Debugging Concurrent Programs

CHARLES E. MCDOWELL and DAVID P. HELMBOLD

Board of Studies in Computer and Information Sciences, University of California at Santa Cruz, Santa Cruz, California 95064

The main problems associated with debugging concurrent programs are increased complexity, the "probe effect," nonrepeatability, and the lack of a synchronized global clock. The probe effect refers to the fact that any attempt to observe the behavior of a distributed system may change the behavior of that system. For some parallel programs, different executions with the same data will result in different results even without any attempt to observe the behavior. Even when the behavior can be observed, in many systems the lack of a synchronized global clock makes the results of the observation difficult to interpret. This paper discusses these and other problems related to debugging concurrent programs and presents a survey of current techniques used in debugging concurrent programs. Systems using three general techniques are described: traditional or breakpoint style debuggers, event monitoring systems, and static analysis systems. In addition, techniques for limiting, organizing, and displaying a large amount of data produced by the debugging systems are discussed.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification—*assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; *diagnostics*; *monitors*; *symbolic execution*; *tracing*

Additional Key Words and Phrases: Distributed computing, event history, nondeterminism, parallel processing, probe-effect, program replay, program visualization, static analysis

INTRODUCTION

The interest in parallel programming has grown dramatically in recent years. New languages, such as Ada[1] and Modula II, have built-in features for concurrency. Older languages, such as C and FORTRAN, have been extended in a variety of ways in order to support parallel programming [Gehani and Roome 1985; Karp 1987].

The added complexity of expressing concurrency has made debugging parallel programs even harder than debugging sequential programs. In the remainder of this section we will justify this claim and outline the basic approaches currently used for debugging parallel programs. In Sections 1–4 we discuss each of these approaches in detail. We conclude with Section 5 and an appendix with tables that summarize the features of 35 systems designed for debugging parallel programs.

Difficulty Debugging Concurrent Programs

The classic approach to debugging sequential programs involves repeatedly stopping

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

This work was supported in part by IBM grants SL87033 and SL88096.

the program during execution, examining the state, and then either continuing or reexecuting in order to stop at an earlier point in the execution. This style of debugging is called *cyclical debugging*. Unfortunately, parallel programs do not always have reproducible behavior. Even when they are run with the same inputs, their results can be radically different. These differences are caused by *races*, which occur whenever two activities are allowed to progress in parallel. For example, one process may attempt to write a memory location while a second process is reading from that memory cell. The second process's behavior may differ radically, depending on whether its reads the new or old value.

The cyclical debugging approach often fails for parallel programs because the undesirable behavior may not appear when the program is reexecuted. If the undesirable behavior occurs with very low probability, the programmer may never be able to recreate the error situation. In fact, any attempt to gain more information about the program may contribute to the difficulty of reproducing the erroneous behavior. This has been referred to as the "Heisenberg Uncertainty" principle applied to software [LeDoux and Parker 1985] or the "Probe Effect" [Gait 1985]. For programs that contain races, any additional print or debugging statements may modify a crucial race, lowering the probability that the interesting behavior occurs. This interference can be disastrous when attempting to diagnose an error in a parallel program.

The nondeterminism arising from races is particularly difficult to deal with because the programmer often has little or no control over it. The resolution of a race may depend on each CPU's load, the amount of network traffic, and nondeterminism in the communication medium (e.g., exponential backoff protocols [Tannenbaum 1981, pp. 292–295]). It is this nondeterministic behavior that tends to make understanding, writing, and debugging parallel programs more difficult than their sequential counterparts.

An additional problem found in distributed systems is that the concept of "global state" can be misleading or even nonexistent [Lamport 1978]. Without a synchronized global clock, it may be difficult to determine the precise order of events occurring in distinct, concurrently executing processors.

Basic Approaches

Some researchers distinguish between monitoring and traditional debugging [Joyce et al. 1987]. *Monitoring* is the process of gathering information about a program's execution. *Debugging*, as defined in the current ANSI/IEEE standard glossary of software engineering terms, is "the process of locating, analyzing, and correcting suspected faults," where a *fault* is defined to be an accidental condition that causes a program to fail to perform its required function. Since monitoring is often an effective procedure for locating incorrect behavior, it should be considered a debugging tool.

For the purposes of this survey, techniques for debugging concurrent systems have been organized into four groups:

(1) Traditional debugging techniques can be applied with some success to parallel programs. These are discussed in Section 1.

(2) Event-based debuggers view the execution of a parallel program as a sequence (or several parallel sequences) of events. The generation and analysis of these sequences or *event histories* is the subject of Section 2.

(3) Techniques for displaying the flow of control and distributed data associated with parallel programs are presented in Section 3.

(4) Static analysis techniques based on dataflow analysis of parallel programs are presented in Section 4. These techniques allow some program errors to be detected without executing the program.

This survey covers a large number of research and commercial projects designed to help produce error-free concurrent software. It focuses primarily on systems that are directed toward isolating program errors. A large body of work in formal program verification and in program testing

has been explicitly excluded from this survey. Most of the systems surveyed fall into one of two general categories, traditional parallel debuggers (or what are sometimes called "breakpoint" debuggers) and event-based debuggers. Of course, some systems contain aspects of both classes. All of the systems (or in some cases proposed systems) in these two general categories are listed in the tables in Appendix A.

In addition to traditional parallel debuggers and event-based debuggers, some static analysis systems are included. The static analysis systems surveyed fall somewhere between debugging and testing. The static analysis systems are distinguished from testing by not requiring program execution and by generally checking for *structural faults* instead of *functional faults.* That is, the analysis tools have no knowledge of the intended function of the program and simply identify program structures that are generally indicative of an error. These systems do not appear in the comparison table in Appendix A but are discussed in Section 4.

Each of the three types of systems surveyed takes a different approach to the debugging problem. The traditional parallel debuggers are the easiest to build and therefore provide an immediate partial solution. They provide some control over program execution and provide state examination. They are also severely limited by the probe effect.

Event-based debuggers provide better abstraction than that provided by traditional style debuggers. They also address the probe effect by permitting deterministic replay of nondeterministic programs. If it is not possible to record event histories continuously, however, the probe effect will still be a problem. Also, event-based debuggers are generally research prototypes, applicable only to systems without shared memory. A notable exception is instant replay [LeBlanc and Mellor-Crummey 1987], which supports event tracing and replay on the shared memory BBN Butterfly provided OS protocol routines are used for all shared memory accesses.

Static analysis tools avoid the probe effect entirely by not executing the programs. They have the potential of identifying a large class of program errors that are particularly difficult to find using current dynamic techniques. These techniques have been applied mostly to parallel versions of FORTRAN that do not support recursion. As with the event-based debuggers, static analysis systems are still in the prototype stage. The primary problem with most static analysis algorithms is that their worst-case computational complexity is often exponential.

All three types of debugging systems have made some progress in presenting the complex concurrent program state and the accompanying massive amounts of data to the user. Multiple windows is a useful mechanism for interfacing with traditional style debuggers for parallel systems. The abstraction capabilities of event-based debuggers (see Section 2) have been used to present interesting and potentially useful views of system states graphically [Hough and Cuny 1987].

1. EXTENDING TRADITIONAL DEBUGGING TO PARALLEL PROGRAMS

The simplest type of debugger to implement for parallel systems is (or behaves like) a collection of sequential debuggers, one per parallel process. To date, all commercially available debuggers for parallel programs fit this description. The primary differences lie in how the output from the several sequential debuggers is displayed and how the separate sequential debuggers are controlled. We will call these collections of sequential debuggers *traditional parallel debuggers.*

The probe effect, discussed in the Introduction, has gone mostly unaddressed by traditional parallel debuggers. This makes traditional parallel debuggers ineffective against timing dependent errors. The probe effect, however, does not always rear its ugly head, allowing many program errors to be isolated using traditional cyclic debugging techniques. This can be attributed to two factors. First, those errors in parallel programs that are not timing dependent would never be masked by the probe effect. Second, even for timing related errors, the

effect of the probe may not disturb the outcome of the critical races.

Another criticism of traditional parallel debuggers is that they operate at too low a level. For programs consisting of many concurrently executing processes, the major difficulty may be in understanding what is happening at the interprocess level. Traditional debugging techniques work well for viewing the behavior at the instruction level or at the procedure level. In Section 2.4 some recent developments for viewing program behavior at a more abstract level are presented.

1.1 Coordinating Several Sequential Debuggers

In addition to the sequential capabilities of standard sequential debuggers, traditional parallel debuggers should be able to do the following:

1. direct any sequential debugger command to a specific task,
2. direct any sequential debugger command to an arbitrary set of tasks,
3. differentiate the terminal output from the different tasks.

The most primitive debugger for parallel programs would be nothing more than a sequential debugger capable of attaching to any single process in a parallel program. All that would then be necessary is to provide the user with multiple real or virtual terminals from which to execute the multiple copies of the debugger. Today's multiple window workstations make this more practical than it might have been a few years ago. With a window manager [Scheifler and Gettys 1986; Sun Microsystems 1986] points 1 and 3 could be satisfied by selecting the desired window. Satisfying point 2 could be achieved simply by repeating the desired command in each of the desired windows. This approach, however, would become fairly unwieldy for more than a few processes. Furthermore, the time lapse between sending the command to the first and last processes in the set could aggravate the probe effect. This is particularly true for commands such as "stop" and "continue."

The Sun Microsystems' *dbxtool* is an example of applying a set of sequential debuggers to concurrent programs without any explicit coordination. It is capable of attaching to an existing UNIX[2] process, making it possible to debug a system of communicating UNIX processes by attaching a separate copy of dbxtool to each process. (The UNIX process may not contain process creation calls such as "fork," and the executable image being debugged cannot be shared.)

An alternative to relying on a window manager to direct commands to the proper sequential debugger is to control all of the debuggers from a single terminal or window [Sequent Corp. 1986]. Commands are then directed at a specific process using a command parameter or by defaulting to a specific "current" process. For example, "continue P1" would continue process P1, and "continue" without a parameter would continue the "current" process. The "current" process can be changed at any time. The use of a single control window also permits the commands to be sent to all processes. For example, "continue all" would continue all currently suspended processes. In general, all processes will not receive the command at the same instant. The commands will, however, arrive at times that differ by an amount approximating the communication delay in the system. If all processes could be instantaneously stopped (and started) then, in the absence of timeouts, "stop all" breakpoints would not cause any probe effect. This is, of course, impossible, but anything that can be done to minimize the time difference for receipt of stop signals should reduce the probe effect. In addition to reducing the probe effect, broadcasting a single command to a set of processes is a useful feature.

