ***i*** in Figure 4-6. There is a checking phase in which ports ***p, q, ...*** specified by the rule are checked for data. This is followed (if the checking is successful) by a removing phase in which data is removed from each of the ports ***p, q, ...*** As mentioned above, ports are queues. So, the queue for inport ***p*** may have data from different predecessors. The replay instrumentation must therefore search the entire queue looking for the data from a particular predecessor. If data from the desired predecessor is found, then the checking of a port is successful. The checking phase completes successfully if each port contains data from the expected predecessor. Now the removing phase can begin. The replay instrumentation must again access each port and remove data of the expected predecessor. Simply, checking and removing from the head of the queue is not enough.

Note that each predecessor in the list is associated with the port-queue from which it was removed. The same queue must be checked and removed from during replay. This association can be established by recording not only the predecessor id, but also the id of the port from which it was detached. During replay, it is then simple to identify the predecessor that need to be checked for a given port. This, however, incurs extra overhead. Our implementation circumvents this overhead by the following observation.

The list of flow-predecessors in an event record contains event-ids in the order in which the firing rule removed the data from the input ports. This order is the same as the order in which the compiler arranges the checking of the input ports. We keep a pointer to the predecessor that is being checked. If the checking is successful, it is made to point to the next predecessor in the list. Otherwise, it is simply rewound to the first predecessor in the list.

Initially, when the checking of a firing rule starts, the current flow predecessor, **`CurFp`**, points to the first predecessor in the list. For each port specified by the rule, **`CheckFpAct()`** is used to check if there is data from the currently pointed predecessor. The instrumentation call goes through the queue checking for the data from the required predecessor. If it finds the right predecessor, the pointer is advanced to the next predecessor in the list. Then, the next port of the rule is checked. This continues until the predecessor list is exhausted or the checking fails to find data on the port, or data is found on the port but not from the same predecessor. The checking of a rule succeeds only if all the predecessors in the list match the appended ids of the data in each input port. If the checking succeeds then the action goes and removes the data.

5.4.2 Enforcing Shared Access in the Recorded Order

Note that the list of shared-predecessors available in the event record is in the order in which shared locks are acquired by an action. The current shared-predecessor pointer, **`CurSp,`** kept in the action is used to remember the next predecessor that needs to be checked. At the start of the execution, this is rewound to the first predecessor in the list. An action trying to acquire a read-lock or a write-lock checks the id appended to the object with the id given by the current shared predecessor.

If the action tries to acquire a read-lock and the comparison between the id appended to the shared object is the same as the id of the current-shared predecessor, then it will increment the appended reader-count in `CheckSpAct()` and will be granted the lock. A writer not only has to compare the id appended to the object with the current shared-predecessor, but also has to compare if the appended reader count is same as the reader count given in the current-flow predecessor record. This is done in `CheckSpAct()`. The writer is not granted the lock until both the conditions are met. After obtaining access, current shared-predecessor is made to point to the next shared-predecessor in the list. Note that a writer replaces the id appended to the object with its current event id, and initializes the reader count to zero as in the recording phase when it releases the lock.

5.5 Interactive Instrumentation

The instrumentation for supporting user interactions with the the program execution provides break-points in the execution of actions where user commands can be evaluated. These commands can either alter the normal course of execution, or they can cause the evaluation of other commands that can provide the user with on-the-fly information about the local states of actions. Note that all of this is done asynchronously from the debugger task.

5.5.1 Controlling the Course of Execution

This requires instrumentation for stopping the execution of an action, resuming its execution from whereever it was stopped, and nexting to the successors.

Stopping and Continuing

The concurrent and local state is undefined during the execution of a UC action when firing and routing is taking place. Stopping of the execution is only meaningful (and safe from the runtime point of view) after the firing takes place and before the routing starts. Currently, our implementation supports two places where the execution can be stopped; before the start of the computation and after the end of the computation.

BefCompAct() and **AftCompAct()** determine if the action has to stop its execution. If the **UCCommand** is **stop** or the **CurState** is **Stopping**, then the instrumentation would change the stopped status of the action to true, and would enque the action in the **StopQ**. After sending the "news" to the debugger that the action has stopped, the template routine returns prematurely. That is, if it is being stopped by **BefCompAct**, then it will not proceed with the sequential computation, and if it is being stopped by **AftCompAct**, then it will not start the routing. Thereafter it is the debugger tasks duty to inform the user, respond to the user queries, and ready the action for execution.

Eventually the action begins its execution. **BeginEvAct()**, then, finds out that the action was stopped. So, it makes the routine to jump to the location where the action was stopped. Note that **CurAt** in the action keeps track of the point from which the last instrumentation call was invoked.

Nexting to the Successors

If the **UCCommand** is **next** or the **CurState** is **Nexting**, then **AppendFpAct()** appends a stop command to the data being sent on the output port. This command is then seen by the **DetachFpAct()** of the successor event. If **DetachFpAct** sees a stop command, the command is simply placed inside the action. This command would then be evaluated after the firing by **BefCompAct()**. This would cause the action to stop as explained above.

5.5.2 User Breakpoints

A user breakpoints can be of two types: It may be an unconditional command to **stop** or **next** the execution as explained above. Or, it may be a conditional command that can be evaluated at different points during the execution of actions. The conditional command specifies the place where the condition is to be evaluated. It can also specify a condition on local state that must be evaluated at that place. The evaluation of the condition is done depending upon whether the **UsrBpOn** flag is true or false. This is set by the interactive facility at the time of creation of an action or before the start of the execution.

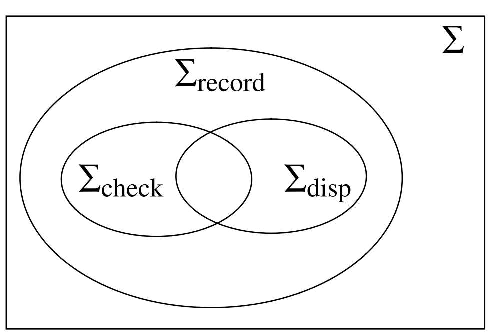
If breakpoint is on, the condition is evaluated and any associated commands executed. The commands may ask the action to stop its execution, print the local state, or next to the successors. Currently, we do this evaluation only in **`BefCompAct()`** and **`AftCompAct()`**. Section 8.2.1 explains how this can be done at other places in the sequential computation.

Chapter 6. Debugging Facilities

Event information generated by the instrumentation is analyzed and presented by various debugging facilities. This chapter explains the facilities provided by the debugger task for sorting, post-restricting, checking, displaying, and filing. The facility that provides interactive control over the execution of actions requires cooperation between the instrumentation and the debugger task and is explained in the next chapter.

6.1 Post Restriction

The restricted recording mode is useful in collecting only the event information which is needed for displaying and/or checking purposes. However, it is often the case that the information collected is much more than what the user is currently interested in viewing through one of the facilities. The debugging information may have been previously collected in a trace, or it may be that it is being collected on-the-fly from a full recording run or the replay run of the program. The user may want to have different views of this information; each one restricted to a different set of actions. The post restriction facility allows further restriction of the information recorded at runtime. Note that when the program is run in either the full recording mode or the replay mode the information arriving at the debugger task is typically much more than what the user is currently interested in viewing.



Σ : Set of all actions
Σ_{record} : Selected for recording
Σ_{disp} : Selected for displaying
Σ_{check} : Selected for checking

Figure 6-1. Relationships between selected actions.

6.1.1 Partial Orderings

The set of all the executable computation actions is Σ. This is maintained in the dynamic instance tree (Section 4.4.2). In the restricted recording mode, the user can select the actions whose executions would be recorded. Let this set be $\Sigma_{record} \subseteq \Sigma$. See Figure 6-1. The event records sent to the debugger task represent the executions of actions belonging to the set Σ_{record} . The partial order seen by the debugger task is therefore restricted to Σ_{record}, i.e. $(V, \Sigma, <, \mu)/\Sigma_{record}$. Note that in the full-recording mode or during the replay run $\Sigma_{record} = \Sigma$. In these modes, the amount of event information seen by the debugger task is $(V, \Sigma, <, \mu)$. This is much more than what the user may be interested in viewing through any of the facilities.

The unified debugger allows the user to select different sets of actions for checking and displaying. The set of actions whose execution behavior would be checked by the model checker is $\Sigma_{check} \subseteq \Sigma_{record.}$ The set of actions whose execution behavior would be displayed is $\Sigma_{disp} \subseteq \Sigma_{record}$. Note that there can be any relationship between Σ_{disp} and Σ_{check} not just the one shown in Figure 6-1. The display facility requires further restriction to $(V, \Sigma, <, \mu)/\Sigma_{record} / \Sigma_{disp}$, while, the checking facility requires further restriction to $(V, \Sigma, <, \mu)/\Sigma_{record} / \Sigma_{check}$. This means that the post-restriction for display should take place separately from the post-restriction for checker.

The restriction for checker and the display facilities takes place as each event is sorted, and the information about its predecessors becomes available. This is explained in Section 6.3.1 after describing the sorting process.

6.1.2 Options Available with Post-Restriction

The unified debugger's ability to do post-restriction allows the facilities to offer several options to the user. Table 6-1 describes the options offered by the facilities that display, check, save and file the recorded trace. The commands for selecting these options are given in Appendix B. If any option other than `off` is selected for a facility, then the facility is considered selected.

DispTr, **FileTr**, **SaveTr**, **CheckTr** are flags which indicate whether a facility is selected or not.

Table 6-1. Various options offered by different facilities.

Facilities	Options	Description
DispTr	record	Display recorded trace; restricted to Σ_{record}
	check	Display trace post-restricted to Σ_{check}
	disp	Display trace post-restricted to Σ_{disp}
	off	**DispTr** is false
CheckTr	on	Check the trace post-restricted to Σ_{check}
	off	**CheckTr** is false
SaveTr	record	Save the recorded trace; restricted to Σ_{record}
	disp	Save the trace post-restricted to Σ_{disp}
	check	Save the trace post-restricted to Σ_{check}
	off	**SaveTr** is false
FileTr	xgrab	Use format displayable by XGRAB [RDB+87] system
	edge	Use format displayable by EDGE [] graph editor
	perftr	Use format for performance post-processing
	default	Use the default debugger format.
	off	**FileTr** is false

6.2 Information Requirements

The user can select one or more facilities. The flags **DispTr**, **FileTr**, **SaveTr** and **CheckTr** inform the debugger what facilities are selected. Note that all the facilities are not available in every debugging mode. Table 6-2 shows the availability of the facilities in each debugging mode. The user can only select from the facilities that are available in a given mode.

Table 6-2. Availability of various facilities in different modes.

Current Mode		Facilities Utilizing Event Trace				Interactive Facility
		`DispTr`	`CheckTr`	`FileTr`	`SaveTr`	
Run Modes (On-the-fly)	Replay Run	X	X	X		X
	Restricted Recording	X	X	X	X	X
	Full Recording	X	X	X	X	X
	Recording Off					X
	Performance Tracing				X	
	Debugging Off					
Postmortem Modes	From File	X	X		X	
	From Records	X	X	X		

The facilities provided by the debugger task need action-specific information described in Table 6-3. Note that the action specific information needed by the instrumentation is maintained in the action (Section 5.1.2). The information needed by the facilities is also kept inside the action[1]. This information becomes accessible to the facilities whenever they have the action id (Section 4.2.1). The flags **`ChkrBpOn`** and **`DispBpOn`** indicate whether the action has been selected for checking/displaying purposes. These flags deter-

[1] This organization will have to change in the distributed implementation.

mine the membership of sets Σ_{check} and Σ_{disp}, respectively. The information needed for post-restricting the event trace to actions of Σ_{disp} and Σ_{check} is kept in `DispRestrict` and `ChkrRestrict`, respectively. Input dependences of the action in the elaborated graph are maintained in `InpArcs`. This is a list of ids of those actions from which data was received by the action. The list of `SortNode`s helps the debugger task in sorting the incoming trace of events.

Table 6-3. Action specific information needed by the facilities.

Field	Type	Usage
`ChkrBpOn`	Boolean	checking, post-restriction
`DispBpOn`	Boolean	displaying, post-restriction
`disp`	`DispRestrict`	post-restriction
`vclk`	`ChkrRestrict`	post-restriction
`InpArcs`	List of `UcActionId`	displaying, filing
`SortNodeList`	List of `SortNode`	Sorting

6.3 Topological Sorting

The debugger task loops around looking for news arriving from the instrumentation (Section 4.3.2). News about the execution events of actions arrive in no particular order. The checking, filing, or displaying of an event must be delayed until all the events that were causally before the event have arrived. This is done by topologically sorting the arriving events using the information in their trace event records. Note that each news item about the execution event of an action contains the event record.

The sorting facility determines that a given event is sorted if all of its predecessors are sorted. Note that the predecessor information inside an event record is in the form of event ids (Section 4.4.3). Therefore, the sorting facility has to ascertain from a given id of an event, whether the event is sorted or not. This means that the sorting status of an event should be readily available given the event's id. For this purpose, each event of an action is represented by a **SortNode** that contains the sorting status. See Figure 6-2. The node is placed in the **SortNodeList** (Table 6-3) maintained inside the action. The list allows the sorting facility to map each event id to its corresponding **SortNode** and obtain the relevant sorting information.

In order to map a given event id, the facility uses the id of the action (available inside the event id; see Figure 4-12) to access the **SortNodeList** for the action. It then uses the execution count of the event, to search the list for the corresponding **SortNode**. If the node exists, it is returned. Otherwise, a new **SortNode** is created (with arrived and sorted flags initialized to false). The new node is inserted in the list, and, then, returned.

```
typedef struct SortNode *SortNode;

struct SortNode {
      EventId        eid;     // corresponding event
      Boolean        arrived; // has the event arrived?
      Boolean        sorted;  // is it sorted?
      List           fpsnl;   // list of flow-pred sort nodes
      List           spsnl;   // list of shared-pred sort nodes
      int            fscnt;   // #flow successors seen
      int            sscnt;   // #shrd successors seen
      ChkrRestrict   vclk;    // For checker restriction
      DispRestrict   disp;    // For display restriction
};
```

Figure 6-2. Sorting information for each event.

On receiving news about the execution event of an action, the sorting facility uses the id of the arriving event to obtain its corresponding **SortNode** as explained above. The facility sets the arrived flag of the **SortNode** to “true”. It, then, initializes the information inside this **SortNode** with the information contained in the event record that arrived with the news item. This involves translating the predecessor lists of the event record (which are in terms of event ids) into predecessor lists of sort nodes. That is, each event id in the predecessor list of the event record, is mapped to its corresponding **SortNode**. The objective is to make the sorting status of the predecessors readily available to the sorting facility. Once this is done, the arrived **SortNode** is inserted in the **`WaitList`**. This list contains sort nodes of all those events that have arrived and are waiting to be sorted (**`arrived`** flag is true, and **`sorted`** flag is false).

```
void  sortTrace()
{
      foreach SortNode sn in WaitList
           if (areAllPredSorted(sn)) {
                delete sn from WaitList;
                sn->sorted = true;
                doPostRestriction(sn);
                provideFacilities(sn);
                delPredsIfNoSucc(sn);
           }
      endfor;
}
```

The sorting algorithm picks up a **`SortNode`** from the **`WaitList`**, and checks if all the predecessors of the event (represented by the node) have been sorted. This is done by looking at the **`sorted`** flag of the predecessor sort nodes. If they are all sorted, then the node is deleted from the **`WaitList`**, and is marked as sorted. The event information inside the sorted node is then used for post-restriction as explained below. Then, the node is given to the facilities requested by the user. After that each predecessor sort node is accessed and the count of successors seen by the predecessor is incremented. The predecessor node is deleted if no other successors are expected by it. The algorithm, then, picks up the next node (if any) from the **`WaitList`**, and repeats the above process.

Note that the algorithm sketched above can be made more efficient by adding additional information to **`SortNode`** as explained in [KiZe93].

6.3.1 Support for Post-Restriction

Information needed for post-restriction is maintained with each action of Σ_{record} along with other action-specific information. See Table 6-3. There is corresponding information inside the **`SortNode`** of each event. See Figure 6-2. Note that the information needed for checker post-restriction is separate from that needed for display post-restriction. This is so because there can be any relationship between Σ_{check} and Σ_{disp}. Consequently, restriction to actions of Σ_{check} (Section 6.5.2) takes place separately from restriction to actions of Σ_{disp} (explained below). However, the process of post-restriction is quite similar. It is a three step process that takes place after an event has been sorted. It uses the **SortNode** of the newly sorted event:

```
void doPostRestriction(SortNode sn)
{
      initRestrictInfoFromUc(sn);
      updateRestirctInfoFromPred(sn);
      updateUcRestrictInfo(sn);
}
```

1. Initialize the post-restriction information inside the **SortNode** using the information of its corresponding action.

Note that a **SortNode** contains the id of its corresponding event. The event id makes available the id of the corresponding action, and hence the post-restriction information inside that action. This information is then copied inside the sort node.

2. Update the post-restriction information inside the **SortNode** using corresponding information from all of its predecessor nodes.

Note that each predecessor is already sorted, and, thus, has its own post-restriction information. The information inside the **SortNode** is brought up to date with respect to all of its predecessors'.

3. Update the post-restriction information inside the action using the sorted event's information.

The post-restriction information inside the **SortNode** was more current than the information inside its corresponding action. So, information inside the action is brought up to date using the information inside the **SortNode**. This is necessary because this information will be later copied inside a **SortNode** when the next event occurrence of the action is sorted.

Need for Maintaining Information with Actions

In order to enable post-restriction, later event occurrences of an action in Σ_{record} - $\Sigma_{restrict}$, must be informed about the predecessor events known to the action through its earlier occurrences. For this purpose, post-restriction information is maintained inside each action. This is a list of its currently known predecessor events. The information allows the i-th event occurrence of an action u in Σ_{record} - $\Sigma_{restrict}$ to forward the predecessor information of event u_i to the successors of its later event occurrences; u_{i+1}, u_{i+2}, ...

To see why it is necessary to maintain current post-restriction information with each action, consider the situation depicted in Figure 6-3. The predecessor information available with u_i has to be carried over to its later occurrence; u_{i+1}, so that it can eventually be forwarded to v_j.[2] The information maintained with each u in Σ_{record} - $\Sigma_{restrict}$ allows this when it is copied inside the `SortNode`. When step (3) of the above process is run for u_i, the post-restriction information with u will have the predecessor id x_k. When step (1) is performed for u_{i+1}, the post-restriction information with `SortNode` of u_{i+1} will get x_k from u. Eventually, when step (2) is performed for v_j , its sort node will get the id x_k from the sort node of u_{i+1}.

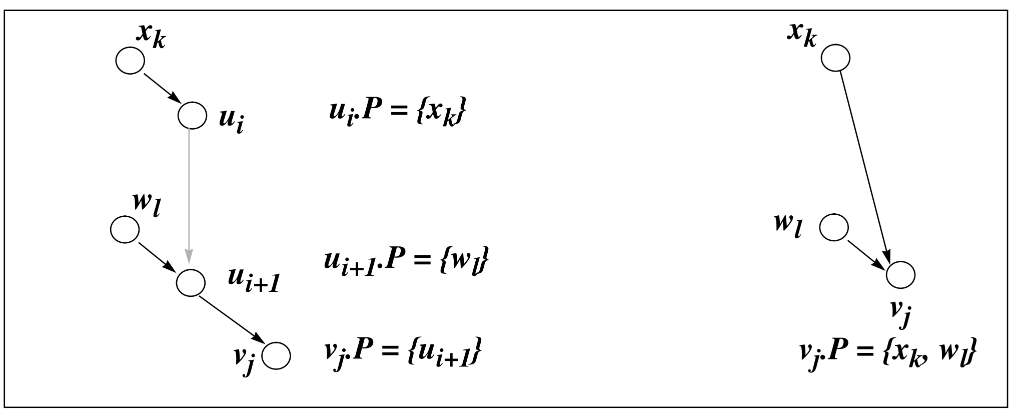


Figure 6-3. Need for maintaining restriction information with actions.

[2] Note that $u_i \notin u_{i+1}.P$. The precedence of u_i to u_{i+1} is only indicated by the execution count stored in their event records.

Note that the information kept inside the action performs a function similar to the one performed by `CurEvRec` during the recording by the instrumentation (Section 5.1).

Display Post-restriction

The partial order execution of DoBTS call graph of Figure 1-4 is shown in Figure 4-13. Its post-restriction to events of **Solve** and **Mult** actions is shown in the event graph of Figure 6-5.

Let $\boldsymbol{u.dr.P}$ denote the set of predecessor event ids kept inside the `DispRestrict` information for each action $\boldsymbol{u}$ in Σ_{record}. Similarly, let $\boldsymbol{u_i.dr.P}$ denote the set of predecessor event ids kept inside the `DispRestrict` information of the `SortNode` for each sorted event $\boldsymbol{u_i}$. The process of post-restriction for each sorted event $\boldsymbol{u_i}$ is given in Table 6-4.

Table 6-4. Display post restriction.[a]

Steps		Post-restriction	
1	$initDispFromAction(\boldsymbol{u_i}) \equiv$	$\boldsymbol{u_i.dr.P} :=\ \boldsymbol{u.dr.P}$	
2	$updateDispFromPred(\boldsymbol{u_i}) \equiv$ $\forall \boldsymbol{v_j} \in \boldsymbol{u_i.P}$	$\boldsymbol{u_i.dr.P} :=\ \boldsymbol{u_i.dr.P} \cup_e \{\boldsymbol{v_j}\}$	if $\boldsymbol{v} \in \Sigma_{\text{disp}}$
		$\boldsymbol{u_i.dr.P} :=\ \boldsymbol{u_i.dr.P} \cup_e \boldsymbol{v_j.dr.P}$	if $\boldsymbol{v} \notin \Sigma_{\text{disp}}$
3	$copyDispToAction(\boldsymbol{u_i}) \equiv$	$\boldsymbol{u.dr.P} :=\ \boldsymbol{u_i.dr.P}$	

a. In the set union $\cup_e$, if there are two events of the same action, then only the most recent event of the two is a member of the set union.

6.3.2 Deletion of Information no Longer Needed

The `SortNode` of an event can be deleted when the count of successors seen by a sorted event is equal to the count of successors expected by it. The expected count of successors is the actual number of successors recorded by the instrumentation. This is available inside the event record (Section 4.4.3).

The number of successors seen by a newly sorted event is initially zero. This is incremented each time a successor is sorted. On getting sorted, an event increments the successor count of each of its predecessors. This is done by accessing the sort node of each of its predecessors and incrementing the count of successors seen by the predecessor which is stored there. On incrementing, if the successor count seen by a predecessor is found to be equal to the number of successors expected by it, then this means that the predecessor's **`SortNode`** is no longer needed. That is, there would be no other successors looking for this node and it can, thus, be safely deleted. At this time, the event record itself can be deleted if the user has not requested that it be saved for a later replay or postmortem analysis.

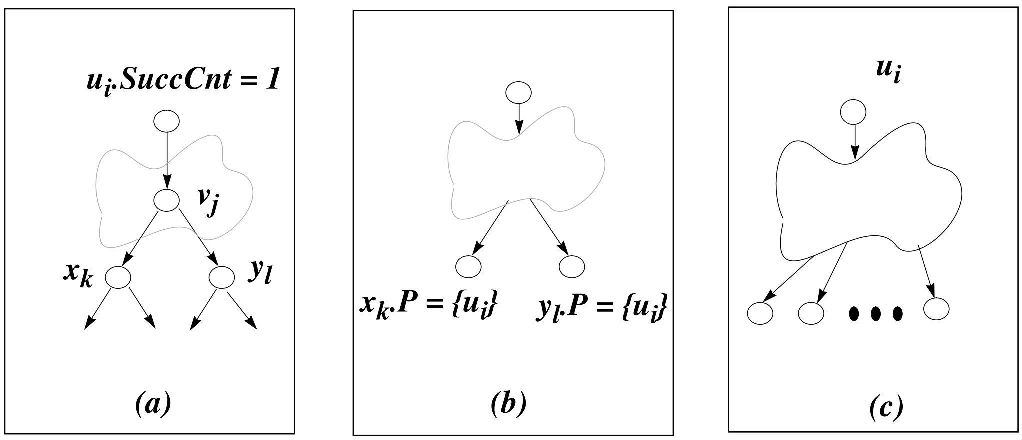


Figure 6-4. Successor counts in the restricted recording mode.

Ability to delete a sort node when it is no longer needed by any of the successors depends upon the accuracy of the successor count. During the full-recording mode and the replay run of the program, the exact count of successors is known. Therefore, sort information is deleted in these modes as soon as it is no longer needed.

However, during the restricted recording mode, the exact count of successors is often not known as shown in Figure 6-4(a). Suppose the instrumentation is recording the event occurrences of actions $\boldsymbol{u}$, $\boldsymbol{x}$ and $\boldsymbol{y}$, and is not recording the occurrences of action $\boldsymbol{v}$. The instrumentation would record the successor count of the $\boldsymbol{i}$-th execution of $\boldsymbol{u}$ as "1" because event id $\boldsymbol{u_i}$ is sent to only one successor, $\boldsymbol{v_j}$. As event occurrences of $\boldsymbol{v}$ are not being recorded, the id of the predecessor (namely, u_i) received by v_j is forwarded to its successors x_k and y_l. Now, both x_k and y_l know that u_i is their predecessor. But, the count of successor with u_i is only one.

The situation can get still more complicated. Suppose actions x and y are not selected for recording. Then, the instrumentation would forward their predecessor id (namely, u_i) to their successors. This id can, thus, go on propagating. As shown in Figure 6-4(c), the successor count of u_i will not be final until its id lands in the predecessor lists of those actions all of which are being recorded. Either we can delay the sorting until that time, which can potentially be until the end of the execution in which case no facilities would be available until the end of the execution. Or, we can delay the deletion of the sort information until after the execution has terminated. We take the later approach. So, in the restricted recording mode, we delay the deletion of sort information until the end of the execution.

6.4 Display Facility

The display facility maintains three graphical representations:

1. An event history graph which shows the ordering relations of an event with respect to other events.

2. An elaborated graph which shows the dependence relations between the actions introduced at runtime, and

3. A program graph which shows the relations between templates specified by the programmer.

Figure 6-5 shows the three graphs restricted to actions and events of templates **Solve** and **Mult** of the DoBTS call graph of Figure 1-4.

The display facility shows mappings between events, actions, and templates on these graphs. The mappings connect the three graphs together. These mappings can be viewed statically when the execution has finished or is stopped at a breakpoint. Or, they can be viewed dynamically as the events arrive during the execution or in a postmortem fashion. The dynamic view shows the progress of the execution and is often termed animation.

Our implementation currently displays these graphs textually. It thus uses the same internal structures for display purposes as are used for other purposes. A proper GUI interface may require separate structures that would be linked with the internal debugger structures. Note that the user has the option to dump the graphs to a file, and then display them using XGRAB graph browsing system (Table 6-1). Work on a proper GUI interface is currently in progress (Section 8.3.1).

6.4.1 Display of Mappings

The user may view these mappings by selecting an action, a template or an event.

By Selecting an Event

If an event is selected, the display would highlight its corresponding action and template. For example, if we select event $m^3{}_2$ in Figure 6-5(c), then action m^3 is highlighted in the elaborated graph and its corresponding template **Mult** is highlighted in the program graph.

When an event is selected, information in its event record becomes available. The information contains action id that implements mapping from an event to its corresponding action (Section 4.4.3). The id of the action stored in the record for (say) event $m^3{}_2$ is as a pointer to the struct for its corresponding action m^3. The struct contains information about the location of the action node in the elaborated graph. Using this information the action node is highlighted.

The mapping from an action to its corresponding template is implemented by keeping the template UID in the action struct (Section 4.4.2). The lookup for the template UID in the symbol table yields the corresponding tem-

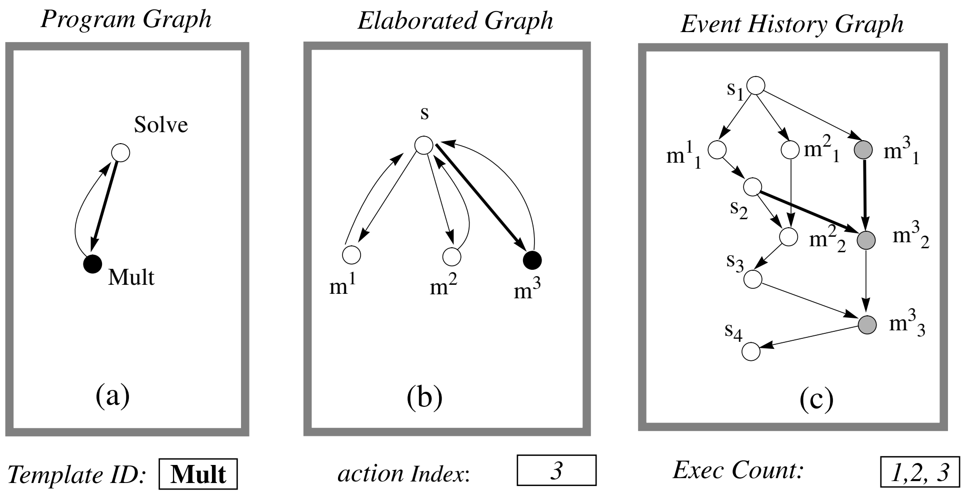


Figure 6-5. Mappings between Program, Elaborated and Event graph.

plate symbol. Information contained in the data structure defining the symbol allows the facility to identify the node ***Mult*** in the program graph.

By selecting an action

If an action is selected, the display would highlight its corresponding template and event(s). For example, if action $\boldsymbol{m^3}$ is selected in Figure 6-5(b), then events $\boldsymbol{m^3{}_1}$, $\boldsymbol{m^3{}_2}$, and $\boldsymbol{m^3{}_3}$ are highlighted in the event history graph, and template **Mult** is highlighted in the program graph.