The "current" task notion is generalized in Griffin [1987] and Intel Corp. [1987] to a current *set* of processes that all receive any process-related commands. In Griffin [1987], processes can be added to or removed from the set simply by pointing to a symbol for the process in a special window.

[2] UNIX is a trademark of AT&T Bell Laboratories.

This could be generalized to permit arbitrary groupings of processes. For instance, it might be desirable to alternate commands between two disjoint sets of processes. With only a single "current" set and no overlap between the desired sets, this would require as many commands from the user as would be required with no support for process grouping. It would, however, still reduce the probe effect. It appears that the macro capability of Intel Corp. [1987] combined with their "context" command for specifying the current set of processes would support this toggling back and forth between disjoint sets of processes.

1.2 Breakpoints

The ability to set breakpoints is possibly the most important feature of a sequential debugger. (Since tracing is equivalent to setting a breakpoint that, when encountered, prints some information and automatically continues, the discussion in this section will refer only to breakpoints.) Traditional parallel debuggers generally support the same types of breakpoints as those found in sequential debuggers. These breakpoints include stop at a source statement, stop on the occurrence of an exception or some user detectable event, stop when a specific variable is accessed, and stop when some conditional expression is satisfied [Seidner and Tindall 1983]. Unlike sequential debuggers, there are two possible actions to take when a breakpoint is encountered. Either all of the processes in the parallel program can be stopped immediately or only the process encountering the breakpoint can be stopped. The former can be difficult to achieve within a sufficiently small interval of time, and the latter can have a serious impact on systems that contain such things as timeouts. Assuming message passing is the communication mechanism, an algorithm to stop all processes in a consistent state is presented in Miller and Choi [1988a].

Using breakpoints to debug systems with explicit time-dependent operations (such as timeouts) can be especially difficult. Some systems have attempted to deal with such explicit race conditions by supporting a notion of logical time that stops when any process reaches a breakpoint [Cooper 1987; DiMaio et al. 1985]. In systems that suspend only the selected process, other processes will continue to execute until they encounter some explicitly time-dependent operation. In that case, the logical clock is the one used for time in the time-dependent operation. For example, if a breakpoint is encountered, no timeouts will expire until the suspended process is continued. In systems that stop all processes upon encountering a breakpoint, logical time is stopped so that all of the suspended processes can be continued with minimal impact. This will certainly not eliminate the probe effect, but it can permit some traditional style debugging in the presence of such explicitly time-dependent operations.

The domain of expressions or predicates used to describe a breakpoint is larger for parallel programs than for sequential programs. These predicate expressions may involve both process state and events. An *event* can be loosely defined as any atomic action visible beyond the scope of a single process.

Predicates involving global state in an executing parallel program can be a problem. This results from the lack of global clock in most systems. For example, an expression such as "process A never modifies variable X while process B is modifying variable X" may appear to be true due to the delay in communicating this information to the debugger, when in fact concurrent modification has occurred. The use of events and some notion of consistent global time can be used to address this (see Section 2). Possibly more important is that it may not be possible to stop the desired processes after detecting that the predicate is satisfied yet before the state has changed.

The distinction between events and global states is admittedly vague in general but is usually well defined for any particular system. For example, an event-based predicate might be "process A sends a message," and a global state-based predicate might be "the message buffer contains a message from process A." This could even be represented as a collection of program counter-based breakpoints, one immediately following each send statement in process A. In

systems that deal with event-based predicates, there must be some language for describing events. The language may be as simple as naming one of a finite set of events such as "taskstart" or "sendmessage"; or it may support relatively powerful abstractions such as those described in Section 2. Table A.4 summarizes the breakpoint capabilities of the systems surveyed.

1.3 OS Support for Parallel Debuggers

Parallel debuggers that support global state-based breakpoints and event-based breakpoints place greater demands on the operating system than sequential debuggers. This is just one more step in the evolution of debuggers. Early debuggers only needed the ability to examine core memory and the saved values of CPU registers after the program terminated. Next was added the ability to set breakpoints. The operating system provided a means of modifying the executable image and of passing control to the debugger when the breakpoint instruction was encountered. Most operating systems also pass control to the debugger for most program exceptions. Some hardware architectures now also include a special trace mode to facilitate single stepping. The final feature that is provided is a mechanism for passing control to the debugger when a specific memory location is accessed. To summarize, a state-of-the-art sequential debugger may need the following capabilities to be provided by the operating system or hardware:

- the ability to read or write a register or memory location,
- the ability to set and trap breakpoints,
- the ability to trap program exceptions,
- the ability to single step a program,
- the ability to trap memory accesses.

Traditional parallel debuggers that go beyond being simply a collection of sequential debuggers acting together may require more of the underlying operating system. Debuggers that fit the traditional cyclic debugging paradigm, using breakpoints and tracing, require one or more of the following facilities:

- the ability to trap on any interprocess communication (IPC),
- the ability to modify/insert/delete IPC messages,
- the ability to control the clock used for timeouts.

Several approaches have been taken to provide these capabilities. Some debugging systems modify the program source in order to provide the necessary hooks for the debugger at run time. This avoids the need for modifying the operating system at the expense of slower performance and restricted capability. A second approach is to provide an alternative set of system routines. This permits the debugger to intervene in any interaction between the user and system. The final approach is actually to modify the operating system to provide the necessary hooks. At some point it may become cost effective to implement more of the debugging hooks directly in the machine architecture. The *interface* entry in Table A.2 summarizes where the hooks were placed for the systems surveyed.

2. EVENT HISTORIES

Since the various debuggers surveyed were designed for different environments, it is only natural that they do not agree on the definition of "event." For example, in the DISDEB system events are memory accesses, in Radar each message send or receive is an event, in Instant Replay an event is the access of an object, and in YODA events represent Ada tasking activity. In some systems, such as TSL and EDL, events can be defined by the programmer. In systems with explicit interprocess communication, events can be divided into two classes: those representing interprocess communication activity and those representing activity internal to a single process. This distinction does not seem to hold for shared memory systems, since each memory access is a potential interprocess communication.

Some systems merely display the events as they occur (see Appendix A). A more

powerful approach is to record an *event history* containing all of the events generated by the program. The history can then be examined by the user after the program has completed. Since the event history is often very large, some debuggers provide facilities to browse or query the history. Event histories can also be used to guide the program's execution, allowing the reproduction of erroneous computations. If the history is complete enough, a single process can be debugged in isolation with the history providing the needed communication. Finally, some systems can automatically check the history for suspicious behavior or transform the lower level history of events into more meaningful high-level events.

2.1 Recording Event Histories

A common approach is for the debugger to do as little as possible, mainly recording information, at run time. By limiting the debugger's activity, the probe effect should be reduced. The recorded information can then be analyzed following the program's execution.

2.1.1 Which Information to Record

The amount of information that must be recorded for each event depends upon how the event history is going to be used. Three general levels of use that require increasing amounts of detail to be recorded for each event are the following:

(1) Browsing—The event history is examined possibly through the use of specialized tools. Examination methods range from text editors to "movies" showing the state changes caused by events [Hough and Cuny 1987; Le Blanc and Robbins 1985].

(2) Replay—The debugger uses the event history to control a reexecution of the program. This permits the use of conventional debugging techniques, such as breakpoints, state examination, and single stepping, without changing the behavior of the program.

(3) Simulation—The event history can be used to simulate the environment of any single process. This permits the use of a sequential debugger on a process without reexecuting the entire program.

Browsing requires only minimal information about each event. Simply recording the kinds of events executed by a process can help isolate an error. Of course, if more information is recorded, then more information will be available to the programmer.

One problem with browsing event histories is that the histories frequently contain enormous numbers of events, making it difficult to locate the events of interest. Some systems allow selective recording of information, and others include powerful mechanisms for examining the event history (see Section 2.2).

To replay an execution requires enough information so that the next event in which each process participates can be determined. LeBlanc and Mellor-Crummey [1987] describe a method that reduces the amount of information needed for replay compared with previous methods that recorded the complete contents of all messages. Their ideas work because the program generates the contents of the messages during the reexecution.

Simulating the rest of the program so that a single process can be debugged in isolation requires that all events visible to the process be recorded. This includes both the contents of messages and the values written to shared memory. Note that reexecuting a single process requires more information than reexecuting the entire system.

If the interesting portion of the execution can be identified, then the amount of information required for replay can be considerably reduced. Instead of recording the entire history, the debugger can take a snapshot of the program's state and keep only that part of the history that follows the snapshot. It may, however, be difficult to obtain accurate snapshots in distributed systems efficiently (see Chandy and Lamport [1985] for one method). This technique may work best for simulating a single process, since only that process's state needs to be recorded.

2.1.2 How the History Gets Recorded

In addition to the amount of information recorded in an event history, some attention must be given to how the recording is done and the resulting impact on performance and the probe effect. In the systems surveyed, the methods used to generate the event history varied from inserting appropriate statements in the original source program to monitoring the buses passively. An intermediate method of recording event histories is to provide modified system routines. These modified routines record the history in addition to performing their normal system functions. In some cases such as LeBlanc and Mellor-Crummey [1987] it is the user's responsibility to insert system calls that record the event information. In others it is only necessary to link the program using special monitoring versions (see the interface entry in Table A.2).

With the exception of the bus monitoring, all other recording methods resulted in possible changes in timing and hence are potentially susceptible to the probe effect. In some papers (see the probe effect in Table A-2) it was argued that the performance impact of monitoring was sufficiently small to justify leaving the recording permanently enabled. In other papers it was argued that the perturbations caused by the monitoring software were sufficiently small to avoid the probe effect in most cases.