The relationship from an action to its corresponding events is one-to-many. This is implemented by storing the list of event records in the action (Section 4.4.3). When an action is selected, the debugging information stored in its action struct becomes available. This allows highlighting of each event of the action.

The mapping from the action to its corresponding template is given by the UID stored in the action. As explained above, this is used to highlight the template node in the program graph.

By selecting a template

If a template is selected, the display would highlight its corresponding actions and events. For example, if template ***Mult*** is selected in the program graph of Figure 6-5, then the display would highlight actions m^1, m^2, and m^3 in the elaborated graph. The display would also highlight all the events for these actions i.e. events $m^3{}_1$, $m^3{}_2$, $m^3{}_3$, $m^2{}_1$, $m^2{}_2$, and $m^1{}_1$.

There is one-to-many relationship from a template to its actions. These actions can be nested in various contexts. Locating all of them, therefore, requires searching the entire dynamic instance tree. Note that the debugger keeps a handle to the root of the tree to allow for this kind of searching. The search for any action with the given template UID involves searching the entire dynamic instance tree. Once an action is identified, its corresponding event records also become available. These records can, then, be displayed as explained above.

6.4.2 Displaying Progress of Execution (Animation)

The arrival of a sorted event introduces new relationships or highlights existing ones between the nodes of the event graph, elaborated graph and the template graph. The process of displaying these relationships, as they are introduced or highlighted, provides a dynamic view of the progress of execution.

The event history graph scrolls forward as nodes corresponding to the newly sorted events are created in the graph and their orderings with respect to the existing nodes of the graph are determined. Simultaneous mappings of these events and their orderings on the elaborated graph provides animation. Figure 6-6 shows this process frame by frame.

The process involves repeating the following steps for each newly sorted event.

1. Create an event node corresponding to the event in the event history graph.

   Establish the mapping between the event and the newly created node.

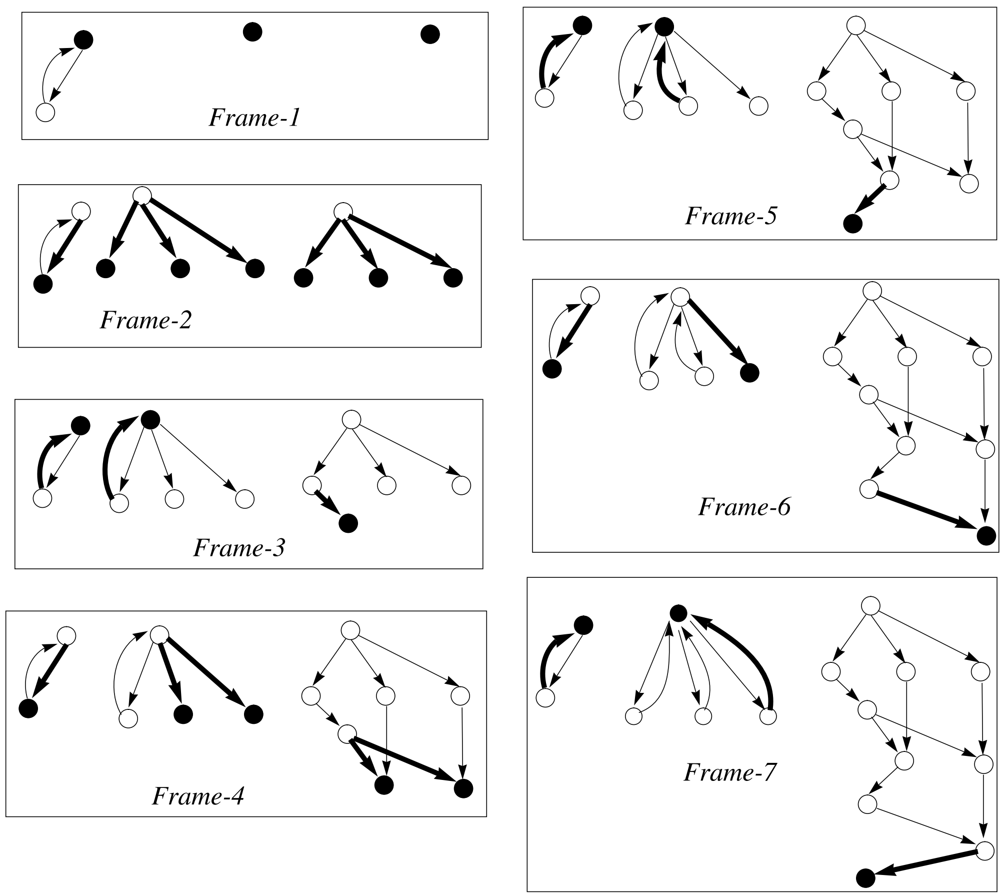


Figure 6-6. Displaying the progress of execution (animation).

2. Display the predecessor orderings of the event.

For each predecessor of the new event, the corresponding node is identified in the event history graph. Note that sorting ensures that such a node already exists. An arc is introduced from this predecessor event node to the new event node. This arc is highlighted. After the orderings with all the predecessors have, thus, been displayed, the new event node is highlighted.

3. Identify the action node in the elaborated graph corresponding to the event.

Create the action node if it does not already exist. This is the current action node.

4. Highlight the dependences of the current action node that correspond to the predecessor orderings.

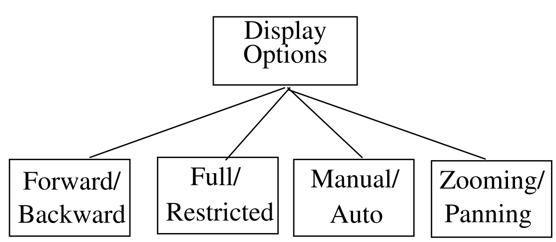
For each predecessor of the newly sorted event, the corresponding action node is identified. Sorting again ensures that the action node is already present in the elaborated graph. A dependence arc is introduced between the predecessor action node and the current action node if it does not already exist. If it exists, then it is simply highlighted. After all the dependences corresponding to the predecessor events have been highlighted, then the current action node is highlighted.

5. Highlight the mappings to the templates of the program graph.

As each action is highlighted, its corresponding template is highlighted. Similarly, as each dependence in the elaborated graph is highlighted, its corresponding dependence in the program graph is highlighted. Note that template UID's associated with each action allow us to identify and display these mappings.

This completes the display of the progress of execution with the arrival of a new event. Note that a proper GUI interface is being developed (Section 8.3.1) that will support options like the ones shown in Figure 6-7.

Figure 6-7. User options for the visualizations.



6.5 Checker Facility

Our checker is able to detect immediate, transitive and concurrent relationships between event occurrences of actions selected for checking. That is, we consider the relationships in the partial order $(V, \Sigma, <, \mu)/\Sigma_{record} / \Sigma_{check}$. Note that an event may become immediately related in this partial order due to post-restriction, even though it was transitively related in the recorded partial order $(V, \Sigma, <, \mu)/\Sigma_{record}$.

Note, also, that events of partial order $(V, \Sigma, <, \mu)$ may become immediately related in $(V, \Sigma, <, \mu)/\Sigma_{record}$. The immediate predecessor relationships of this partial order can be checked at runtime during the execution of actions at user breakpoints (Section 7.4.2).

6.5.1 Checking Relationships

The checker detects various relationships during the sorting of the events when post-restriction to Σ_{check} is taking place. The record for an event contains the list of immediate predecessors. Sorting of an event ensures that all the predecessors have causally arrived and have already been sorted.

Checking of immediate relationships is simple because the record for each event contains a list of its immediate predecessors.

Checking of transitive relationships of an event can either be done by recursively searching each predecessor of the event, or by timestamping each event with a vector clock. Searching the predecessors in a breadth-first or a depth first manner may require the traversal of entire causal history (state) of the event which is often expensive [FoZw90]. On the other hand, vector clocks provide a brief summary of the causal history. Each element of a vector clock corresponds to an action and identifies the most recent predecessor event known through the clock [Mat89].

We decided to use the vector clocks because they not only help in establishing ordering relationship between events, but are also helpful in establishing concurrent relationships.

Vector clocks can either be maintained during the execution of actions, or they can be maintained during sorting of the events by the checker as explained below. We take the latter approach for several reasons:

1. It allows separation of the checker concerns from those of the recording and instrumentation concerns.

2. It avoids runtime overhead.

Maintaining vector clocks during sorting avoids the computation and communication overhead incurred when the clocks are maintained during the execution of actions.

3. Events are sent to the checker and other facilities anyway for detecting of concurrent relationships, displays, and/or filing.

As events are sent irrespective of whether the vector clocks are maintained during the execution or during the sorting, it is, therefore, more efficient to maintain the vector clocks with the checker. Note that events must also be sent in case the user requests a view of the unexpected behavior. As mentioned in Section 1.3.5, it is not enough for the checker to simply state whether a relationship holds or not. It should also be able to provide more information about the actual execution behavior.

The downside for maintaining the vector clock with the checker is that it may delay the triggering of breakpoints. But, this happens in any case for conditions that can not be checked locally [WaG91], [ReSc94].

6.5.2 Post-restriction and Vector Clock

Post-restriction to actions of Σ_{check} is carried out using the same vector clocks that are being maintained for the checker. This involves the following steps:

1. Associate a vector clock with each action.

Each action in Σ_{record} maintains current information about its most recent predecessor events. This information is restricted to the events of Σ_{check} and is thus maintained in a vector clock of size $|\Sigma_{check}|$.

Let $\boldsymbol{u.V}$ be the vector clock maintained for an action $\boldsymbol{u} \in \Sigma_{\text{record}}$. Thus, the `ChkrRestrict` information seen in Table 6-3 and in Figure 6-2 actually refers to a vector clock. Each slot of this vector corresponds to an action in Σ_{check}, and contains the execution count of that action. Note that an event is identified by the action id and the execution count. Thus, if the slot corresponding to action id $\boldsymbol{u}$ in the vector contains the execution count $\boldsymbol{i}$, then the most recent event of $\boldsymbol{u}$ that is known through this vector is $\boldsymbol{u_i}$.

The reason for maintaining vector clocks for actions of Σ_{check} is that their relationships is being checked by the model checker. However, the reason for maintaining vector clocks for actions of Σ_{record} - Σ_{check} is to help in post-restriction as explained in Section 6.3.1.

2. After each event occurrence of an action, increment the slot for the action in the action's vector clock.

The events seen by the sorting facility are the executions of actions of Σ_{record}. Whenever an event $\boldsymbol{u_i}$ gets sorted, $\boldsymbol{u}$'s vector clock is incremented only if $\boldsymbol{u}$ is a member of Σ_{check}.

If $\boldsymbol{u} \in \Sigma_{\text{check}}$, then its event occurrences are recognized by the checker. Therefore, the slot for $\boldsymbol{u}$ in its vector clock is incremented i.e. $\boldsymbol{u.V}[u] := \boldsymbol{u.V}[u] + 1$. However, if $u \notin \Sigma_{\text{check}}$, then its occurrences are not being recognized. There is no slot for $\boldsymbol{u}$ in $\boldsymbol{u.V}$. So, there is no change in the current value of the vector clock $\boldsymbol{u.V}$.

3. Time-stamp each event occurrence of the action with the current vector clock of the action.

Whenever an event $\boldsymbol{u_i}$ is sorted, it is time-stamped with the current value of $\boldsymbol{u.V}$. This is done by copying the vector $\boldsymbol{u.V}$ inside its corresponding `SortNode`. Let this time-stamp be $\boldsymbol{u_i.V}$.

4. Update the event's timestamp using the timestamp of each of its predecessors.

Note that each predecessor is already sorted, and, hence, its vector timestamp is available. So, $\forall\, \boldsymbol{v_j} \in \boldsymbol{u_i.P}$, we do:

$$updateVclk(\boldsymbol{u_i.V}, \boldsymbol{v_j.V}) \equiv$$

$$\forall\, \boldsymbol{w} \in \Sigma_{\text{check}}: \boldsymbol{u_i.V}[w] < \boldsymbol{v_j.V}[w] :: \boldsymbol{u_i.V}[w] = \boldsymbol{v_j.V}[\boldsymbol{w}]$$

5. Use event's timestamp to update its action's vector clock.

After the update in (4), the event's timestamp, $\boldsymbol{u_i.V}$, contains the most recent information about the event's causal predecessors. This information is, then, used to bring the vector clock of its corresponding action, $\boldsymbol{u.V}$, up to date by performing $updateVclk(\boldsymbol{u.V}, \boldsymbol{u_i.V})$. Note that the value of the vector clock, $\boldsymbol{u.V}$, resulting from this update will be later used to time-stamp the next event occurrence of the action.

6.5.3 Checker Commands

Currently, the unified debugger supports two types of commands. One checks predecessor relationships and the other concurrent relationships between occurrences of actions. Each command issued to the checker is assigned a reference id and is entered in a reference list **`CmdRefList`**. As explained later, this list is used in translation of each command into an appropriate breakpoint before the execution is run. The reference id is also used in deletion of the breakpoints and their associated commands.

Checking immediate and/or transitive predecessors

If $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ are the action identifiers supplied by the user, then the command to check if event occurrences of $\boldsymbol{u}$ are preceded by an event of $\boldsymbol{v}$, *and* an event of $\boldsymbol{w}$ is:

`check pred` $\boldsymbol{u}$**`{`**$\boldsymbol{v}$**`,`** $\boldsymbol{w}$**`}`**

That is, $\forall \boldsymbol{u_i} \in \mathrm{V} : \exists\, \boldsymbol{v_j}, \boldsymbol{w_k} \in \mathrm{V} :: \boldsymbol{v_j} <^{c} \boldsymbol{u_i} \wedge \boldsymbol{w_k} <^{c} \boldsymbol{u_i}$.

The command is thus of the form:

`check pred` *<action_id>* `'{'` *<action_id_list>* `'}'`

Each ***<action_id>*** is composed of a template identifier and an index. Syntax for the debugger commands is given in Appendix B. Two commands for the same action result in the *or*-ing of the checking. Note that a firing rule is a set of subsets of input dependences. So, to construct a firing rule like $\{\{(\boldsymbol{u}, \boldsymbol{v}), (\boldsymbol{v}, \boldsymbol{v}), (w, v)\}, \{(\boldsymbol{w}, \boldsymbol{v}), (\boldsymbol{x}, \boldsymbol{v})\}\}$ the user has to give two check predecessor commands.

`check pred` ***v*****`{`*****u*****`,`** ***v*****`,`** ***w*****`}`**, and **`check pred`** ***v*****`{`*****w*****`,`** ***x*****`}`**

The result of these two commands is $\forall \boldsymbol{v_i} \in \mathrm{V} ::$

$$(\exists\, \boldsymbol{v_j}, \boldsymbol{u_m}, \boldsymbol{w_k} \in \mathrm{V} :: \boldsymbol{v_j} <^{c} \boldsymbol{v_i} \wedge \boldsymbol{u_m} <^{c} \boldsymbol{v_i} \wedge \boldsymbol{w_k} <^{c} \boldsymbol{v_i}) \vee$$

$$(\exists\, \boldsymbol{w_k}, \boldsymbol{x_l} \in \mathrm{V} :: \boldsymbol{w_k} <^{c} \boldsymbol{v_i} \wedge \boldsymbol{x_l} <^{c} \boldsymbol{v_i})$$

Checking concurrent relationship

The checker command to see if event occurrences of a set of actions were concurrent is:

`check parallel '('`*<action_id_list>* **`')'`**

`check parallel (`***u, v, w*****`)`** detects if there were any events of *u*, *v* and *w* that occurred concurrently. That is, $\exists\, u_i, v_j, w_k \in \mathrm{V} :: u_i \parallel v_j \parallel w_k$.