2.1.3 Linear Versus Partially Ordered Event Histories

An issue that arises in distributed systems is whether the event history should be partially or linearly (i.e., totally) ordered. On the one hand, a linearly ordered history is simpler to understand and can be easier to work with. On the other hand, a linear stream is often misleading since it implies an ordering between every pair of events—even when the events are completely unrelated. A partial ordering on the events is necessary to reflect the behavior of a distributed system accurately.

One way to represent the partial order (used by Instant Replay [LeBlanc and Mellor-Crummey 1987]) is to record a separate history tape for each process. Although each history tape is a linear stream, together they (with the program) represent the partial ordering of the events in the computation. The Traveler debugger [Manning 1987] for Acore (a LISP-like language) keeps a history tape ("lifeline" in their terminology) for each object in the program. In addition, the Traveler system records the partial order by explicitly linking each action to the child actions it causes (or parent action it allows to continue).

A general technique for obtaining the partial order involves associating each event with a vector of "logical timestamps" [Fidge 1988; Haban and Weisel 1988]. The order (or absence thereof) between two events can be easily determined by comparing the vectors of timestamps associated with the events.

2.2 Browsing Event Histories

Many systems recognize the need for facilities that help interpret massive event histories. Graphical techniques such as time-process diagrams and animation are discussed in Section 3. A simple feature found in many systems is *filtering*, whereby the events that the programmer feels are unimportant are automatically discarded, usually based on process or kind of event (see Table A.5). Two systems go further, storing the event history in a database for easy examination.

The YODA system for Ada tasking programs [LeDoux and Parker 1985] stores the event history as Prolog facts. Prolog predicates define the common temporal relationships such as during, before, and after. The user is able to define and store additional predicates (either temporal or nontemporal). Existential queries, such as "which tasks updated variable X before task T," are used to retrieve information from the history. Note that the YODA system stores when (using a global event counter) and to what values variables are updated, as well as the explicit intertask communications.

In [Snodgrass 1984], the events are captured in a relational database. In addition to the event relations, the database

contains interval relations indicating behavior with duration. Time is kept in microseconds, presumably using a global clock. TQuel, a version of Quel augmented with temporal constructs, is used to build new relations from those in the event history. The user queries the history by building and printing the appropriate relation.

2.3 Replaying Event Histories

Several of the debugging systems allow a program to be reexecuted under the control of an event history (see Table A.5). We call this capability *replay*. If the history correctly reflects the interprocess communication of the original execution, then the replay produces the same results. Although replay helps solve the reproducibility problems, it is useful only if additional information is gained.

One way to gain information is to replay the program in "debug mode," with a traditional sequential debugger attached to each process. This allows the internal state of the processes to be examined, giving the programmer significantly more information than a stream of IPC events. Some systems allow a single suspect process to be replayed in isolation, with the remainder of the program simulated by the event history. This approach reduces the parallel debugging problem to that of debugging a single sequential process, once the faulty process is identified.

Another way of gaining information is to execute a modified program under the control of an event history. The event history can control the new program as long as its behavior is compatible with the old [Curtis and Wittie 1982; LeBlanc and Mellor-Crummey 1987]. This facility allows the programmer to add additional debugging statements or experiment with modified algorithms. Another advantage of this feature is that corrected programs can be tested against the same input and history that caused the previous version to fail.

The added synchronization involved during replay can dramatically slow down a parallel program. The Instant Replay system [LeBlanc and Mellor-Crummey 1987] reported some of the best results in this regard. On one example (gaussian elimination of a 400 × 400 matrix on up to 64 processors) they report that (with tuned monitoring) the overhead involved in collecting the event history amounts to less than 1 percent. Furthermore, there was no additional decrease in performance when the execution was replayed.

Replaying real-time systems has several additional problems. The external I/O and interrupts must be recorded in addition to the communications between the processes. Since real-time systems generally have a strong time dependency, it may be important to simulate time during the replay. If behavior due to the absence of communication, such as timeouts, takes place, then faithfully reproducing such behavior requires that additional events appear in the history.

2.4 Checking and Transforming the Event History

Several debugging systems compare the event history generated by the program with a set of predicates. These systems are more than browsers since the analysis is done at run time. In addition, the violation of a predicate can trigger additional debugging action. Because the analysis is done at run time, all of the predicates to be tested must be written before the program starts executing. This disadvantage can be reduced if the event history is recorded so that the execution can be replayed.

The DISDEB system [Lazzerini and Prete 1986] relies on programmable debugging aids that eavesdrop on bus traffic, avoiding the probe effect. Although this system contains several interesting features, it has a very low-level interface. An event definition is of the form "process P with certain permissions accesses memory location X reading/writing value V." The memory location is mandatory, and the value may be a range. If part of the event definition is omitted, then that part is treated as "don't care." DISDEB allows state to be stored in two ways. First, counters and timers can be defined and used; second, event definitions can be enabled and disabled. Once a suitable set of events has been defined, it can be used to

trigger debugging actions; for example, "when $E1$ occurs, display counter C and stop process N." Other potential actions include starting and stopping traces of memory locations and manipulating timers.

A similar approach is taken by the HARD system for Ada tasking programs [Di Maio et al. 1985]. There the predicates and debugging actions are encoded in special Ada tasks called D tasks. Manually inserted calls to the D tasks enable them to obtain information about the program's execution. Based on this information, they can call routines that display or modify the program state. All of the Ada facilities can be used inside of a D task, so the programmer can use a familiar high-level language to control the debugging process.

Rather than using the stream of events to control debugging activity, the following systems automatically check specifications for the program. Although this requires that the programmer learn an additional language, it can complement a formal specification/verification approach to program development. Most of these systems have their own way of specifying complex event formulas, usually based on the sequential and parallel composition of events.

The IDD system [Harter et al. 1985] uses an interval logic specification of the program. This specification is checked against the program's behavior. When a specification is violated, the program is stopped for inspection. Temporal logic views the computation as a sequence of states. The main operators are "always" and "eventually," meaning that the following predicate on the state is either always true or eventually becomes true. Interval logic adds expressibility by restricting the temporal operators to portions of the computation.

The ECSP debugger [Baiardi et al. 1986] can check *behavior specifications* that completely describe the allowable communication behavior of the processes. The specifications can refer to various communication activity and can contain assertions on the process's state. One of the constructs in their language causes control to be returned to the user (presumably so that the process can be examined). The ordering of events and checking of specifications in this system is simplified by the restriction that each specification can only refer to events from a single process.

The TSL system [Helmbold and Luckham 1985a] automatically checks specifications against the events generated by an Ada tasking program. Each TSL specification is of the form "*when* this occurs *then* that occurs *before* something else," where each of the three parts is an event formula. TSL contains placeholders allowing a single specification to constrain multiple tasks. Additional abstraction is gained by using macros for event subformulas. An important contribution of the TSL system is its use of Ada semantics to guarantee that, even in distributed systems, certain pairs of events appear in the history in the correct order.

The Event Description Language (EDL) takes a slightly different approach [Bates and Wileden 1983]. Instead of checking specifications against the event history, it provides a method for defining multiple levels of abstract events from the primitive events generated by the program. Each high-level event is defined by an event formula over lower level events. There is one clause that constrains the values associated with the lower level events and another that determines the values associated with the higher level event. The Belvedere system uses EDL to help control its display (see Section 3.3).

All of these specification methods have simplifying restrictions. In EDL, an accurate global clock is assumed, the event recognizer is a potential bottleneck, and some ambiguity arises when a low-level event can be used in multiple higher level events (see, however, Bates [1988]). The TSL specification checker requires a linearly ordered stream of events and is also a bottleneck in the current application. In the IDD system, events are restricted to broadcasts on a shared medium (such as an Ethernet). A tree structure method for evaluating the IDD interval logic expressions is briefly described. The ECSP assertion checker is for a hierarchical fork-join method of parallelism. Its main disadvantages are that each specification deals only with the activity of one process and all processes must be completely specified.

3. GRAPHICS

Sequential programs lend themselves reasonably well to being debugged with a single sequential output device. There is only one thread of execution that can accurately be displayed as sequential text. Also, the data are logically stored in one place and can be displayed when desired. Parallel programs are different in these two areas. In parallel programs, there are multiple threads of control and the data may be logically as well as physically distributed. An important goal of research into parallel debugging systems is to find ways of presenting the distributed data and control of parallel programs to users in a manner that aids in comprehension.

Four basic techniques for displaying debugging information are as follows:

(1) *Textual* presentation of the data, which may involve color, highlighting, or a display of control flow information. Time may not progress monotonically from the top of the screen to the bottom.

(2) *Time-process diagrams* that present the execution of the program in a two-dimensional display with time on one axis and individual processes on the other axis. The points in the display are labeled to indicate the activity of the specified process at the specified time.

(3) *Animation* of the program execution, whereby both dimensions of the display are spatial dimensions. The display corresponds to a single instant in time, or snapshot of the program state. These snapshots can be displayed one after another, animating the program's execution. The actual format of a single frame can take many forms as discussed below.

(4) *Multiple windows*, whose use permits several simultaneous views of the program being debugged. This frequently involves using one window per process.

Each of these approaches will be discussed below with examples taken from the surveyed papers.

```
proc Simple = reply(request(A) + request(B)
    + request(C));
proc A = reply(request(X) + request(Y));
proc B = reply(b);
proc C = reply(c);
proc X = reply(x);
proc Y = reply(y);
```

Figure 1. A stylized transaction program.

3.1 Text Windows

A simple text presentation of the debugging information is the most common type of display. All of the systems make use of some simple text displays. For a traditional parallel debugger, this may be the only type of data display (see Section 1.1). In an event-based system, a sequential display of the events as seen by a particular process can be useful. The *Traveler* [Manning 1987], which is an object-based system, can display a "lifeline" that is a sequential list of processes in the order that they accessed a particular shared object.