6.5.4 Command Translation

The size of the vector clock used for checking depends upon the number of actions being checked i.e. $|\Sigma_{\mathrm{check}}|$. The number of such actions is not known until the start of the execution because a user may add or delete the commands issued to the checker. So, each command that is issued to the checker is simply inserted in the **`CmdRefList`**. The commands in this list are translated just before the start of the execution. The translation generates a **`ChkNode`** for each action that is mentioned in a command. The list of these nodes, **`ChkList,`** now represents the set Σ_{check}. The size of the vector clock, i.e. $|\Sigma_{check}|$, is the number of check nodes in this list. The check node for an action is used for the following purposes:

```
struct ChkNode {
      TmpltUID uid;
      Index index;      // index of the action
      UcActionId id;    // pointer to the inst struct
      Vclk vc;          // Vector clock for inst
      ChkId cid;        // checker id; slot in vector
      List chkcmds;     // commands to be checked
};
```

1. It maintains the current vector clock for the action.

2. It determines the mapping between the **`<action_id>`** and the check id.

The check id of an action determines the slot in the vector clock reserved for the action. Note that during checking, the actions are identified by this id.

3. The node keeps the list of commands that need to be evaluated by the checker for each occurrence event of the action. Note that each action specified in the command is represented by the check-id.

4. The node allows the insertion of a checker breakpoint in actions that would be later created or re-run.

Because of their lazy creation (Section 4.2.1) and the dynamism of the CODE 2 programs, UC actions may not exist when the checker command is issued or translated. The insertion of a checker breakpoint for an action has to be delayed until when it is created or re-run. The check node keeps the **`TmpltUID`** and the **`Index`** which was obtained from the **`<action_id>`** given on the command line by the user. The mapping of this identifier to the **`UcActionId`** is established later when the action is created or re-run. Whenever a new action is created, its UID and index is used to search the **`ChkList`** for a **`ChkNode`** with the same UID and Index. If there exists such a node, then the action id is noted inside the **`ChkNode`** for later reference. The breakpoint is inserted in the action by turning the **`ChkBpkon`** flag inside the action (Table 6-3).

After post-restriction, each sorted event is given to the facilities selected by the user. Recall that the event information is now represented by a **`SortNode`**, and contains the up to date **`ChkrRestrict`** information (Section 6.3). As explained above this information is actually a vector clock for the event. If the event is of an action of Σ_{check}, then there is a corresponding **`ChkNode`** containing the list of commands that need to be checked for the action. The relationships specified by these commands are checked using the vector clock for the event. Note that the current vector clocks of all the actions with which the relationship has to be checked are available in **`ChkList`**. The predecessor and concurrent relationships between actions are checked in the usual manner [ReSc94], [GaW92].

Chapter 7. Interactive Facility

The interactive facility establishes cooperation between the instrumentation inserted inside each action, and the debugger task. This also requires cooperation from the runtime system. The facility allows the user to interactively control the execution of actions and to query their local state at various breakpoints set by the user.

7.1 Controlling the Execution

Interactive control is available at the scope of the program and at the scope of an action. A given command is interpreted in the context of the currently selected scope (Section 4.5). If the currently selected scope is that of the whole program, then the commands are considered global. Otherwise, they are applied to the currently selected scope of the action.

Running the program under the control of a traditional debugger like *dbx* typically involves the use of signals and **`ptrace`** utilities. In response to a run command, such a debugger would usually execute the object file of the program with some **`execv`** type of a (Unix) command in a child process. The debugger would maintain the current state of all processes being run by the application. It would manage their creation and deletion and would catch all the signals generated by the processes. Querying the state of each process would involve switching from the context of the debugger process to the context of the desired process and then interpreting the commands. Re-running would involve exiting the child process and doing another **`execv`** of the object file.

Running the program under the control of the unified debugger is, however, much simpler in our implementation because we deal with the actions and not processes. The template routine of an action executes like a thread in the name space of other actions[3]. The actions cooperate with the debugger

[3] This is true for all actions in the Sequent runtime system [New93]. It is true for those actions that have been mapped to the same machine by the distributed runtime system [Vok94].

task in controlling the execution. Control is exercised by enqueuing and dequeuing the actions from the ready and the stop queues.

7.1.1 Controlling the Execution of a UC Action

There are three commands to control the execution of an action: **`stop`**, **`cont`**, and **`next`**. A user may give a command to an action by selecting the action and, then, directly issuing the command. Or, the user can set a breakpoint that would conditionally trigger the issuance of the command during the execution of the action (Section 7.2).

The command issued to an action is stored inside the struct for that action. The default setting for the stored command is **`cont`**. It allows the execution of the action to proceed normally. If the stored command is either **`next`** or **`stop`**, then this acts as a breakpoint during the execution of the action as described below. A command can cause a transition between the stopped or not-stopped state of the action as shown in Figure 7-1.

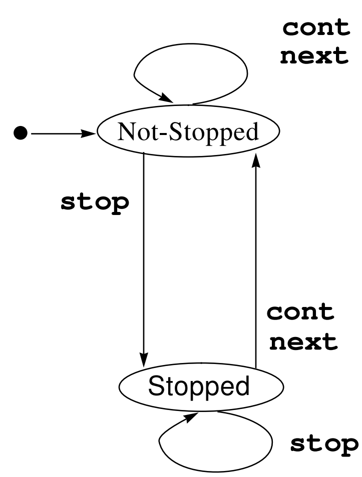


Figure 7-1. State transitions for a UC action.

Stop Command

The command to **`stop`** the execution of an action is stored inside the action. During the execution of the action, the stored command acts as a request to the instrumentation for stopping the action before the start of its sequential computation. Stopping is implemented by changing the state to stopped, and enqueuing the action in the **`StopQ`**. Simultaneously, the debugger task is informed about the stopping of the action. The debugger then forwards the news to the user.

Cont Command

The **`cont`** command continues the execution of an action from where ever it was stopped. It removes the stopped action from the **`StopQ`**, and readies it for execution by putting it on the **`ReadyQ`**. The state of the action is not changed at this time. Eventually, a worker task picks the action from the ready queue and starts executing it. When the action begins its execution, the instrumentation notes that the state of the action is stopped. It now changes the state of the action to not-stopped and ensures that the action resumes its execution from the point indicated by the field **`CurAt`** (Table 5-2) inside the action.

Next Command

The **`next`** command issued to an action causes all the successors of the action to stop before they begin their sequential computation.

If the action is stopped, then the command removes the stopped action from the **`StopQ`** and readies it for execution by putting it on the **`ReadyQ.`** Later, when the action is executing and is about to send data on its output ports, the **`next`** command stored inside the action would inform the instrumentation to stop the successors. This is done by appending the **`stop`** directive to the data being sent. When the successors are about to begin their sequential computation they note the directive and stop themselves as explained in Section 5.5.

7.1.2 Global Control of the Execution

Commands issued when the currently selected scope is "program" (Section 4.5) are considered global. Figure 7-1 shows global commands that cause transition between different global states of the computations.

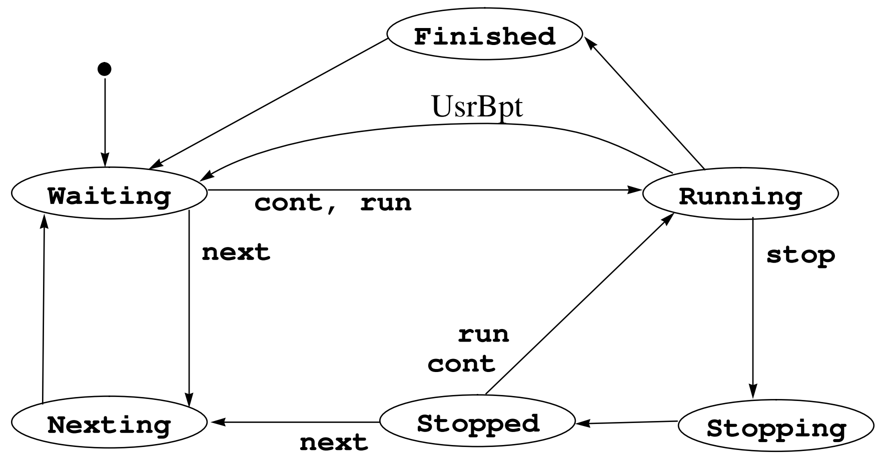


Figure 7-2. Global state transitions in response to global commands.

Running

The user starts a new execution with a **`run`** command. The command causes a transition to **`Running`** state when the start node specified in the CODE 2 program graph is readied for execution by putting it on the **`ReadyQ`**. **`Running`** state indicates that one or more actions are executing or are ready to execute.

Execution of the start node readies for execution all actions to which it sends data. Their execution would, in turn, ready other actions for execution and so on. Eventually, the termination node specified in the CODE 2 program graph would complete its execution.This would cause a transition to **`Finished`** state. The state indicates that the program has run to completion.

Continuing

Issuance of a **`cont`** command readies for execution all the stopped actions by flushing them from the **`StopQ`** to the **`ReadyQ`**. This causes a transition to **`Running`** state.

Stopping

The user can stop all the actions that are running by issuing a **`stop`** command. This causes a transition to **`Stopping`** state. The state informs the instrumentation to stop any action that is about to begin its sequential computation. Eventually, all the actions that were executing or were ready to run would stop. This would cause a transition to `Stopped` state as explained below.

Nexting

The user can ask the successors of all the actions that can run to stop by issuing a `next` command. The command causes all the stopped actions to be readied for execution by flushing the `StopQ` to the `ReadyQ`. It also causes a transition to `Nexting` state. The state informs the instrumentation to ask every successor of the currently executing action to `stop`. This is done by appending the `stop` command with each data that is sent out on an output port. Eventually, all the executing actions would stop causing a transition to `Waiting` state.

Waiting or Stopped

`Waiting` and `Stopped` states indicate that there are no actions that are ready to run (the ready queue is empty), and there are no actions that are currently being executed by any of the workers (workers are idle). This implies that an action can not become ready for execution except through a user command. It is, then, safe for the user to query the state of the actions and issue other global commands.

Soon after the debugger comes up, the global state becomes `Waiting`. This happens because `ReadyQ` is empty and all the workers are idle. Issuing the run command can then start the execution. The stopping of an action on a set breakpoint may also cause a transition to `Waiting` state. This can happen if all the other actions that can currently execute are dependent upon the action. So, the action's stopping will cause them to starve for data and eventually stop. For instance, in Figure 4-13, if **Dist** action stops as a result of a user breakpoint, then all actions would eventually become idle because they depended on **Dist** for the data that could have triggered their execution, causing an eventual transition to `Waiting` state.

Recall that workers loop around looking for actions to run from the **`ReadyQ`**. A worker that fails in a certain number of attempts to find an action to run from the ready queue, declares itself to be idle. This activates a typical termination detection algorithm [ChMi88] for detecting whether all the workers are idle or not. If it turns out that all the workers are idle, then the global state is changed to **`Stopped`** if the current state is **`Stopping`**. Otherwise, it is changed to **`Waiting`**.

Our shared memory implementation of the termination detection is simple. When the application starts, each worker task is assigned a logical id. The worker with the largest id is designated as the decider. Each worker only checks for the workers with lower ids. A worker task idles itself if it is idle and all the workers whose ids are less than its id, are also idle. Eventually, the decider worker (one with the maximum id), discovers that all other workers are idle. So, it idles itself and declares the state to be **`Waiting`** or **`Stopped`**.

7.1.3 Rerunning the Execution

Rerunning the program involves preparation of the objects created in the previous execution by the program and the debugger for a new run. Some of these objects will be reused in the new run. Therefore, they need to be re-initialized. Others objects are no longer useful and the storage allocated to them must be recovered. Re-initialization and storage recovery of objects created by the debugger is simple because the debugger knows what objects it created and what are their initial values. However, re-initialization and storage recovery of the objects created by the program is difficult because the debugger has no knowledge about those objects.

Objects Created by the Debugger

Rerunning the program in a given mode requires mode-specific preparations of the objects created by the debugger. If the program is being run in the replay mode, then the debugger ensures that event records are available from which the execution would be replayed. It also determines whether the events were recorded in the full-recording mode. Note that in order to ensure a proper replay, the trace should have been recorded in the full recording mode.

If the program is being rerun in the recording mode, then the instrumentation will be generating new event records. So, previously recorded events of the actions are deleted. This is done by accessing the debugging information kept for each action and deleting the event records in the list kept there. Action-specific information maintained by the instrumentation (Section 5.1.2) and the facilities (Section 6.2) is re-initialized. Some of the previous user selections like recording options are, however, retained.

Display, checker and user breakpoints are reevaluated to reflect any deletions or insertions. All the check nodes are deleted and the checker commands are re-translated to generate a new list of check nodes (Section 6.5). Options selected for various facilities (Table 6-1) are retained subject to the availability of the facilities (Table 6-2). **`ReadyQ`**, **`StopQ`** and **`NewsQ`** are emptied.

Objects Created by the Program

When a program is rerun by a traditional debugger like *dbx*, recovering the storage allocated in its previous execution is relatively simple. It only involves **`exit`**-ing the child process that **`exec`**-d the program object file when the program was run. However, in our implementation there is no child process with which the program was executed, that could be exited to get rid of the storage allocated by the program.

The debugger task and the worker tasks are simply light-weight threads. Actions that execute are objects (Figure 4-5) that hang around along with all their local state. Their local state consists of objects that were allocated by the specifications provided by the user. The only objects of the program that the debugger knows about are the actions created during the execution (Section 4.4.2). Local state of the actions is entirely managed by the program. There are two ways of recovering the storage allocated by the program:

1. Exit the debugger along with the program and then re-run the program. Exiting will recover all the storage allocated by the program.

2. Re-run the program without exiting the debugger. The debugger must recover all the storage allocated by the program.

Exiting and reentering the program each time the execution is run is not a user-friendly option. Furthermore, exiting the program with the debugger would also destroy the debugger objects that need to be reused. This would necessitate dumping of the contents of those objects to a file before the debugger is exited, and then their re-construction when the debugger is reentered. Note that event records use pointers to action structs for action ids. In the replay mode, these records are used for enforcing replay. Exiting and then reentering meant that we will have to reconstruct the pointers when new actions are created as the program is rerun.

Hence, we decided to reuse the existing actions and recover the local state allocated by the program.[4] Fortunately, CODE 2 runtime system manages all the objects that are allocated on the heap, and keeps a table of the space allocated to them. The table was accessible to the debugger. So, a complete clean up of all the objects allocated by the action was possible.

7.2 User Breakpoints

The interactive facility allows the user to set various breakpoints for a specified action. The instrumentation arranges to break the execution of the action at these points, and execute the breakpoint commands conditionally or unconditionally, as specified. Setting of a breakpoint requires the following information:

1. The id of the action for which the breakpoint is being set.

---

[4] The ability to reconstruct the debugger state is also quite useful. It will be needed when we implement the starting of another session (or replaying) from a file.

The action id may be explicitly supplied in commands like **`stop in <action_id>`**. Otherwise in commands like **`stop if`** *condition*, it is assumed to be the currently selected scope (Section 4.5).

2. The point where the break should occur during the execution of the action.

Currently, the user has option to break the execution of an action at two points; before the serial computation of a UC or after the computation of a UC. The option to break at other points within the sequential computation is fairly well understood in the context of sequential debugging and was, thus, not implemented. Its implementation is tied with the provision of stepping facility as described in Section 8.2.1.

3. The commands that need to be executed on hitting the breakpoint.

The user can specify the commands that would be executed at a breakpoint. For example, **`when at`** *bpt* **`{`***cmd1***`;`** *cmd2***`; ...}`** allows the user to specify the commands to be executed at the specified breakpoint. These commands are only allowed to access and change the local state of the action. Their syntax is given in Appendix B.

4. The condition under which to execute the specified commands.

This is optional. The breakpoint commands can, thus, be conditionally or unconditionally evaluated. The specified condition is on the local state of the action and may specify the immediate predecessors expected for the given execution. Note that the immediate predecessors of an action can be checked easily during the execution of the action because the current event record (Section 5.1.2) maintains this information.

The commands to set user breakpoints are entered in a list. The list helps in insertion, and deletion of breakpoints. Each command in the list is translated into a **`UsrBreakPt`**. The breakpoint contains a reference to the command that produced it. It contains the point where the break should occur, and the list of commands that are to be executed. There is an AST corresponding to each command in this list. As the commands may have to be conditionally evaluated, the AST includes any specified conditions.

```
struct UsrBreakPt {
      UsrBpCmd  usrref;   // Reference to bp command
      Where     point;    // point of insertion
      List      commands; // actions to be taken
      UcActionId id;      // back-end id of action
      TmpltUID  uid;      // front-end id
      Index     index;    // index of action
};
```

The breakpoint contains the **`<action_id>`** given by the user. Whenever a new action is created, the breakpoint list is consulted to see if there is a breakpoint specified for this action. If there is, then the action is informed about the breakpoint by setting **`UsrBpOn`** (Section 5.5.2) to true. The id of the action is noted for future reference. The list of breakpoints is, therefore, useful in setting and un-setting of breakpoints.

7.3 Evaluation of User Commands

The evaluation of a user command can simultaneously occur in different contexts. It may be the case that when the debugger frontend is evaluating a user command, several UC actions could be at their respective breakpoints evaluating their breakpoint commands. At the same time, the checker could be busy evaluating its commands. The routine that evaluates user commands should, therefore, be simultaneously callable from several contexts. This is enabled by allocating a separate stack for each context from which the evaluation routine is invoked. The context information is associated with the stack. Evaluation of context sensitive commands like **`print <expr>`**, then, takes place depending upon the context in which the evaluation is being done.

- **Evaluation by the Debugger**

The debugger maintains information about the currently selected scope (Section 4.5). Conditions and expressions specified in a command are evaluated in the context of the currently selected scope. For example, if the debugger is evaluating the command **`print <expr>`**, then the names given in the **`<expr>`** are evaluated in the context of the currently selected action.

- **Evaluation at a User Breakpoint**

During the execution of an action, commands evaluated at a user breakpoint refer to the scope of the action. For example, the condition specified in **`print in`** ***Solve*** **`if <bool_expr>`** refers to the scope of ***Solve***. So, when the breakpoint is hit during the execution of ***Solve***, the evaluation routine is given the base address of Solve where the local state is saved. Evaluation of **`<bool_expr>`** takes place in this context.

- **Evaluation by the Checker**

The checker evaluates the commands during sorting. The only information that is available to it is the event information. The commands are thus interpreted in the scope of the event that has just been sorted. The user may need to trigger other commands as a result of this checking. However, this was not implemented.

7.4 Querying the State of Actions

It is safe to query the local state of an action when it is stopped. It is also safe to query any action when the global state is stopped or waiting.

7.4.1 Local State of an action

Given the name of a symbol, the symbol table provides information about the type of the symbol, its UID and a reference to the symbol of its enclosing block (Section 4.4.1). This information is needed for printing the value of a given symbol or testing conditions on its state.

In a CODE 2 program, the programmer can declare objects of various types within the scope of an action. The available types are **`array`**, **`struct`**, **`int`**, **`real`**, and **`char`**.

CODE 2 arrays are dynamic. A two dimensional array is actually an array of arrays. A multi-dimensional array is an array of an array, and so on. The size of the array is runtime dependent. In order to print out the contents of a given array, we need the size of the array and the index into it. The CODE 2 runtime library provides a routine that returns the size of the array given its address. Symbol table information provides the type of the array. Therefore, the contents of a given array can be printed. The debugger not only allows the user to see the contents of an array, but also provides the ability to index into a particular element of the array and see its contents. Note that the content of a given index of an array may be a single object or a (multi-dimensional) array.

An input port of an action is a queue of a given type. The user can look at the data values currently waiting in a given input port. The debugger obtains the address of the port, and uses some runtime knowledge about the queue for printing out the values waiting inside the queue.

7.4.2 Debugger Defined Objects

In order to help the user in debugging, access to three objects available to the debugger was found to be useful. These are **`ExecCnt`** (Table 5-2), **`NodeIndex`** (Figure 4-5), and **`Pred`** (Figure 4-12). These objects are used by the debugger for supporting various facilities. The user may employ these symbols in the conditions and expressions specified for various commands. For example,

`stop in` <u>*Mult*</u> **`if NodeIndex = 3`**

will cause the stopping of that action of ***Mult*** whose index is 3. Similarly,

`stop in` <u>*Mult*</u> **`if ExecCnt = 3`**

will stop any action of ***Mult*** that executes for the third time. Similarly, the user can insert a breakpoint that tests for the immediate predecessors:

`stop in` <u>*Gath*</u> **`if Pred (`**<u>*Mult*</u>**`)`**

would cause ***Gath*** to stop if its immediate predecessor is an event of ***Mult***. Also,

```
print Pred
```

would print all the predecessors of the currently selected event.

7.4.3 Addresses of Objects

A traditional debugger like *dbx* computes the addresses of an object by adding the offset of the object to the base address of its enclosing block. It obtains the offset from the symbol table information that is provided by the compiler. The debugger also maintains a runtime stack of active frames which can provide the base address of each frame. During the computation of the address of a given object, the debugger maintains the base address of the currently selected object/frame from which the offset of the object is to be computed. The offset is added to the base address to generate the address of the object.

The unified debugger takes a different approach for computing the addresses. Instead of providing the offset of a symbol, the compiler provides the UID of the symbol (Section 4.4.1). It also provides a routine for computing the address of an object whose corresponding symbol's UID is given along with the parent UID and the address of the symbol's enclosing block. A part of the code generated by the compiler is shown in Figure 7-3.

```
void *_c2_getSymAddr(UID, pUID, Data)
int UID, pUID;
void *Data;
{
   switch(pUID) {
    case 31:
      switch(UID) {
        case -1111:
          return (void *) ((struct _c2_nv31 *) Data)->B_FR_S;
          break;

        case -1121:
          return (void *) &((struct _c2_nv31 *) Data)->b;
          break;
    :
    :
}
```

Figure 7-3. Compiler generated routine for obtaining addresses.

Chapter 8. Future Work

The research reported in this dissertation includes both the definition of the unified model of concurrent debugging and also a feasibility demonstration implementation of the unified model. It has become clear that the unified model does provide a framework for a complete and comprehensive debugging system for parallel and concurrent programming. The future work which would be needed to realize this comprehensive debugging system for concurrent and parallel software includes the following:

1. Enhancements and extensions to the current unified model for concurrent and parallel debugging. The extensions include:

* Dynamic vector clocks.
* Hierarchical representation of the events.
* Hierarchical replay.

2. Interfaces to other modes of program validation:

* Integration with an interpreter-based sequential debugger.
* Interface to or integration with static analysis and symbolic analysis systems.

3. Enhancements of the implementation making it more effective for the user and more user friendly. These enhancements include:

* A graphical user interface for the debugger and a better integration with the CODE2 programming interface.
* An implementation for a distributed execution environment.
* Implementation of rollback and recovery.
* Buffering of logical event traces for performance measurements.
* A stepping facility for sequential code of the computation actions.

* More extensive facilities for model checking.

* Definition of hierarchical replay.

* Reconstruction of a debugging session from a file.

Another extension which would be significant, although not a part of the unified model of debugging itself, is to map programs written in sequential and parallel text string languages into the action/relationship programming model so that the unified debugger can be applied to them as well.

The following sections explain each of the above points.

8.1 Enhancements to the Unified Model

Extensions to the unified model for concurrent debugging will allow the debugger to support dynamic vector clocks, hierarchical representation of events, and optimization of the amount of recording necessary for replay.

8.1.1 Dynamic Vector Clocks

The use of vector clocks typically assumes that the number of slots in a vector is fixed, and the mapping of each slot to its corresponding executable entity (process/thread) is already available [GaW92], [Mat89], [ReSc94], [Fid89]. However, when the number of executable entities vary or when the entities migrate, the assumption is no longer valid. For example, processes may get created (or deleted) at runtime, or they may migrate. The interpretation of the slots in the vector clock must reflect these changes.

The unified debugger will need the dynamic vector clocks for implementing race detection (Section 8.3.5) and optimizing the recording overhead for replay (Section 8.3.6). Note that the debugger maps each slot of the vector clock to an action. As UC actions are instances of templates, and their number varies at runtime (Section 4.2.1), the vector clocks should be able to adjust to increasing number of slots (i.e. actions).

Note that the checker facility of the unified debugger currently uses vector clocks for checking relationships among occurrences of actions

(Section 6.5). It fixes the size of the vector before the execution is run to the number of actions whose relationships are being checked. The dynamic vector clocks will also make possible the deletion and addition of commands given to the checker during the execution. Note that the checker fixes the number of actions before the execution, and addition and deletion of commands can vary the number of actions, and hence the interpretation of the slots of the vector.

Dynamic vector clocks will also allow the checker facility to check relationships between the templates. Currently, the facility, checks relationships among execution occurrences of UC actions. It is unable to check relationships among the templates, because the number of instances of a template can vary at runtime.

The problem of increasing number of slots in the vector clock can be addressed by (say) maintaining a current count of slots with each vector clock object. The creation of a new instance, will cause a new slot to be added to the vector. The routine that evaluates the checker commands and interprets the slots will be informed of the new mapping.

8.1.2 Hierarchical Representation of Events

CODE 2 graphs are hierarchical. The runtime hierarchical context of each UC action is available in the dynamic instance tree (Section 4.4.2). However, the events of these actions are not displayed in their hierarchical context because the notion of a graph event is not defined. In order to represent the events hierarchically, it is necessary to define the conditions that determine when the execution of a graph started and when it ended.

Note that there are explicit firing and routing rules that determine the start and the end of an execution of a computation action (Section 3.1.2). But, there are no such rules for a CODE 2 call graph. Once a CODE 2 call graph is instantiated, the actions inside the graph can execute independently of the graph whenever the firing rules of the actions are satisfied [New93].

There are several definitions which give semantics to the execution of a graph [Ho91], [ZeR91], [ReSc94]. The execution of a graph can also be considered as a sub-pomset [Pra86], and the *refinement* operator [Gis88] can, then, be used to map one pomset to another. Given an appropriate definition, it is possible to use the information currently available at runtime with the unified debugger to represent the events in their hierarchical contexts.

8.1.3 Hierarchical Replay

The storage requirements for the execution replay facility can be further reduced by only recording the event orderings of actions belonging to a set of selected subgraphs. Event orderings of actions that do not belong to this set of subgraphs will not be recorded. The subgraphs whose orderings are not being recorded will be considered as “holes”.

In the recording phase (Section 3.5), the instrumentation will record the contents of any communication that occurs across these holes. An event record will, then, contain the content of the message from a predecessor event if the predecessor belongs to a subgraph that is not selected. However, the record will only contain the id of the predecessor if the predecessor belongs to a selected subgraph. The replay phase will work as follows: An action will get the actual message sent by the predecessor during replay if the predecessor belongs to a selected subgraph. The action will, however, be given the pre-recorded message if the predecessor belongs to a subgraph that is not selected.

8.2 Interfaces to Other Systems

The capability of the unified debugger can be greatly enhanced by interfacing it with an interpreter-based sequential debugger and systems for static and symbolic analysis.

8.2.1 Interpreter and Stepping Facility

Advanced debuggers often use an interpreter to obviate the need to recompile each time a small modification is made to the user code [Sy83]. The interpreter can especially be useful in the CODE 2 runtime system for a network of heterogenous workstations where compiling code for different machines can be time consuming. An interpreter like UPS [Rus91] can be interfaced with the unified debugger to support small modifications made to the sequential computation of a UC action.

The interpreter could also provide a facility to step through the sequential computation. The interactive facility currently allows stopping at the start and at the end of the sequential computation in a UC action (Chapter 7). It supports **`next`**-ing from the begining of the sequential computation of a UC action to its end, and **`next`**-ing from the end of the sequential computation of a stopped action to the begining of the computation of all of its logical successors. The interactive facility currently does not support stepping through the sequential computation. Ability to step through the sequential code is however quite useful and can be provided with an interpreter. The interpreter will only be invoked only when the user requests single stepping of the sequential computation. Otherwise the execution will proceed through the computation as it does currently.

8.2.2 Static Analysis and Symbolic Analysis

Information generated during static analysis has been used to simplify the runtime information requirements for race detection and execution replay [MiC89]. It is, however, also possible to use the information collected at runtime to manage the complexity of static analysis facilities. Note that the static analysis of parallel programs for detecting access anomalies and deadlocks is intractable [Tay84]. The logical orderings of events actually exhibited by the execution at runtime can be used for simplifying static analysis because this information is not available at compile time. Similarly, this information can also be used for simplifying symbolic analysis [YoTa86].

8.3 Enhancements to the Current Implementation

The objective of the implmentation described in Chapter 4-Chapter 7 was to demonstrate that the unified model of concurrent debugging provides a framework that allows a single debugger to support all of the debugging facilities that have previously been defined separately. The current implementation is a prototype that provides the basic functionality needed for a demonstration of feasibility. In order to allow the user to fully exploit this functionality following enhancements in the short-term are foreseen.