It is not always necessary for time to progress monotonically from the top of a sequential text display to the bottom. In Traveler events are displayed in their *causal* order instead of in their *temporal* order. Traveler is used to debug message passing programs. In their model, all messages come in pairs, with a request and a reply. All requests block until a reply is received, and the programming model is such that making one request may result in many more nested requests before a reply is sent. Also, one process may send several requests concurrently. The Traveler display presents these *nested* request–reply pairs, which they call *transactions*, in such a way as to emphasize the nested structure. Requests sent concurrently will be displayed paired with their responses and nested within the send–receive pair that caused them to occur. Figure 1 gives a stylized example program using requests and replies. "Request(A)" sends a message to request a response from process A. "Reply(x)" sends the reply "x". Figure 2 gives a possible sequential ordering of the events for a partial execution of the request "request(Simple)". Figure 3 gives the nested transaction display for the same partial execution.

```
request(Simple)
request(A)
request(B)
request(X)
request(C)
request(Y)
reply(y)
reply(c)
```

Figure 2. A sequential display.

```
request(Simple)
  request(A)
    request(X)
    [no response]
    request(Y)
    reply(y)
  [no response]
  request(B)
  [no response]
  request(C)
  reply(c)
[no response]
```

Figure 3. A nested display.

For large programs, all transactions and their subtransactions might not fit on the display simultaneously. To handle this, the user may selectively *open* and *close* transactions. When a transaction is closed, all of its subtransactions are hidden. For example, in Figure 3, if the transaction for "request(A)" were closed, then "request(X)", "request(Y)", and their corresponding response lines would not be shown. As the computation advances, the various "[no response]" entries will be filled in.

3.2 Time-Process Diagrams

A *time-process* diagram is a two-dimensional representation of the state of a parallel system over time. One axis represents time, and the other axis represents the processes. Each point in the display gives some information regarding the state of the corresponding process at the corresponding time. For example, a simple time-process diagram for a hypothetical system is shown in Figure 4. Each row describes the events in which a process engages, and each column describes the state of the system at a particular instant in time. At time 1, process 1 engages in event A; at time 2, process 2 engages in event B and process 3 engages in event C; and so on.

Process 1	A		D	F		...
Process 2		B			G	...
Process 3		C	E		H	...

Figure 4. A simple time-process diagram.

In Griffin [1987] time-process diagrams are used without enhancement to display the activity of this shared memory-based system. It has the advantage that it can be presented on a simple text screen (see Figure 5). This system is actually a uniprocessor simulation of a multiprocessor, and this is evident in the display. The unit of time is the occurrence of an event. A single character is used to represent each of the possible events in which a process may engage. Because this is a uniprocessor, each column will have one event character and all other active processors will have a "." in that column. The last character other than "." in a row corresponds to the last event in which the process engaged. This can be thought of as the process's current state. The display can be scrolled forward or backward in time. The last non "." in each row that was scrolled off to the left is maintained in the leftmost column. The display contains additional text that provides additional information about the current (rightmost column) state. This includes such things as which signal a process is waiting for and which signals have been posted. (To avoid confusion with our broader use of the word event, we use signal here where the mtdbx system [Griffin 1987] uses event.)

In [Harter et al. 1985] the message-passing system, Idd, there is heavier use of graphics. Instead of placing one character at each point in the display, two points in the display are connected by a line to indicate the transmission and receipt of a message (see Figure 6). To aid in comprehension of a potentially very cluttered display, the user can *magnify* or scroll to see only a selected portion of the display. The *display* option allows the user to rearrange the rows so that related processes can be placed close together. The user may also select various filters to be used in displaying

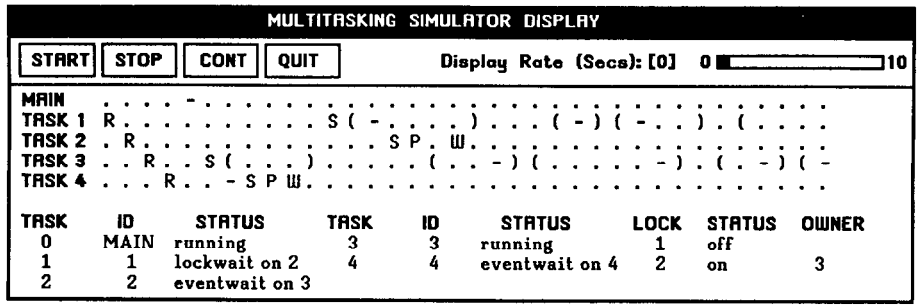

Figure 5. The mtdbx time-process diagram.

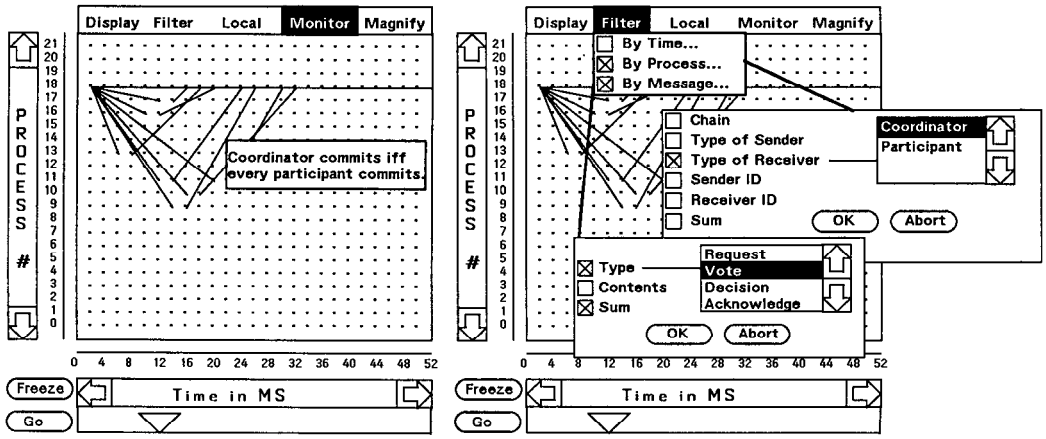

Figure 6. The Idd time-process diagram. Used with permission [Harter 1985] © 1985 IEEE.

subsets of the messages that fall within the time-process space currently being displayed. Similar displays are included in PPUTT [Fowler et al. 1988] in which the emphasis is on the programmer noticing irregularities in the patterns of communication.

For many of the variations on time-process diagrams a global clock is required. At least one system [Stone 1988], however, uses a type of time-process display without needing a global clock. This display is called a *concurrency map*. Instead of displaying exactly when events occurred based on a global clock, events are arranged to show only the order in which they occurred. This order is derived from the time dependencies in the program. For example, the receipt of a message must follow the sending of a message. Figure 7 shows process histories for three processes and the corresponding concurrency map. The horizontal lines (partially obscured by the boxes) correspond to logical time divisions. An event may have occurred during any time division touched by the box containing the event.

Time-process displays appear to have a definite place in viewing the activity of parallel systems. They do have their limitations. As the number of processes becomes large, the display may become too cluttered with information to be useful. This can be addressed to a degree with filtering and such features as the *display* and *magnify* options described above for Idd. In the next section, we present an alternative display that gives a much different view of the system.

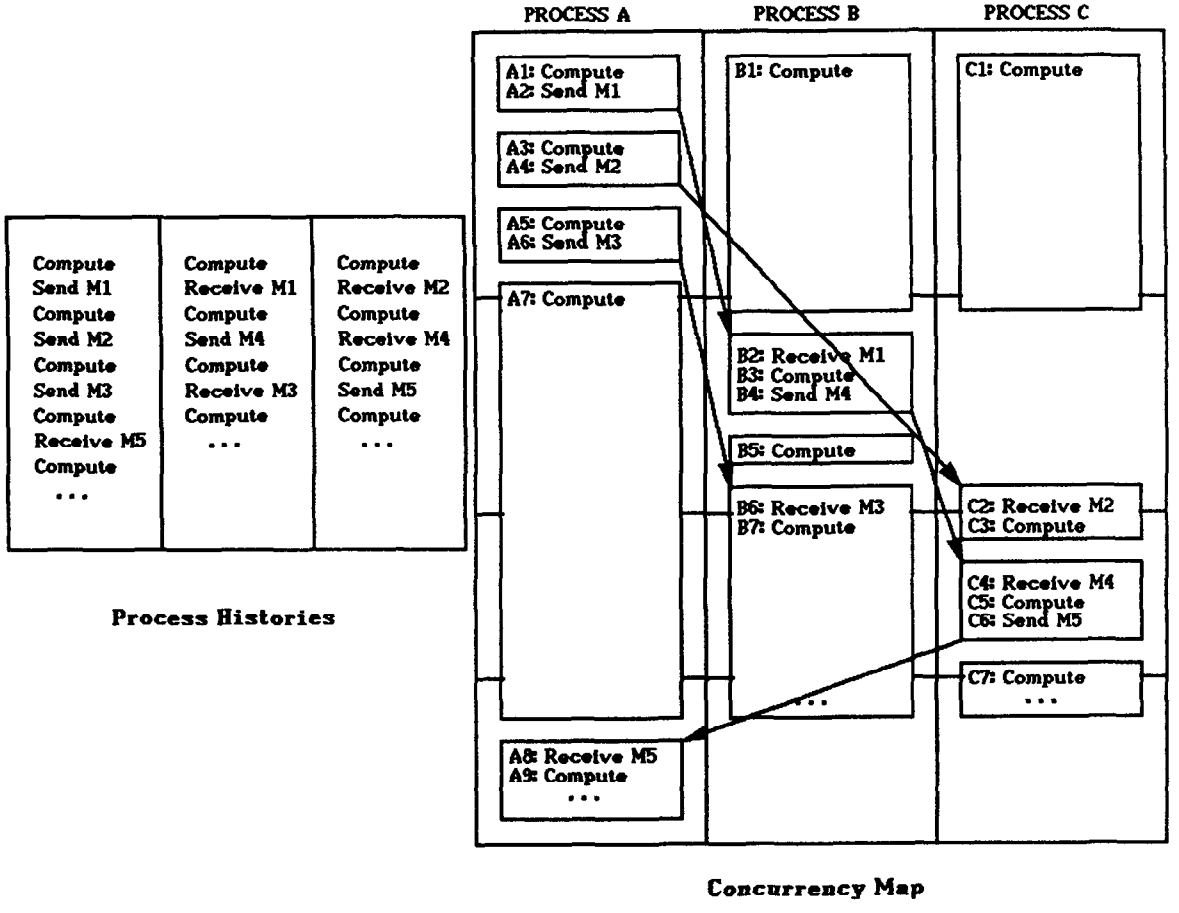

Figure 7. Process histories and a concurrency map.