8.3.1 A Graphical User Interface (GUI)

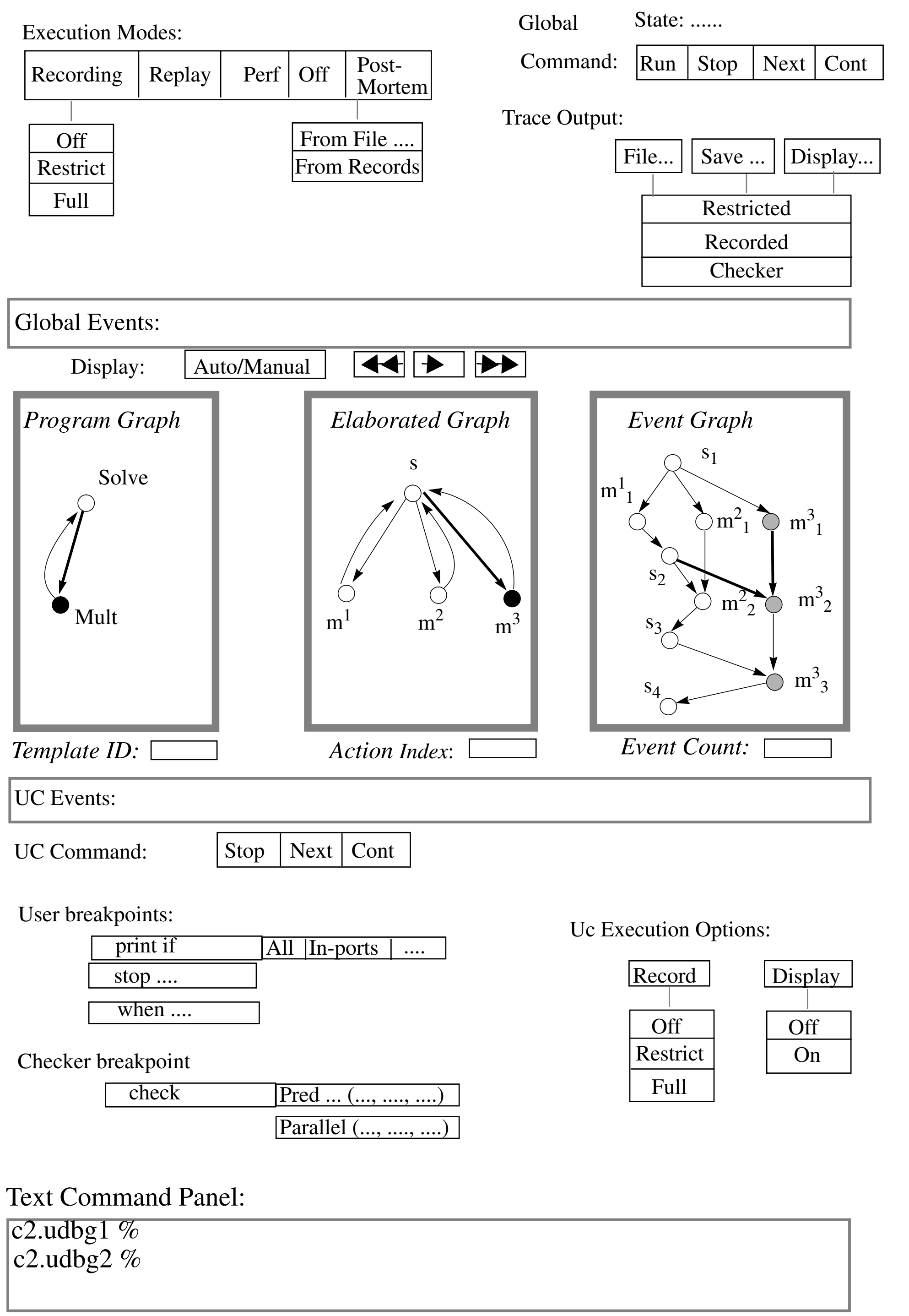
The implementation currently uses a modification of the textual interface provided by the Berkeley (BSD) ***dbx***. The interface is only capable of accepting textual input and providing a textual output. In order to view the different graphical representations produced by the debugger, the graphs are first dumped using one of the filing options described in Section 6.1.2. The dumped file can, then, be picked up and displayed with the XGRAB graphical editor [RDB+87].

Work on the next generation of CODE 2's graphical frontend is currently in progress. The plan includes the provision of a proper GUI environment for interactive debugging. In this interface, interactions between the user and the debugger would be based on the display of three graphs; the elaborated graph, program graph and the execution event graph (Section 6.4). The figure given on the next page shows some of the icons and displays planned for the interface. The user would be able to click on the interesting nodes of these graphs to see their execution behavior, and make various selections. The commands and selections described in Appendix B will be offered through icons and buttons.

8.3.2 Distributed Implementation

This dissertation describes the implementation of the unified model of concurrent debugging in the context of a shared memory machine. However, the abstractions used by the model are machine independent and are also applicable in a distributed environment.

CODE 2 environment is able to translate the programs into executables that can run on a network of workstations using PVM message passing primi-


Execution Modes:
Recording
Replay
Perf
Off
Post-Mortem
Off
Restrict
Full
From File ....
From Records
Global
State: ......
Command:
Run
Stop
Next
Cont
Trace Output:
File...
Save ...
Display...
Restricted
Recorded
Checker
Global Events:
Display:
Auto/Manual
Program Graph
Solve
Mult
Elaborated Graph
s
m1
m2
m3
Event Graph
s1
m11
m21
m31
s2
m22
m32
s3
s4
m33
Template ID:
Action Index:
Event Count:
UC Events:
UC Command:
Stop
Next
Cont
User breakpoints:
print if
All
In-ports
....
stop ....
when ....
Checker breakpoint
check
Pred ... (..., ..., ...)
Parallel (..., ..., ...)
Uc Execution Options:
Record
Off
Restrict
Full
Display
Off
On
Text Command Panel:
c2.udbg1 %
c2.udbg2 %

tives [Vok94]. Current implementation of the unified debugger for Sequent shared memory machine was designed keeping in view this distributed implementation of the CODE runtime system. For example, the decision to have a separate debugger task that would communicate with the worker tasks through **`NewsQ`** and **`StopQ`** (Section 4.3) was prompted by this consideration. Note that the debugger task could have been implemented in a manner similar to a template routine (Section 4.2.2) that is picked up by a worker task for periodic running.

The distributed implementation of the unified model has been designed and is being implemented. In this implementation, the debugger task and the worker tasks will be PVM tasks (processes). Interactions between the debugger task and the instrumentation described in Section 4.3, will be through the PVM messages. Global commands will be sent to each worker task. Whereas, commands that only apply to a particular action will be sent to that worker task to which the action has been mapped by the runtime system. In order to execute the debugger commands on a remote machine, there will be a remote debugger with the worker task executing on that machine. The worker task and the remote debugger will multiplex their execution as they do in the serial implementation of the debugger. Note that there is a serial implementation of the unified debugger for the CODE 2 runtime system for Sun4 workstations. The remote debugger will execute the commands sent to it. It will maintain the structures local to the machine like the **`ReadyQ`** and **`StopQ`**. It will also control the instrumentation inserted inside the UC actions that have been mapped to its machine by the runtime system.

The central debugger task will maintain the **`NewsQ`**. News from the instrumentation (Section 4.3.1) will be directly sent to the central debugger from the remote locations. The central debugger task will maintain a dynamic instance tree which will be a union of all the branches instantiated by the runtime system at various machines. It will also maintain a **`StopQ`** that will reflect the status of all the stop queues at remote locations.

The manner of identifying an action will change. Currently, the debugger identifies an action with a pointer (Section 4.2.1). In the distributed environment, this pointer may not be unique. So, the debugger will maintain another level of indirection between the full-pathnames with which an instance is identified by the user (Section 4.5), and the address of the struct with which the runtime system identifies the instance at a remote location.

8.3.3 Recovery and Roll-Back

The replay facility exactly replays an earlier execution using the ordering information recorded for that execution (Section 5.4). The information required for recovery and roll back of a concurrent execution [JhZw90] is contained in the information recorded for replay purposes. So, the capability provided by the replay facility can be extended to provide roll back and recovery. The orderings will, then, have to be recorded on a stable storage. Note, however, that the replay facility can only replay the execution from the beginning. During recovery and rollback it is not always convenient, or even feasible, to re-start the execution from the beginning each time there is a crash. In order to avoid this, the recovery mechanism can periodically checkpoint the state of the execution on stable storage. The execution can, then, be started from the last checkpoint.

In the CODE 2 environment, a facility for checkpointing is already available for process migration and load balancing purposes [Wag94]. This checkpointing facility can be used in conjunction with the replay facility to provide recovery and roll-back from the last checkpoint. The checkpointing and recording of orderings will, then, have to done on stable storage. Note that it is not necessary to checkpoint the entire state of each process. The recovery mechanism can benefit from approaches like [MiC89]. These approaches use data-flow analysis to limit the amount of information that must be recorded in order to replay an execution from some point in the past. This can greatly reduce the checkpointing overhead.

8.3.4 Extending Facilities for Model Checking

The checker facility currently provides checking of three types of relationships (Section 6.5). The implemented functionality is, however, also capable of supporting the checking of several other types of relationships. For example, linked predicates [MiC88] and path expressions [HsK90]. These predicates/models only require immediate ordering information which is currently available. Disjunctive predicates [GaW92] require maintenance of vector clocks which are also currently available. There are other relationships [ReSc94] which can also be supported by the implemented functionality.

8.3.5 Implementation of Race Detection

As explained in Section 3.4, the orderings $<^C$ are the transitive closure of the immediate orderings recorded by the debugger. The instrumentation described for race detection in Table 3-3 makes use of these orderings. These $<^C$ orderings are not directly available from the immediate orderings that are being recorded. These can, however, become available if the instrumentation maintains the vector clocks as described in Table 8-1. Note that $\omega.D.T$ is the vector timestamp of the last writer, and $\rho.D.T$ is the combined timestamp for all the readers of an object.

Table 8-1. Data-race detection instrumentation with vector clocks.

$u \in D$	u_i obtains access to D.		u_i releases access to D
	Race is detected if:	If race is detected	
$u \in R_D$	$\omega.D.T \not< u_i.T$[a]	$update$[b]$(u_i.T, \omega.D.T);$	$\rho.D := \rho.D \cup \{u_i\};$ $update(\rho.D.T, u_i.T)$
$u \in W_D$	$\omega.D.T \not< u_i.T$	$update(u_i.T, \omega.D.T);$	$\rho.D := \{\};$ $\omega.D := u_i;$ $\omega.D.T := u_i.T$
	$v_j \in \rho.D \wedge v_j.T[v] \not< u_i.T[v]$	$update(u_i.T, \rho.D.T)$	

a. If $w_k = \omega.D.T$, then $\omega.D.T \not< u_i.T$ *if* $\omega.D.T[w] \not< u_i.T[w]$

b. $update(T, T') \equiv \forall u: T'[u] > T[u] :: T[u] := T'[u]$

8.3.6 Optimizing the Replay Recording

The replay facility currently records all the flow-predecessor and shared predecessor orderings (Section 3.5). It is possible to optimize the amount of recording necessary for replay [Net93]. The unified model of concurrent debugging can provide further optimization by only recording the non-deterministic choices. But, this requires the provision of dynamic vector clocks (that can adjust according to increasing number of actions), and implementation of the race detection facility at runtime. The instrumentation will check the races as explained in Table 8-1, but will record the orderings as described in Table 3-3 only if a race is detected.

8.3.7 Implementation of Other Options

Section 6.1.2 describes several options that are offered by various debugging facilities. Most of these options have been implemented and can be selected by the user. However, the options to save and/or file the post-restricted event trace is not implemented. That is, the implementation can currently file/-save the events of the partial order which has been restricted to Σ_{record}, but is unable to file/save the events of the partial order that has been post-restricted to actions of Σ_{disp} or Σ_{check}. This remains to be implemented.

In the postmortem mode, facilities offered by the unified debugger utilize event records that were traced in an earlier execution (Table 6-2). The trace can be in a file or in the form of records saved in internal structures. The implementation currently provides postmortem utilization of trace from the records, but does not provide utilization from a trace file. That is, the loop from the trace file to records as depicted in Figure 4-2 is not implemented.

Dumping the state of a debugging session to a file and reconstructing it later can be quite useful to the user. Although, the debugger can "source" the commands saved in a file (Appendix B), it is unable to reconstruct the entire debugging session. This ability remains to be implemented.

The interactive facility provides two types of commands for setting user breakpoints; **`stop at....`** and **`when at....<cmd_list>.`** (Section 7.2). Other types of breakpoints can also be implemented with the current functionality. For example, breakpoints like **`once at....`** and **`watch at... <var_list>`**. The former is activated "once" the execution reaches the specified point, and is then reset. The latter monitors the changes to the variables at a given point and only informs the user when the value changes. Similarly, the ability to "assign" a different value to a variable at runtime when stopped at a breakpoint may also be quite useful to the user.

8.3.8 Buffering Requirements

A buffer maintained with each worker task can make the performance tracing mode more robust, and can make the evaluation of commands at a breakpoint more efficient.

Performance Tracing

The unified debugger provides a performance-tracing mode for generating the timings of the logical trace of the execution (Section 5.1.1). In this mode, the instrumentation saves the event records with timing information in the internal structures. No input/output is allowed during performance tracing because it may disrupt the timings that are being recorded. After the execution, the mode is switched to the postmortem mode. Then, the event records saved during the performance-tracing mode are dumped to a file in the **`perftr`** format mentioned in Table 6-1.

Saving all the event records in internal structures until the execution has run to completion can potentially cause the memory to run out. The problem can be avoided by implementing (say) a circular buffer with each worker task. Each worker task, after executing an action (Section 4.2.3), will put the execution event record in its buffer. The debugger task will, then, be responsible for periodically removing the event records from the buffers and dumping them to a file.

Breakpoint Evaluations

The interactive instrumentation (Section 5.5) can use the buffer maintained with each worker task for performance-tracing purposes. Note that the evaluation of commands at a user breakpoint requires a separate stack (Section 7.3). Currently, the implementation reserves a chunk of memory for the stack each time the breakpoint command evaluation routine is entered during the execution of an action. The allocation and de-allocation that takes place each time an action enters and exits the breakpoint evaluation routine is inefficient. If there is a buffer maintained with each worker task, then this buffer can be used as a stack during the evaluation of the breakpoint commands. Note that there is no conflict in this usage because the performance tracing mode and the modes in which interactive facility is available are mutually exclusive (Table 6-2).

Furthermore, the commands activated at a user breakpoint often generate information that has to be communicated to the debugger task. Currently, the result of these commands is printed on the standard output. This is unsatisfactory, because the information should be sorted and presented to the user in the context of the execution event of the action. The buffer allocated for each worker task can also serve the purpose of saving the partial results of command evaluations. It will be filled with the result of any breakpoint commands executed during the execution of actions. After the execution of the action, the contents of the buffer will be communicated to the debugger task.

8.4 Textual languages

Computation actions are naturally available in the CODE 2 graphical programming environment. These are not readily available in the textual parallel languages which add synchronization/communication to an existing sequential language. However, as explained in Appendix A, the computation actions can be extracted from the graphical representation of the program maintained by the compilers. The unified model of concurrent debugging can, then, be applied to these languages.

This requires that the mappings from the basic blocks of the graphical representation maintained by the compiler to the computation actions of the program graph needs to be formalized. Furthermore, the runtime should be able to map the instances of these computation actions to the templates defined in the program graph. This may involve extra work for the debugger if these instances are allocated on the stack by the runtime system. Note that in the CODE 2 runtime system, instances of a template are allocated on the heap and the mapping information is readily available to the debugger in the struct for each action (Section 4.2.1).

Chapter 9. Conclusions

This dissertation has defined and described a formal model of concurrent debugging in which the entire debugging process is specified in terms of program actions and executions of program actions. This unified model of parallel debugging places all of the approaches to debugging of parallel programs such as execution replay, race detection, model/predicate checking, execution history displays and animation, which are commonly formulated as disjoint facilities, in a single, uniform framework.

We have also developed a feasibility demonstration prototype of a debugger implementing this unified model of concurrent debugging in the context of the CODE 2.0 parallel programming system. This implementation demonstrates and validates the claims of integration of debugging facilities in a single framework. It is further the case that the unified model of debugging greatly simplifies the construction of a concurrent debugger. All of the capabilities previously regarded as separate for debugging of parallel programs, both in shared memory models of execution and distributed memory models of execution, have been given an implementation in this prototype.

The critical concept underlying the unification of concurrent debugging is the concept of separation of concerns. Separation of concerns leads to parallel programs which are formulated so that specifications for parallel considerations such as synchronization and communication are completely separated from the specification of sequential transformations on data. The model of concurrent debugging which we have specified here can be applied to any parallel programming language or system in which this condition is met. The CODE 2.0 parallel programming system is an example of such a programming system.

It is the case, however, that conventional text string parallel programming systems do not explicitly display such a structure unless separately analyzed to identify the computation actions. However, this analysis to identify computational actions and to separate specification of sequential computations from specification of communication and synchronization can be accomplished with the data gathered by the parallelizing compilers. Therefore the unified model of parallel debugging is, in fact, applicable to all parallel programs provided they have been pre-processed into an appropriate extended generalized data-flow graph where computational actions are cleanly specified and separated from communication and synchronization behavior. It is our belief that this representation of programs is in itself beneficial and its adaptation will lead to simpler and more effective parallel programming.

Appendix A. Computation Actions in a Textual Representation

The computation actions used by the unified model are naturally available in the CODE 2 graphical parallel programming environment. As explained below, the abstraction of computation actions can also be extracted from a parallel program written in a textual sequential language with extensions for synchronizations and communications. Note that the abstraction of computation actions permanently associates the synchronization statements of a concurrent program with its sequential text segments. It permanently associates a blocking synchronization with the sequential text that follows it, and permanently associates a signal synchronization with the sequential text that precedes it.

A.1 Extracting a Graphical Representation

A parallel program written in a sequential language with extensions for synchronization primitives may be considered as consisting of three types of statements; *blocking* synchronization*, signal* synchronization and *non-synchronization* statements*:*

- A *blocking* synchronization makes the execution progress of a process where it resides, dependent on the occurrence of one or more synchronizations elsewhere. Execution of the process suspends if the corresponding synchronization has not occurred. For examples, **`call evwait()`** in Cray Fortran, **`in()`** or **`read()`** in *Linda*, rendezvous in Ada, **`P`** operation on a semaphore, and so on. In distributed systems, the primitive is usually a **`receive()`** of message. Note that a non-blocking receive is actually a blocking receive at a lower level [Lam86].
- A *signal* synchronization is a non-blocking synchronization operation that does not change the execution state of the process in which it executes. It is independent of the execution state of any other process, but may be instrumental in changing the execution state of a dependent process.

Examples: **`call evpost()`** in Cray *Fortran*, **`out()`** in *Linda*, **`V`** operation on a semaphore, **`set()`** in *PPL*, and so on. Its equivalent in distributed systems is the **`send()`** of a message.

From a text containing *blocking* synchronization, *signal* synchronization and *non-synchronization* statements, *static analysis* techniques routinely extract a synchronization-control-flow graph [Tay83], [TO80], [BBC88], [CaS89], [McD89], [MiC89]. The graph typically contains three types of nodes and two types of arcs. Nodes represent blocking synchronization, signal synchronization, and control decision statements. While arcs represent inter-process synchronization dependences and intra-process control-flow dependences.

```
main () {
    :   /* workers = n */
    :   /* counter ct = n */
    post (ev_i : i = 1..n);
    while (1) {
        c_wait (ct);
        :   /* calculate work done */
        :
        if (work_done)
            exit ();
        post (ev_i : i = 1..n);
} }
                                        (a)
```

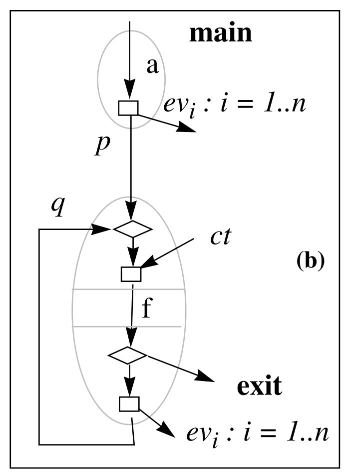


Figure A-1. (a) Process ***main*** (b) Synchronization-control-flow graph.

Figure A-1(a), and Figure A-2(a) show a textual parallel program written in a PPL like extension of *C*. Process ***main*** signals ***n*** workers to starts, and then waits on a synchronization counter ***ct***. On completion, each *worker* reports by incrementing ***ct***. When count ***ct*** reaches ***n***, ***main*** wakes up. It determines if more work is needed. If not then it exits, else it signals the ***workers*** again, and so on.

Figure A-1(b), and Figure A-2(b) show the synchronization control flow graphs extracted from their corresponding text. Note that in Figure A-

1(b), ***a**, **p**, **q*** and ***f*** are control dependences; ***exit*** and ***ev_i*** are signal synchronization dependences and ***ct*** is a blocking synchronization dependence. Similarly, in Figure A-2(b)*,* ***m*** and ***r*** are control dependences; ***ct*** is a signal synchronization and ***ev_i*** is a blocking synchronization dependence. Intra-process control arcs labeled by ***a***, ***w*** and ***f*** correspond to the sequential text containing non-synchronization statements.

```
worker i () {
    while (1) {
        wait (ev_i);
        :    /* do some work */
        :
        c_set (ct);
    }
}
```

(d)

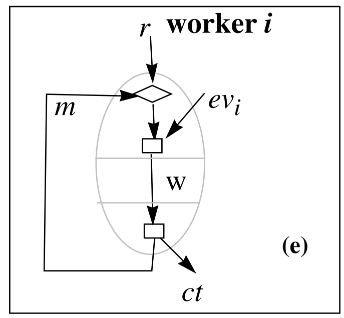


Figure A-2. (a) Worker *i* (b) Synchronization-control-flow graph

A.2 Extraction of Computation Actions

The abstraction of computation actions, shown by the dotted ovals in Figure A-1(b) and Figure A-2(b), permanently associates the synchronization statements with the sequential texts. It permanently associates a blocking synchronization with the sequential text that follows it, and associates a signal synchronization with the sequential text that precedes it. Thus, *blocking* synchronizations **wait** *(ev_i)* in Figure A-2, is associated with the text ***w*** that follows it, and blocking synchronization **c_wait (***ct)* in Figure A-1 is associated with the text ***f*** that follows it. Also, *signal* synchronizations **c_set(***ct*) is associated with the text ***w*** that precedes it, and signal synchronization **post** *(ev_i : i = 1..n)* is associated with texts ***a*** and ***f*** that precede it. Thus,

Def. A-1 A *computation action* is a block of a flow graph that:

[1] may contain internal control flow provided the internal control structures (loop, if-then-else, etc.) do not contain any synchronization statement;

[2] it may begin with a *blocking* synchronization, that must be the first statement of the block; and

[3] it may end with a *signal* synchronization, that must be the last statement of the block.

[4] it may contain more than one *blocking (signal)* synchronization provided they are all together at the start (end) of the block with no other intervening statement-types.

After the synchronization statements have been associated with their bordering sequential texts, we are left with the control flow decision nodes. The abstraction of firing rules subsumes these decision nodes as shown in Figure A-3(a) and (b). For example, in Figure A-3(b), input firing rule of ac-

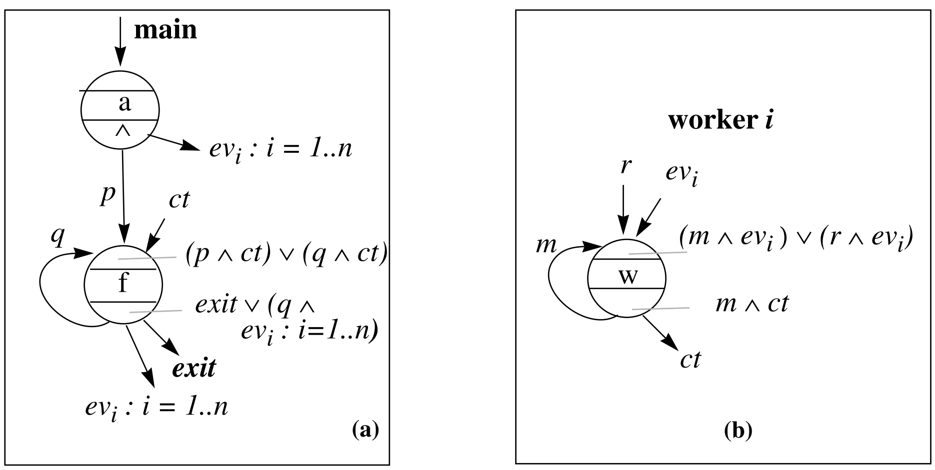
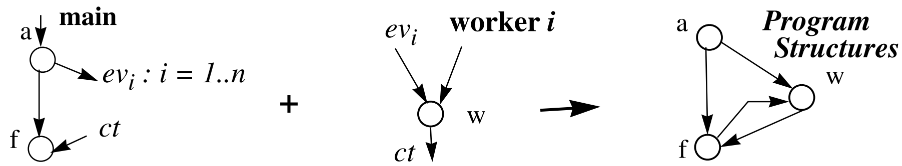


Figure A-3. (a) Computation actions *a* and *f*, and (b) *w*

tion ***w*** is $(m \wedge ev_i) \vee (r \wedge ev_i)$, and the output firing rule of w is $m \wedge ct$.

The set of *computation actions* Σ_P, then, characterizes the specified behavior of the textual program. Thus, in the above example, $\Sigma_P = \{a, w, f\}$; $F_P =$

$\{(a, w), (a, f), (w, f), (f, w)\}$; $O_P(a) = \{\{(a, w), (a, f)\}\}$; $O_P(f) = \{\{exit\}, \{(f, w)\}\}$; $I_P(w) = \{\{(a, w)\}, \{(f, w)\}\}$; $I_P(f) = \{\{(f, f), (w, f)\}\}$.



Note that *firing-rules* explicitly state the conditions that were implicit in the semantics of the synchronization primitives. Debuggers often keep these semantics implicit. In Figure A-3(b), *input firing rule* that initiates the execution of the computation block *"w"* has been stated as "$(m \wedge ev_i) \vee (r \wedge ev_i)$". It says that ***w*** may initiate its execution if one of the two disjuncts is true i.e. control reaches at ***m*** or ***r*** and $\boldsymbol{ev_i}$ has been posted. Similarly, the post-condition that follows the execution of ***w*** can be stated as a rule "$m \wedge ct$". It says that ***w*** signals by synchronizing the counter "*ct*" and transferring control to *"m"*.

Appendix B. The Unified Debugger for CODE 2

This appendix provides a brief over-view of the commands with which a user interfaces with the unified debugger. For details see the user manual[1].

B.1 Description of Commands

Table B-1 gives a brief description of the commands accepted by the unified debugger. Note that:

- Some of the commands are context-sensitive. The legal context for such commands is specified in the table. The debugger may assume a default context for a command if a proper context is not selected. For example, **`run`** command is always considered global, i.e. applying to the program scope, whereas **`cont`** will return with an error message if the currently selected scope is not a UC instance or the program scope.
- There are default aliases for some of the commands.
- Output of some of the commands can be redirected to a file.

Table B-1. Brief listing of the available commands.

Command	Alias	Redir-ectable	Legal Contexts	Description
`alias`			-	Define alias for a command
`check`			-	Check for parallel/predecessor relationship
`cont`	`c`		Pgm/UC	Continue the execution
`debug`			-	Internal command for maintenance
`delete`	`d`		-	Delete a user/checker/display breakpoint
`disp`			-	Select actions for display restrictions
`dump`		yes	Call /UC	Dump the local state of the Call (NI) /UC
`help`	`h`	yes	-	Print a help message
`list`	`l`	yes	Pgm/Call/UC	Provide a listing of instances/events

[1] Available as an internal document in the CODE project, CS Dept., UT Austin.

Table B-1. Brief listing of the available commands.

Command	Alias	Redir-ectable	Legal Contexts	Description
`list ‘e’`	`lse`	yes	Pgm/Call/UC/ event	List all events of the selected scope
`list ‘i’`	`lsi`	yes	Pgm/Call/ UC	List all instances of the selected scope.
`mode`		yes	-	Change/show the currently selected mode
`next`	`n`		Pgm/UC	Stop at aftcomp; or at befcomp of successors
`output`			-	Select trace display/filing/saving option
`print`	`p`	yes	Call/UC	Print result of the specified expression
`psym`		yes	-	Print info about the specified symbol
`quit`	`q`		-	Exit the debugger
`rcopt`			Pgm/Call/UC	Select from restrict/full/off recording option
`run`	`r`		-	Start a new run of the execution
`select`	`cd`		Pgm/Call/ UC/Event	Select the specified scope
`source`			-	Read and execute commands from the file
`stop...`	`st`		Pgm/UC	Stop the execution or set a stop breakpoint
`whatis`		yes	Call /UC	Print the type of the given symbol
`when...`			UC	Set a breakpoint for evaluating given cmds
`where`	`pwd`	yes	Pgm/Call/ UC/Event	Show the currently selected scope.
`which`		yes	Call /UC	What is the current context of the symbol
`unalias`			-	Remove an alias