3.3 Animation

An alternative to time-process displays is to place each process (or selected portions of distributed data) at a different point in a two-dimensional display and have the entire display represent the system at a single instant in time. As time advances, the display changes and these changes can be played in sequence to give a type of *animated* movie. This movie displays the evolution of the state of the system. The placement of the processes (or data) in the two-dimensional display could be arbitrary, be under user control, or correspond to the underlying structure of the program (or data) being represented.

In Belvedere [Hough and Cuny 1987], the placement of processes is very important and is specified by the user. The basic animation elements are depicted in Figure 8. This system animates primitive events and user defined events specified using EDL. To further help organize the display, the user may request that events be displayed from different perspectives: that of a processor, a channel or a data item. For example, when viewed from the perspective of a single processor, the events will be displayed in the order that they appeared to that processor. Examples of this can be found in Hough and Cuny [1987] and in Figure 8. Figure 8 is a snapshot of message traffic animation during a traveling salesman program. Activity is depicted by highlighting the appropriate ports (small boxes) and channels (lines). A port is highlighted during a receive and a channel during a send. Arrowheads indicate direction for sends, with multiple arrowheads indicating more than one message in the buffer.

The Radar [LeBlanc and Robbins 1985] system also uses animation of messages. In Radar, the user can control how long each time frame is displayed. Also, at any time, the user can have the contents of any message displayed.

Voyeur [Socha 1988] is a prototype system for the construction of application specific "views" of parallel programs. Its input is a sequence of events generated from user-inserted instrumentation code. The user

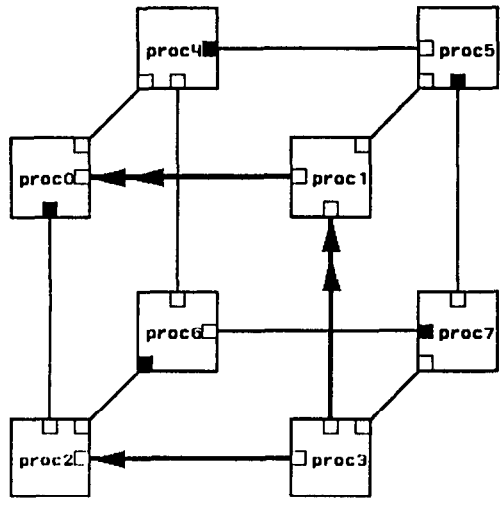

Figure 8. Animation using Belvedere. Used with permission of Hough and Cuny [1988].

can use existing views or build new ones. Four views have been constructed and are described in Socha et al. [1988]: icon view, vector view, trace view, and linked-list view.

In the icon view, the events indicate where in a two-dimensional picture a particular icon should be drawn and when to start a new animation frame. By observing the position of the icons within a single frame and their change in position from frame to frame, several errors have been detected. The vector view is a variation of this where instead of drawing icons, vectors are drawn.

The trace view provides a small box (window) for each process. Connections to other processors are shown as lines to other boxes. Values local to the process are then displayed inside the box. The linked-list view is a variation of the trace view that displays a linked-list data structure from the program. The nodes are drawn as boxes, with the actual data values displayed. The boxes are then connected to show the actual link structure.

The emphasis in Voyeur is the ability to provide an easy method for programmers to animate their parallel programs for the purpose of uncovering errors.

3.4 Animation Versus Time Process

It should come as no surprise that neither animation nor time-process diagrams alone is sufficient to detect all of the errors in parallel programs easily. Animation is good for observing the instantaneous state of the system. By only displaying a single instant of time, more state information can be displayed simultaneously. This may mean more information per process or more processes. Patterns of concurrent behavior can also be viewed (e.g., all processes *except one* are sending messages to their neighbor). Animation, however, does not clearly show patterns of behavior that occur across time. This is addressed to some degree in Belvedere by the use of high-level abstract events that may encompass an interval of time.

Time-process diagrams can display patterns of behavior over time. This can be especially helpful in finding performance bugs. The trade-off is that only a very small amount of information can be displayed for each process at any point in time. In mtdbx [Griffin 1987] only a single character is displayed for the entire state of a process. In PPUTT a process is either waiting, running, sending, or receiving.

It appears that the ideal debugger for complex concurrent systems would support both animation and time-process displays. Two pieces of evidence to support this claim are the following:

- The animation systems find some errors by noticing changes from one frame to the next (the passage of time), and
- The time-process displays generally provide some mechanism for displaying detailed system state for a particular instant in time.

One approach to combining the two would be simultaneously to present a time-process diagram in one window of a graphic workstation and an animation in another. The time-process display would guide the programmer to the important point in time, and the animation display would present detailed information about the state of the program in a comprehensible way. An alternative method of displaying both time and the detail found in animation frames would be to display several animation frames simultaneously. Using the abstraction of Belvedere, it might even be useful to display several different perspectives in different windows simultaneously. It may be that the flexibility of a system like Voyeur will be necessary because no single view is sufficient for the many different types of errors that must be addressed.

4. STATIC ANALYSIS FOR DEBUGGING PARALLEL PROGRAMS

When the probe effect renders the techniques in Sections 1 and 2 useless, what options are left to a programmer to debug a parallel program? Some researchers are pursuing static analysis techniques for detecting certain classes of anomalies in parallel programs. This is distinct from formal proof of correctness, because no attempt is made to prove conformance with a written specification. Instead, an attempt is made to give assurance that the program cannot enter certain predefined states that generally indicate errors.

Static analysis is being used to detect two classes of errors in parallel programs: *synchronization* errors and *data-usage* errors. Synchronization errors include such things as deadlock and wait forever. Data-usage errors include the usual sequential data-usage errors, such as reading an uninitialized variable, and parallel data-usage errors typified by two processes simultaneously updating a shared variable.

There appear to be two related but distinct areas being investigated. One is applying dataflow analysis techniques, similar to those used by optimizing and vectorizing compilers, to determine data-usage properties in parallel programs. The other is answering the question,

Is it possible for two statements S1 and S2 in a parallel program to execute in parallel?

These two areas are discussed in the remainder of this section. Some closely related work on combining static analysis with dynamic debugging and a system designed to aid in the development of parallel algorithms are presented at the end of Section 4.

4.1 Dataflow Analysis of Parallel Programs

Probably the most frequently referenced work on dataflow analysis of parallel programs is that of Taylor and Osterweil [1980]. Their algorithms generate four data-usage sets for each node of a program flow graph: *gen*, *kill*, *live*, and *avail*. These correspond to the sets by the same names used in the global dataflow analysis of optimizing compilers [Fosdick and Osterweil 1976; Hecht and Ullman 1975]. The original algorithms used to compute *live* and *avail* have been extended to pass data-usage information across edges in the flow graph corresponding to synchronization operations. By reinterpreting the meaning of *gen* and *kill*, it is possible to use the modified data-usage sets to arrive at algorithms to detect anomalies in parallel programs.

The algorithms in Taylor and Osterweil [1980] assume a simple process synchronization model. One process may cause another process to begin execution with the statement "schedule X" and wait for the completion of another process with "wait X". Their model does not permit a process to execute (be scheduled) in parallel with itself. This would correspond to a recursive process invocation. Recursion is also not allowed within any single process. They present algorithms based on their modified data-usage sets that operate on a representation of the program called a Process Augmented Flowgraph (PAF). This is constructed by taking the flowgraphs of the individual processes and connecting them with edges to indicate process synchronization constraints. For example, there would be an edge connecting the "schedule X" statement in one process with the initial statement in process X. Their algorithms can detect the following:

(1) a reference to an uninitialized variable,
(2) a variable that is referenced while being defined in parallel,
(3) a definition of a variable that is never referenced,
(4) a variable that may have an indeterminate value,
(5) a process waiting for the completion of an unscheduled process,
(6) a process waiting for the completion of another process that is guaranteed to have already completed, and
(7) a process that is scheduled to execute in parallel with itself.

In addition to not permitting recursion, the algorithm for item (2) assumes the existence of an algorithm for determining if two statements can execute in parallel. This is the subject of Section 4.2. Also, it is recognized that it is impossible to "create a fixed static procedure capable of constructing the PAF of any program written in a language which allows run-time determination of tasks to be scheduled and waited for."

The difficulty of using dataflow to analyze parallel programs is clearly shown in Callahan and Subhlok [1988]. They present an algorithm for determining which data dependencies present in a sequential execution of a program are preserved in a parallel execution of the program. They then show that determining if all data dependencies are maintained is Co-NP-hard using only the information found in their verison of the PAF which they call the *synchronized control flow graph*. They also present approximations that execute in polynomial time on programs written using a simple programming model. Two notable limitations of the model are that no synchronization operations are permitted within loops and all synchronization is done with event variables that cannot be cleared.

4.2 Parallel(*i*, *j*)

A Boolean function *parallel*(i, j), which returns true if it is possible for program points "i" and "j" to execute in parallel, can be used to detect *parallel access* errors. These occur when a variable is being read and written in parallel or when two processes can simultaneously write to the same variable. If the program can be represented as a Petri net [Peterson 1977], then this function can be implemented by examining the reachable states for the net. Unfortunately, the number of reachable states in a bounded Petri net grows exponentially with the number of places (nodes) in the net.

An algorithm similar to computing the reachable states in a Petri net applied to Ada programs is presented by Taylor [1983]. Like the Petri net algorithm, the number of states generated by Taylor's algorithm can increase exponentially with the number of parallel tasks in the Ada program.

An algorithm presented in McDowell [1989] computes *parallel*(i, j) for programs written in FORTRAN with extensions to support explicit parallelism. Whereas the simple language in Taylor and Osterweil [1980] explicitly prohibits the execution of a process with itself, the algorithm in McDowell [1989] uses the fact that many parallel numerical applications are expressed as collections of identical tasks executing in parallel on shared data. The result is that many fewer states are generated. This algorithm is being used in a prototype debugging tool [Appelbe and McDowell 1985].

A somewhat different approach to computing *parallel*(i, j) was taken in Bristow et al. [1979a]. Their algorithms operate on the same PAF representation of a program described in Section 4.1. They can build PAFs for the real language HAL/S. Although more powerful than the simple language used in Taylor and Osterweil [1980], HAL/S is still nonrecursive and contains relatively simple synchronization operators.