B.2 Syntax of Commands

```
<cmd>       ::=   <mode_cmd>          |
                  <output_cmd>        |
                  <disp_cmd>          |
                  <rcopt_cmd>         |
                  <list_cmd>          |
                  <select_cmd>        |
                  <exec_cmd>          |
                  <alias_cmd>         |
                  <intrnl_cmd>        |
```

```
                  <del_cmd>          |
                  <query_cmd>        |
                  <usrbp_cmd>        |
                  <source_cmd>


<output_cmd> ::=  output {<savetr> | <filetr> | <disptr> }
<chk_cmd>    ::=  check pred <action_id> '{'<actionid_list>'}'
                  | check parall {<action_id_list>}
<disp_cmd>   ::=  disp <action_id_list>
<mode_cmd>   ::=  mode [ record [<rc_opt>] | replay |perftr |
                        dbgoff | postmortem [<from_opt>] ]
<rcopt_cmd>  ::=  rcopt [<rc_opt>]
<list_cmd>   ::=  lsi <list_opt> | lse <list_opt>
<select_cmd> ::=  where |
                  cd |
                  cd '.' '.' |
                  cd '/' [<scope_path>] |
                  cd <scope_path> |
                  cd : INT
<exec_cmd>   ::=  run | cont | stop | next
<alias_cmd>  ::=  alias NAME  NAME |
                  alias NAME STRING |
                  alias NAME '(' <name_list> ')' STRING |
                  alias NAME |
                  alias |
                  unalias NAME


<intrnl_cmd> ::=  debug [-] INT | psym
<del_cmd>    ::=  delete [check | disp] INT
                        default: user breakpoint
<query_cmd>  ::=  print <expr> |
                  dump |
<usrbp_cmd>  ::=  stop <usrbp_opt> |
                  when <usrbp_opt> '{' <cmd_list> '}'
<source_cmd> ::=  source <filename>

      <savetr>          ::=   record <rctype>
      <filetr>          ::=   ufile <filetype>
      <disptr>          ::=   disp <disptype>}
      <rctype>          ::=   [elabgph | evgph |bothgph | off]
                              default: bothgph
      <filetype>        ::=   [record |edge | xgrab|off| perftr]
                                    default:  record
      <disptype>        ::=   [restrict | record | check| off]
                                    default:  record
```

```
<rc_opt>           ::=   restrict | off | full
<from_opt>         ::=   record | ufile
<scope>            ::=   <tmplt_name> |
                         <tmplt_name> <index_list>
<action_id>        ::=   <scope>
                               (of a UC action)
<action_id_list>   ::    <action_id> |
                         <action_id> ',' <action_id_list>
<name_list>        ::=   NAME | NAME ',' <name_list>
<file_name>        ::=   STRING
<tmplt_name>       ::=   NAME | UID
                         (of a UC , NS or a Call node template)
<index_list>       ::=   '[' <int_list> ']'
<usrbp_opt>        ::=   [in <where_in>] [at <where_at>]
                         [if <bool_exp>]
<cmd_list>         ::=   <cmd> ';' |
                         <cmd> ';' <cmd_list>
<where_in>         ::=   <scope>
<where_at>         ::=   befcomp | aftcomp
<expr>             ::=
<bool_expr>        ::=
<int_list>         ::= INT | <int_list> INT
```

Keywords:

alias	aftcomp	and	at	befcomp
check	cont	debug	dbgoff	delete
disp	div	edge	ufile	full
help	if	in	list	mod
mode	next	nil	not	or
off	output	perftr	parall	postmortem
pred	print	psym	quit	record
replay	restrict	run	select	source
status	step	stop	rcopt	unalias
use	whatis	when	where	whereis
which	xgrab			

References

[BAS89] J. C. Browne, M. Azam, and S. Sobek, "A Unified Approach to Parallel Programming", *IEEE Software,* Jul '89.

[Bat89] Peter Bates, "Debugging Heterogeneous Distributed Systems using Event-Based Models of Behavior," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 24(1), pp. 11–22, January 1989.

[BH83] Bernd Bruegge and Peter Hibbard. Generalized path expressions: A high level debugging mechanism. *ACM SIGPLAN Notices*, 18(8):34–44, March 1983.

[BFM83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. Development of a debugger for a concurrent language. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, published in ACM SIGPLAN Notices*, 18(8):98–106, August 1983.

[BJN95] J. C. Browne, J. Dongarra, S. I. Hyder, K. Moore and P. Newton, "Experience with CODE and HeNCE in Visual Programming for Parallel Computing," To appear in *IEEE Parallel and Distributed Technology*, 1995

[BBC88] V. Balasundaram, D. Baumgartner, D. Callahan, K. Kennedy, and J. Subhlok, "PTOOL: A System for Static Analysis of Parallelism in Programs," Technical Report TR88-71, Rice University, Department of Computer Science, June 1988.

[CaS89] David Callahan and Jaspal Subhlok, "Static Analysis of Low-level Synchronization," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 24(1), pp. 100–111, January 1989.

[ChS91] Jong-Deok Choi and Janice M. Stone, “Balancing Runtime and Replay Costs in a Trace-and-Replay System,” *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, in ACM SIGPLAN Notices,* 26(12), pp. 26-35, Dec. 1991.

[ChMi88] K. Mani Chandy and J. Misra, “Parallel Program Design,” Addison-Wesley, 1988.

[CFH93] J. E. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, “The Ariadane Debugger: Scalable Application of Event Based Abstractions,” *ACM/ONR Workshop on Parallel and Distributed Debugging,* pp. 85-95, 1993.

[EGP89] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua, “Event Synchronization Analysis for Debugging Parallel Programs,” In *Proceedings of Supercomputing ’89*, pp. 580–588, Reno NV, November 1989.

[Fid89] C. J. Fidge, “Partial Orders for Parallel Debugging,” *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices* 24(1), Jan. 1989.

[FLM89] Robert J. Fowler, Thomas J. LeBlanc, and John M. Mellor-Crummey, “An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-scale Multiprocessors,” *Proceedings of the ACM Workshop on Parallel and Distributed Debugging, ACM SIGPLAN NOTICES*, 24(1), pp. 163–173, January 1989.

[For89] Alessandro Forin. Debugging of heterogeneous parallel systems. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):130–140, January 1989.

[FoZw90] J. Fowler and W. Zwaenpoel, “Causal Distributed Breakpoints,” proc. 10th International Conference on Distributed Computing, France, pp.

134-41, May 1990

[Gai88] Haim Gaifman, "Modeling Concurrency by Partial Orders and Nonlinear Transition Systems," *Linear Time, Branching Time, and Partial Order in Logics and Models of Concurrency, LNCS*, (354), pp. 467–488, May 1988.

[GaW92] V. K. Garg and B. Waldecker, "Detection of Unstable Predicates in Distributed Programs," *Proc. 12th Conference of the Foundations of Software Technology and Theoretical Computer Science, R. SShyamasundra (ed.), Springer-Verlag*, LNCS 652, pp. 253-264, 1992.

[HWB93] S. I. Hyder, J. F. Werth, J. C. Browne, "A Unified Model for Concurrent Debugging," Proc. Intl. Conf. on Parallel Processing, pp. II-58-67, Chicago, July 1993.

[HHK85] Paul K. Harter, Jr., Dennis M. Heimbigner, and Roger King, "IDD: An Interactive Distributed Debugger," Technical report, University of Colorado, 1985.

[HoC87] Alfred A. Hough and Janice E. Cuny, "Belvedere: Prototype of a Pattern-oriented Debugger for Highly Parallel Computation," In *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 735–738, University Park PA, 1987.

[HoC90] Alfred A. Hough and Janice E. Cuny, "Perspective views: A Technique for Enhancing Parallel Program Visualization," In *Proceedings of 1990 International Conference on Parallel Processing*, pp. II.124–II.132, University Park PA, August 1990.

[Ho91] Alfred A. Hough, "Debugging Parallel Programs using Abstract Visualizations, "Technical Report 91:53, University of Massachusetts at Amherst, Computer Sciences Department, September 1991.

[HsK90]] Wenway Hseush and Gail E. Kaiser, "Modeling Concurrency in

Parallel Debugging," In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 11–20, Seattle WA, March 1990.

[HMW90] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang, "Analyzing Traces with Anonymous Synchronization," In *Proceedings of 1990 International Conference on Parallel Processing*, pp. II.70–II.77, University Park PA, August 1990.

[JhZw90] D.B. Johnson and W. Zwaenpoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," Journal of Algorithms, Vol. II, No. 3, pp. 462-91, Sep. 1990.

[KiZe93] Doug Kimelman and Dror Zernik, "On-the-fly topological sort- A Basis for Interactive Debugging and Live Visualizations of Parallel Programs, ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Sigplan Notices, Vol. 28(12), pp. 12-20 Dec. 1993.

[Lam86] Leslie Lamport, "Mutual exclusion problem: Part I – a theory of interprocess communication, " *Journal of ACM*, 33(2), pp. 313–326, April 1986.

[LM86] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," IEEE Transactions on Computers, Vol. C-36, NO. 4, 471-81, April 1987.

[LeM89] Ed. T. J. Leblanc and B. P. Miller, "Workshop Summary: What we have learned and where do we go from here?" In *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices, 24(1)*, pp. ix–xxii, January 1989.

[LMF90] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler, "Analyzing Parallel Program Executions using Multiple Views," *Journal of Parallel and Distributed Computing*, 9, pp. 203–217, June 1990.

[Mat89] Friedemann Mattern, “Virtual Time and Global States of Distributed Systems,” *Parallel and Distributed Algorithms*, pp. 215–226, 1989.

[McD89] Charles E. McDowell, “A Practical Algorithm for Static Analysis of Parallel Programs,” *Journal of Parallel and Distributed Computing*, 6, pp. 515–536, 1989.

[McH89] Charles E. McDowell and David P. Helmbold, “Debugging Concurrent Programs,” *ACM Computing Surveys*, 21(4), pp. 593–622, December 1989.

[Mi92] B. P. Miller, “What to draw? When to draw? An assay on Parallel Program Visualization,” TR-1103, University of Wisconsin, Madison, WI, July 1992.

[MiC88] Barton P. Miller and Jong-Deok Choi, “Breakpoints and Halting in Distributed Programs,” *Proc. 8th Intl. Conf. on Distributed Computing Systems,* pp. 316-23, Jan 1988.

[MiC89] Barton P. Miller and Jong-Deok Choi, “A Mechanism for Efficient Debugging of Parallel Programs,” *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 24(1), pp. 141-50, Jan 1989.

[MaI92] Y. Manabe and M. Imase, “Global Conditions in Debugging Distributed Programs,” Journal of Parallel and Distributed Computing, No. 15, pp. 62-69, 1992.

[NM90a] Robert H. B. Netzer and Barton P. Miller, “Detecting Data Races in Parallel Program Executions,” In *Proc. of Third Workshop on Programming Languages and Compilers for Parallel Computing*, August 1990.

[NM90b] Robert H. B. Netzer and Barton P. Miller, “On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions,” In *Proceedings of 1990 International Conference on Parallel*

Processing, pp. II.93–II.97, University Park PA, August 1990.

[NM91a]] Robert H. B. Netzer and Barton P. Miller, “Improving the Accuracy of Data race detection,” *Proc. of the 3rd ACM Symposium on Principles and Practices of Parallel Programming*, Williamsburg, VA, (April 1991).

[NM91b] Robert H. B. Netzer and Barton P. Miller, “What are Race Coonditions? Some Issues and Formalism,” Technical Report TR91-1014, University of Wisconsin, Madison, Department of Computer Science, March 1991.

[Net93] Robert H. B. Netzer, “Optimal Tracing and Replay for Debugging Shared Memory Parallel Programs,” Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging, Sigplan Notices 28(12), pp. 1-11, Dec. 1993.

[NeB92] P. Newton and J. C. Browne, “The Code 2.0 Graphical Programming Environment,” In *Proceedings of ACM Intl. Conf. on Supercomputing*, July 1992.

[New93] Peter Newton, “A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation, Ph. D. Dissertation, The University of Texas at Austin, December 1993.

[PaU89] Cherri M. Pancake and Sue Utter, “Models for Visualization in Parallel Debuggers,” In *Proceedings of Supercomputing ’89*, pp. 627–636, Reno NV, November 1989.

[PaN93] Cherri M. Pancake and Robert H. B. Netzer, “A Bibliography of Parallel Debuggers, 1993 Edition,” Proc ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Sigplan Notices, 28(12), pp. 169-86, Dec. 1993.

[Pra86] Vaughan Pratt, “Modeling Concurrency with Partial Orders,” *International Journal of Parallel Programming*, 15(1), pp. 33–71, 1986.

[RDB+87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C., Spirakis and A. Tuan. “A Browser for Directed Graphs” , Software Practice & Experience, Volume 17, #1, pp. 61-76, January 1987.

[ReSc94]] Reinhard Schwarz and Friedemann Mattern, “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,” *Distibuted Computing* 7(3), pp. 149-174, (1994)

[Rus91] Mark Russel **<mtr@ukc.ac.uk>**, “UPS: interpreter, symbolic debugger,” package available by ftp from **contrib/ups*.tar.Z** from **export.lcs.mit.edu**

[SBN89] David Socha, Mary L. Bailey, and David Notkin, “Voyeur: Graphical Views of Parallel Programs,” *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1), pp. 206–215, January 1989.

[Sch89] Edith Schonberg. On-the-fly detection of access anomalies. *Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation, published in ACM SIGPLAN Notices*, 24(7):285–297, July 1989.

[Sy83] *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, Pacific Grove, CA, March 1983. Also published in *Sigplan Notices*, Vol 18(8), Aug. 1983.

[Tay83] Richard N. Taylor, “A General Purpose Algorithm for Analyzing Concurrent Programs,” *Communications of the ACM*, 26(5), pp. 362–376, May 1983.

[Tay84] Richard N. Taylor, “Complexity of Analyzing the Synchronization Structure of Concurrent Programs,” *Acta Informatica*, 19(1), pp. 57-84, 1983.

[TaO80] Richard N. Taylor and Leon J. Osterweil, “Anomaly detection in Concurrent Software by Static Data-Flow Analysis,” *IEEE Transactions on Software Engineering*, SE-6(3), pp. 265–278, May 1980.

[UtP89] Sue Utter and Cherri M. Pancake, “A Bibliography of Parallel Debugging Tools,” *ACM SIGPLAN Notices*, 24(10), pp. 24–42, 1989.

[Vok94] Rajeev M. Vokkarne, “Distributed Execution Environments for the CODE 2.0 Parallel Programming System,” Masters Thesis, Department of Computer Science, UT Austin, December 1994.

[WaG91] Brian Waldecker and Vijay Garg, “Detection of Predicates in Distributed Programs,” Ph.D. dissertation, Univ. of Texas at Austin, 1991.

[Wag94] Yogesh S. Wagle, “Process Migration In Network Parallel Software Systems,” Masters Thesis, UT Austin, December 1994.

[YoTa86] M. Young and R. N. Taylor, “Combining Static Concurrency Analysis with Symbolic Execution”, Proceedings Workshop on Software Testing, pp. 10-18, 1986,

[ZeR91] Dror Zernik and Mark Rudolph, “Animating Work and Time for Debugging Parallel Programs; Foundation and Experience,” *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Sigplan Notices*, 26(12), pp. 46-56. Dec 1991.

Vita

Syed Irfan Hyder was born in Rawalpindi, Pakistan on August 29, 1962, the son of Syed Ahsan Hyder and Huzabra Hyder. He did his intermediate from Islamabad College for Boys in 1979 and secured a silver medal in the Higher Secondary School Certificate examination of the Federal Board. He received the degree of Bachelor of Engineering in E. E. from the NED University of Engineering and Technology, Karachi, in 1985. He, then, obtained the M.B.A degree from the Institute of Business Administration, Karachi University in 1987. After a brief stay at Space and Upper Atmosphere Research Commission (SUPARCO), Karachi, he entered the Graduate School of the University of Texas in August of 1987. In 1989, he obtained an M.S. degree from the Department of Electrical Engineering. Since then, he has been working as a Research Assistant in the CODE graphical/visual parallel programming project of the Department of Computer Science.

Permanent address: 201/O Block-II, PECHS, Karachi-75400, Pakistan

This dissertation was typed by the author.