Instead of computing a single function, *parallel*(i, j), they compute eleven functions which they call execution sequence sets. Included are three execution sequence sets: *concurrent*, *always_concurrent*, and *possibly_concurrent*. The sets are computed for each node in the PAF. The set *concurrent* for node N contains all nodes M such that on all execution paths on which both M and N occur, they occur with no forced ordering. The set *always_concurrent* is a subset of *concurrent* that satisfies the additional restriction that all program execution paths containing N also contain M. Node M is in the set *possibly_concurrent* for node N if there is some execution path in which both M and N occur with no forced ordering between the two. For use in the anomaly detection algorithms of Taylor and Osterweil [1980], if a node M is in any of the above three execution sets at node N, then the nodes N and M would be assumed to execute in parallel. The result of this could be a potentially large number of extraneous anomaly reports corresponding to infeasible paths.

The three functions are actually represented as *execution sequence sets* attached to each node of the PAF. The algorithms for computing the execution sequence sets are very similar to the dataflow analysis algorithms mentioned in Section 4.1. The details of the algorithms can be found in Bristow et al. [1979b]. For the nonrecursive language HAL/S, the execution sequence sets can all be computed in polynomial time.

The result of applying the analysis mentioned above is an anomaly report. It should be possible to provide the user with sufficient information to determine what source statements are involved in the anomaly. There are, however, two problems related to presenting the anomaly report. One problem is that the anomaly report may contain many anomalies that are the result of infeasible paths and do not correspond to a real error in the program. Some progress in removing infeasible paths from static analysis of sequential programs has been reported [Werner 1988]. There is, however, nothing in the literature concerning the removal of infeasible paths from static analysis of concurrent programs.

The second problem with presenting the anomaly report is presenting the information in such a way that the user understands how the erroneous concurrent state could arise. It may not be sufficient to report that variable X is modified concurrently by a process executing line 100 and another process executing line 200. If the user cannot understand how lines 100 and 200 can execute in parallel, then it may be difficult to determine how to resolve the problem. Furthermore, the user may simply decide (erroneously) that this situation could never arise and that the anomaly report should be ignored.

The approach taken in Applebe and McDowell [1988] allows the user to examine not only the concurrency state causing

the anomaly report but also the concurrency states that led up to that state. A multiwindow user interface is provided that displays an anomalous concurrency state along with a description of the anomaly [McDowell 1988]. The concurrency state is represented by displaying a small portion of the source for each concurrent task in a separate window. The user may then display any previous or successor concurrency state to determine how the situation arose. This is somewhat like performing a coarse forward or backward simulation.

4.3 Combining Static Analysis with Dynamic Debugging

Taylor [1984] describes several ways in which static analysis could be productively combined with dynamic analysis. One approach would be to use the information from static analysis to help develop test data for use in conjunction with a dynamic debugger. Conversely, information from dynamic monitoring could be used to guide partial static analysis when complete static analysis would generate too many states. If subparts of a program could be shown to be free of errors using static analysis, then those portions of the program would not need monitoring. This could reduce the overhead associated with monitoring. If run-time assertion testing is included in the program, then the static analyzer could assume that the assertions are true, reducing the number of states that must be examined. A related technique is the use of symbolic execution to reduce the state space of a static analysis tool by eliminating infeasible paths [Young and Taylor 1986].

A somewhat different combined use of static and dynamic techniques is described in Allen and Padua [1987], Miller and Choi [1988b], and Stone [1989]. Each of these systems applies static analysis to a dynamically generated trace in order to identify parallel access anomalies that they call *races*. If a particular trace can be shown to be free of races, then the program is free of races for the given input. This does not mean that the program is free of races in general. For example, a race condition could be present in a conditionally executed block that is not executed with the given input. The analysis performed to identify races can also be used to help with breakpoint debugging. If the critical point in each process involved in a race can be identified, then a breakpoint can be placed just before that point in each process. This will stop the system in the state necessary to induce the race and permit close examination. In addition, by selectively continuing the processes, alternative race outcomes can be explored.

4.4 Static Analysis in the Development Process

In addition to analyzing parallel programs statically, and debugging them with run time debuggers and monitors, there is the possibility of eliminating the errors in programs before they occur. Here we would like to present some current work that seeks to aid in the development of parallel programs that are free of the kinds of errors outlined at the beginning of Section 4.

Automatic vectorizing compilers are the predecessors of the work presented here. They represent a very restricted form of parallel programs that are free of parallel bugs, assuming, of course, that the compilers are correct. This work has been extended most notably by Banerjee et al. [1979] to permit parallel execution of a wide range of loops. Again, assuming that both the sequential programs and the compilers are correct, the parallel programs that result will also be correct.

In addition to the fully automatic techniques of Banerjee et al., researchers at Rice University are working on a system called PTOOL [Allen et al. 1986]. PTOOL performs interprocess dataflow analysis to determine when a loop can be parallelized. It does this only for loops selected by the programmer. It interacts with the programmer for three reasons. First, the amount of time to perform the analysis for all loops and all combinations of loops is prohibitive. It is assumed that the programmer understands the overall structure of the program and knows which sections are most suitable for parallelization. Second, the programmer can make a judgment about the typical

values of certain variables at run time that affect the decision of whether to parallelize a particular loop. This is particularly important when the overhead for parallel execution is relatively high. Finally, by interacting with the programmer, PTOOL can provide information that might permit the programmer to change the program slightly, thereby allowing an important loop to be parallelized. In a fully automatic system the compiler would have to reject the loop as a candidate, possibly missing an important opportunity for parallel speedup.

By using automatic or semiautomatic techniques based on correctness preserving transformations, it is possible to debug a sequential version of a program using conventional debugging tools and then transform it into an equivalent parallel version.

5. CONCLUSION

Having completed the survey, the question remains, What progress has been made and where is more work needed? Because of the diversity of applications, languages, and systems, no single approach can satisfy all parallel debugging needs. The following paragraphs summarize what has been achieved and speculate on possible research directions.

The deficiencies of both static and dynamic techniques have been discussed in this paper. One promising approach that alleviates some of these deficiencies is the creation of a toolkit that integrates both approaches (see Section 4.3).

As the saying goes, "A picture is worth a thousand words." With program activities distributed across both space and time, simple sequential displays of program activity are inadequate. The time-process diagrams (see Section 3.2) give a compact view of the event history, whereas the animation diagrams (see Section 3.3) give a more detailed view of a single instant in time. Both representations are valuable, and the use of multiwindow workstations makes it possible to have both.

The appropriateness of each of these diagrams needs further research. For example, a major problem with the animation diagrams is the placement of the symbols representing the processes. Hough and Cuny [1987] make it clear that proper placement can be very important in comprehension of the display (see Figure 8).

In addition to the problem of placement is the problem of too much information—even for a picture. A possible solution is a language for abstracting low-level events into higher level events for display. Event description languages can also be used to filter out irrelevant events, reducing the amount of information that must be displayed.

One prominent feature of several systems is modularity [Joyce et al. 1987; Victor 1977]. By carefully designing a modular system, the addition or modification of various features can be managed easily. An event-based system might have modules for low-level event monitoring, filtering, and recording of events (from the low-level event modules), display of recorded events, analysis of recorded events, and controlled reexecution of the program. In traditional parallel debuggers as described in Section 1, there may be separate modules for interacting with the low-level machine and for interacting with the user. "Plug compatible" modules are advantageous because they allow experimentation with different debugger functions.

A well-defined interface (or hierarchy of interfaces) between the user and the low-level machine isolates most components from changes in any one part of the system. The user modules become machine independent; the low-level machine modules become user interface and language independent; and possibly some user interface modules may become language independent.

The probe effect is possibly the most significant difference between debugging parallel programs and sequential programs. The most obvious solution to the problem of the probe effect is to have the probes permanently in place. This does not help with breakpoint debugging, but it solves the problem for event-based debugging using monitoring and event histories. The

problem with this solution is the performance penalty for having software probes permanently enabled. The use of hardware assistance for high-level debugging was proposed in Gentleman and Hoeksma [1983], and systems using hardware monitoring for multiprocessors are described in Lazzerini and Prete [1986] and Rubin et al. [1988].

Before hardware designers will dedicate precious silicon to "hooks" for parallel debuggers, it will be necessary to identify just what hooks are useful. Having specified the hooks, a cost–benefit analysis could determine which low-level debugging elements should be implemented in hardware. Most uniprocessors today have hardware hooks for breakpointing and single stepping. It seems only natural that hooks for parallel debugging be added to parallel systems.

Event histories may be the most natural abstraction of distributed systems. A variety of tools and methods for examining them have been developed. For small and simple parallel programs, it may suffice to print the events as they occur. In larger systems, it may be preferable to save the event history for later examination. An alternative to examining the event history manually is to check the event history against a set of specifications as it is generated. Although this approach incurs a large overhead, it may be the only effective way to monitor large continually executing systems. Ideally, specifications from the program's design phase would be used to detect errors. In current systems, the programmer must write the specifications for checking the event history. This is usually done as part of a testing phase and is not directly connected to any design specification.

The event specification languages we have seen [Baiardi et al. 1986; Harter et al. 1985; Helmbold and Luckham 1985b] can all express simple constraints on the event stream. It is unreasonable, however, to expect the programmer to specify completely the intended behavior of a program with these languages. Much work needs to be done before we know what kinds of assertions are most useful and the best languages for expressing them. Furthermore, the issue of integrating design specifications with run-time checking has not been addressed.

A variation of the specification approach is taken by EDL [Bates and Wileden 1983]. They do not check the event history against specifications but instead transform it into a higher level history. This approach may help bridge the gap between low-level communication primitives and the more abstract communication mechanisms used in the program. EDL has been successfully used to control the presentation of graphic data in the Belvedere system [Hough and Cuny 1987].

We have seen some attempts at software solutions to the probe effect. These involve some mechanism for the debugger to manipulate the logical passage of time. In none of the systems surveyed was this completely successful; real-time events do not lend themselves well to manipulations of logical time. Although appearing unsolvable in general, software solutions might be attainable for some systems such as message-passing systems without timeouts. It certainly appears that designers of new parallel languages and synchronization constructs should keep the probe effect in mind.

Perhaps the best way to avoid the bugs associated with current parallel constructs is to design languages for parallel machines that make such errors impossible. Dataflow and functional languages are one example of systems that attempt to "define away the problem." Still other examples can be found in declarative languages such as Prolog or higher level languages as described in Goldberg [1986]. A somewhat less radical approach is the use of tools for automatically detecting parallelism. These may be fully automatic or require some user interaction. In either case, such systems would ensure that the parallel program produces exactly the same results as the sequential version.

ACKNOWLEDGMENTS

We would like to thank Anil Sahai who contributed to an early version of this survey. We would also like to thank the referees for their comments.

APPENDIX A. SUMMARY TABLES

The tables in this appendix present brief descriptions of the systems surveyed in a form that permits quick comparisons. To make it possible to place as much information as possible in each summary table, we use one- or two-word descriptors in the tables. Informal explanations of the descriptors are given before the tables.

A.1 General Characteristics Part 1

O.S. Operating system debugger runs under

Hardware Hardware configuration debugger uses or requires

Status Completeness of debugger implementation

- partial — Prototype missing major features
- production — Production version available
- prototype — Complete in-house system
- n/s — Not specified

Table A.1. General Characteristics Part 1

System		O.S.	Hardware	Status
Agora	[For88]	Agora	LAN	prototype
Amoeba	[Els88]	Amoeba	n/s	prototype
belvedere	[HC87]	simple sim	emulator	partial
BUGNET	[CW82]	MICROS	MICRONET	partial
CBUG	[Gai85]	UNIX	any UNIX	prototype
cdbg	[Int87]	iPSC	iPSC	prototype
dbxtool	[AM86]	UNIX(Sun)	Sun	production
defence	[Web83]	n/s	uniprocessor	partial
DISDEB	[LP86]	Mara	Mara	prototype
EDL	[BW83]	VMT UMass	VMT UMass	partial
HARD	[MCR85]	UNIX	any UNIX	prototype
IDD	[HHK85]	UNIX	network(Sun)	partial
Instant	[LM87]	Chrysalis	BBN butterfly	prototype
Jade	[JLSU87]	Jipc	vax/Sun	prototype
MAD	[RRZ88]	n/s	mimd sh. bus	prototype
Meglos	[GK86]	UNIX	MC68000	production
mtdbx	[Gri87]	UNIX/COS	Sun/Cray	prototype
Multibug	[CP86]	n/s	multi-macro-proc	prototype
Parasight	[AG88]	Mach/Umax	Multimax	prototype
pdbx	[Seq86]	Dynix	Sequent	production
Pilgram	[Coo87]	Mayflower	Cambridge DCS	partial
PPD	[MC88b]	n/s	n/s	partial
RADAR	[LR85]	n/s	PERQ	prototype
Recap	[PL88]	n/s	n/s	proposed
Traveler	[Man87]	Apiary	emulator	prototype
TSL	[HL85b]	any	any	prototype
Voyeur	[SBN88]	n/s	n/s	prototype
YODA	[LP85]	n/s	n/s	prototype
	[AP87]	Cedar	n/s	partial
	[BDV86]	MuTEAM	MuTEAM	partial
	[GB85]	PathPascal	n/s	partial
	[GGK84]	n/s	n/s	prototype
	[GKY88]	any	any	prototype
	[MMS86]	UNIX	any UNIX	prototype
	[Sno84]	Medusa/StarOS	Cm*	prototype

A.2 General Characteristics Part 2

Interface How the debugger hooks into the program
- hardware: Additional hardware is used instead of a software interface
- manual: Calls to the debugger are manually inserted by the programmer
- object: Compiler modifies the object code
- oper sys: Debugger interacts with the normal operating system
- source: Automatic insertion of source code statements calling the debugger

Probe Effect How the debugger addresses the Probe Effect
- fast calls: Fast monitoring operations minimize the effect on program timing
- leave in: Leave the debugger in the system
- logical time: Logical time hides the effects of debugging operations

Global Clock Whether debugger requires a global clock
- assumed: Debugger assumes the existence of an accurate global clock
- self-timed: Debugger simulates its own global time
- uniprocessor: Debugger is for a single processor system or simulation

Languages Languages the debugger supports

Model The model of communication
- block-send: Message passing with blocking sends
- gmem: A bank of global memory equidistant from all processors
- hybrid: Has both shared memory and message passing
- lmem: The shared memory is located at the processors
- messages: Message passing
- rndzv: Ada rendezvous
- rpc: Remote procedure call

n/s: Not specified

Table A.2. General Characteristics Part 2

System		Interface	Probe Effect	Global Clock	Languages	Model
Agora	[For88]	oper sys	leave in	self-timed	n/s	lmem
Amoeba	[Els88]	object	n/s	none	n/s	messages
belvedere	[HC87]	object	logical time	uniprocessor	Simple Simon	messages
BUGNET	[CW82]	object	n/s	self-timed	Modula2	messages
CBUG	[Gai85]	source	fast calls	none	C	gmem
cdbg	[Int87]	oper sys	n/s	none	C, Ftn	messages
dbxtool	[AM86]	oper sys	n/s	none	C, Pscl, Ftn	messages
defence	[Web83]	object	n/s	uniprocessor	Conc. Euclid	monitors
DISDEB	[LP86]	hardware	none[a]	none	any	gmem + lmem
EDL	[BW83]	n/s	n/s	assumed	n/s	n/s
HARD	[MCR85]	source	logical time	assumed	Ada	rndzv + gmem
IDD	[HHK85]	object	n/s	none	C, Modula2	messages
Instant	[LM87]	object	leave in	none	several	gmem
Jade	[JLSU87]	object	n/s	none	several	block-send
MAD	[RRZ88]	man + hard	leave in	assumed	PARC(C)	gmem
Meglos	[GK86]	source	n/s	none	C	messages
mtdbx	[Gri87]	object	logical time	uniprocessor	f77 + Cray	gmem
Multibug	[CP86]	oper sys	n/s	none	low level	messages
Parasight	[AG88]	oper sys	fast calls	none	C	gmem

[a] DISDEB uses additional hardware to eavesdrop on the network traffic. This allows the DISDEB debuggers to run transparently, without disturbing the program's timing.

(*continued*)

Table A.2. (*Continued*)

System		Interface	Probe Effect	Global Clock	Languages	Model
pdbx	[Seq86]	oper sys	n/s	none	C, P, F + dynix	gmem
Pilgram	[Coo87]	object	logical time	none	Conc. Clu	rpc
PPD	[MC88b]	object	n/s	none	C	gmem
RADAR	[LR85]	object	n/s	none	Pronet	messages
Recap	[PL88]	object	n/s	none	n/s	hybrid
Traveler	[Man87]	oper sys	n/s	uniprocessor	Acore(lisp)	messages
TSL	[HL85b]	source	n/s	none	Ada	rndzv
Voyeur	[SBN88]	manual	n/s	assumed	several	hybrid
YODA	[LP85]	source	n/s	assumed	Ada	rndzv + gmem
	[AP87]	oper sys	n/s	none	FORTRAN	gmem
	[BDV86]	object	logical time	none	ECSP	messages
	[GB85]	oper sys	n/s	none	Path Pascal	gmem
	[GGK84]	oper sys	none	none	n/s	messages
	[GKY88]	source	fast calls	none	Occam, NIL	messages
	[MMS86]	oper sys	fast calls	none	C	messages
	[Sno84]	object	n/s	assumed	n/s	hybrid

A.3 User Interface

Exam/Mod State — Capabilities for examining/modifying the program's state

- global, glb — Global state can be examined
- ipc — Communication state can be examined
- local — Local states can be examined
- + — Modification of state is also possible
- sequent — (Plans to) interface with a sequential debugger

Rename Objects — Whether program objects are given special names during debugging

- no — No objects can be given names
- procs — Only processes can be given names
- yes — Most objects can be given names

Graphics — Whether debugger uses graphics

- commun — Animated view of interprocess communications
- tp — Time-process diagrams can be displayed
- windows, win — Processes are displayed in separate windows

Exam Event History — How user examines a recorded event history

- browser — Using an editor/browser
- ⟨language⟩ — Using queries in the indicated language
- replay — Only examination is to replay the history
- scroll tp — Scrollable time-process diagrams

Event Lang — What language (if any) is used to express patterns of events

Control Sched — Whether user can control scheduling of the program

- hist — History guides to scheduling
- select, sel — Can select which process to run next
- sus/cont, sc — Can suspend or continue individual processes

- n/s — Not specified

Table A.3. User Interface

System		Exam/Mod State	Rename Objects	Graphics	Examine Event History	Event Lang	Control Sched
Agora	[For88]	local	no	windows	replay	none	sus/cont
Amoeba	[Els88]	local+	no	none	replay	reg. exp.	sus/cont
belvedere	[HC87]	ipc	no	commun	replay	EDL	hist
BUGNET	[CW82]	local, ipc	no	none	replay	none	sus/cont
CBUG	[Gai85]	local, ipc	no	windows	none	none	sus/cont
cdbg	[Int87]	local+	yes	none	none	none	sus/cont
dbxtool	[AM86]	local+	no	windows	none	none	sus/cont
defence	[Web83]	local+	no	none	none	none	sus/cont
DISDEB	[LP86]	global+	no	none	none	[c]	sus/cont
EDL	[BW83]	n/s	no	none	replay	EDL	none
HARD	[MCR85]	glb+, ipc+	no	none	none	Ada[d]	sus/cont
IDD	[HHK85]	local, ipc	no	tp	scroll tp	int. log.	sc, hist
Instant	[LM87]	sequent	no	tp[a]	replay	none	hist
Jade	[JLSU87]	sequent	yes	tp, commun	browser	none	sel, hist
MAD	[RRZ88]	global	no	tp	browser	path rules	n/s
Meglos	[GK86]	local+	no	none	none	none	sus/cont
mtdbx	[Gri87]	global+	no	tp, win	scroll tp	none	sc, sel, hist
Multibug	[CP86]	local+	yes	none	none	none	sus/cont
Parasight	[AG88]	local+	no	none	none	none	sus/cont
pdbx	[Seq86]	local+	no	windows	none	none	sus/cont
Pilgram	[Coo87]	global+	no	none	none	none	sus/cont
PPD	[MC88b]	local, ipc	no	[b]	replay	none	sus/cont
RADAR	[LR85]	ipc	no	commun	replay	none	none
Recap	[PL88]	sequent	no	none	replay	none	hist
Traveler	[Man87]	n/s	no	windows	browser	none	select
TSL	[HL85b]	sequent	yes	none	browser	TSL	select
Voyeur	[SBN88]	global	yes	programmable	browser	none	none
YODA	[LP85]	ipc	no	none	prolog	none	none
	[AP87]	n/s	no	none	none	none	none
	[BDV86]	sequent	no	none	none	BS	none
	[GB85]	global+	no	win, commun	replay	yes, n/s	sus/cont
	[GGK84]	local+	no	none	scroll tp	yes, n/s	sus/cont
	[GKY88]	glb+, ipc+	no	windows	browser	temp. logic	sc, select
	[MMS86]	n/s	no	none	browser	none	none
	[Sno84]	local	no	none	TQuel	TQuel	none

[a] In Fowler et al. [1988] a toolkit by the same authors includes process time diagrams that require a global clock. The process time diagrams are not presented in the 1987 paper.

[b] PPD generates and displays dynamic dependence graphs.

[c] DISDEB allows complex events to be built out of very low-level machine code like events using a low-level language. For example, the language can only refer to physical addresses rather than using identifiers from the source code.

[d] There is a facility for calling the debugger from special tasks. These tasks can be used to implement arbitrarily complex breakpoints.

A.4 Breakpoints

State Breakpoints Types of state-based breakpoints supported

global Breakpoints can be set on global (and local) state

local Breakpoints can be set on local state

stmt Breakpoints can be set at a source statement

Event Breakpoints Types of event-based breakpoints

mult.⟨language⟩	Breakpoints on conjunction, disjunction, or repetition of events
seq.⟨language⟩	Breakpoints on complex sequence of events
single	Breakpoints on the occurrence of single events
Modify Breakpoints	Whether breakpoints can be added/disabled

Breakpoint Effect	What is halted when a breakpoint is reached during program execution
either	Either one process or the entire program may be halted
process	One process is halted
program	The entire program is halted
n/a	Not applicable
n/s	Not specified

Table A.4. Breakpoints

System		State Breakpoints	Event Breakpoints	Modify Breakpoints	Breakpoint Effect
Agora	[For88]	local	single	yes	process
Amoeba	[Els88]	local + stmt	seq(reg. expr.)	yes	either
belvedere	[HC87]	no	none	no	n/a
BUGNET	[CW82]	n/s	single	yes	either
CBUG	[Gai85]	stmt	none	yes	process
cdbg	[Int87]	local + stmt	single	yes	process
dbxtool	[AM86]	local + stmt	none	yes	process
defence	[Web83]	local + stmt	none	yes	program
DISDEB	[LP86]	no	multiple	no	either
EDL	[BW83]	no	none	n/a	n/a
HARD	[MCR85]	local + stmt	multiple	remove	process
IDD	[HHK85]	global	seq	n/s	program
Instant	[LM87]	local + stmt	none	yes	program
Jade	[JLSU87]	no	single	yes	process
MAD	[RRZ88]	n/s	n/s	n/a	n/a
Meglos	[GK86]	local	single	yes	either
mtdbx	[Gri87]	local + stmt	none	yes	either
Multibug	[CP86]	local + stmt	single	yes	process
Parasight	[AG88]	stmt	none	yes	process
pdbx	[Seq86]	local + stmt	none	yes	process
Pilgram	[Coo87]	stmt	none	yes	process
PPD	[MC88b]	stmt + local	none	yes	program
RADAR	[LR85]	no	none	n/a	n/a
Recap	[PL88]	local + stmt	n/s	yes	process
Traveler	[Man87]	no	no (planned)	n/a	n/a
TSL	[HL85b]	no	seq(TSL)	no	process
Voyeur	[SBN88]	n/s	n/s	n/s	n/s
YODA	[LP85]	no	none	n/a	n/a
	[AP87]	n/a	n/a	n/a	n/a
	[BDV86]	local	seq(BS)	no	process
	[GB85]	no	seq	yes	process
	[GGK84]	local + global	seq	yes	either
	[GKY88]	stmt + global	single	no	program
	[MMS86]	no	none	n/a	n/a
	[Sno84]	no	none	n/a	n/a

A.5 Event Monitoring

Event Type	What is an event
ipc	Every (explicit) interprocess communication
sh mem	Shared memory references
stmt	Each statement execution
History	Kind of event history recorded
buffer	Last *n* events are stored in a buffer
chk pt	All events since the last checkpoint are saved
complete	All events are recorded and preserved
sparse	Some events are kept; others are not
Filtering	How the information recorded (or replayed) can be reduced
event	Using predicates on single events
global	Global state
history	Using predicates on the history of events
⟨language⟩	Specified language describes "interesting" events
local	Local state
process	Specifying important process(es)
Replay	How complete are the replay facilities
commun	Communication state can be deduced
complete	Entire state (including local vars) is available
Ordering	How are event histories ordered
linear	All events are forced into a linear history
partial	"Concurrent" events are not ordered
n/a	Not applicable
n/s	Not specified

Table A.5. Event Monitoring

System		Event Type	History	Filtering	Replay	Ordering
Agora	[For88]	sh mem	chk. pt.	n/s	complete	partial
Amoeba	[Els88]	ipc	chk. pt.	history	complete	partial
belvedere	[HC87]	ipc	complete	none	commun	partial
BUGNET	[CW82]	ipc	chk. pt.	proc, event	complete	linear
CBUG	[Gai85]	ipc	none	none	none	n/a
cdbg	[Int87]	ipc	none	n/a	n/a	n/a
dbxtool	[AM86]	stmt	none	none	none	n/a
defence	[Web83]	stmt	none	n/a	n/a	n/a
DISDEB	[LP86]	ipc, sh mem	none	[a]	none	linear
EDL	[BW83]	n/s	complete	EDL	commun	linear
HARD	[MCR85]	stmt	none	none	none	partial
IDD	[HHK85]	ipc	buffer	proc, event	none	linear
Instant	[LM87]	sh mem	complete	none	complete	partial
Jade	[JLSU87]	ipc	complete	proc, event	complete	linear
MAD	[RRZ88]	stmt	sparse	history	none	linear
Meglos	[GK86]	sh mem	none	none	none	partial
mtdbx	[Gri87]	ipc	complete	none	complete	linear
Multibug	[CP86]	ipc	none	n/a	n/a	n/a
Parasight	[AG88]	stmt	none	none	none	n/a
pdbx	[Seq86]	stmt	none	none	none	n/a
Pilgram	[Coo87]	stmt	none	none	none	n/a
PPD	[MC88b]	sh mem	complete	none	none	partial
RADAR	[LR85]	ipc	complete	none	complete	partial
Recap	[PL88]	ipc, sh mem	chk. pt.	process	complete	partial

[a] DISDEB allows complex events to be built out of very low-level machine code like events using a low-level language. For example, the language can only refer to physical addresses rather than using identifiers from the source code.

[b] Filtering is done by *transactions*. The nested function calls can be hidden, giving a clearer picture of the high-level activity (see Section 3.1).

(*continued*)

Table A.5. (*Continued*)

System		Event Type	History	Filtering	Replay	Ordering
Traveler	[Man87]	ipc	complete	[b]	none	partial
TSL	[HL85b]	ipc	complete	TSL	(planned)	linear
Voyeur	[SBN88]	stmt	complete	n/s	none	linear
YODA	[LP85]	ipc, sh mem	complete	none	none	linear
	[AP87]	ipc, sh mem	sparse	none	none	partial
	[BDV86]	ipc	none	n/a	n/a	n/a
	[GB85]	ipc	complete	n/s	commun	partial
	[GGK84]	n/s	complete	suggested	complete	partial
	[GKY88]	ipc, stmt	complete	history	complete	linear
	[MMS86]	ipc	complete	event	none	linear
	[Sno84]	stmt	sparse	event	none	linear

REFERENCES

[ABKP86] ALLEN, R., BAUMGARTNER, D., KENNEDY, K., AND PORTERFIELD, A. 1986. Ptool: A semiautomatic parallel programming assistant. In *Proceedings of the International Conference on Parallel Processing.* IEEE, pp. 164–170.

[AG88] ARAL, Z., AND GERTNER, I. 1988. High-level debugging in parasight. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 151–162.

[AM85] APPELBE, W. F., AND MCDOWELL, C. E. 1985. Anomaly reporting: A tool for debugging and developing parallel numerical algorithms. In *Proceedings of the 1st International Conference on Supercomputing Systems.* IEEE, pp. 386–391.

[AM86] ADAMS, E., AND MUCHNICK, S. S. 1986. Dbxtool: A window-based symbolic debugger for sun workstations. *Softw. Pract. Exper. 16,* 7, 653–669.

[AM88] APPELBE, W. F., AND MCDOWELL, C. E. 1988. Developing multitasking applications programs. In *Proceedings of Hawaii International Conference on System Sciences.* IEEE, pp. 94–101.

[AP87] ALLEN, T. R., AND PADUA, D. A. 1987. Debugging FORTRAN on a shared memory machine. In *Proceedings of the International Conference on Parallel Processing.* Penn State University, pp. 721–727.

[Bat88] BATES, P. 1988. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 11–22.

[BCKT79] BANERJEE, U., CHEN, S., KUCK, D. J., AND TOWLE, R. A. 1979. Time and parallel processor bounds for fortran-like loops. *IEEE Trans. Comput. 28,* 9 (Sept.), 660–670.

[BDER79a] BRISTOW, G., DREY, C., EDWARDS, B., AND RIDDLE, W. 1979. Anomaly detection in concurrent programs. In *Proceedings of the 4th International Conference on Software Engineering.* IEEE.

[BDER79b] BRISTOW, G., DREY, C., EDWARDS, B., AND RIDDLE, W. 1979. Design of a system for anomaly detection in HAL/S programs. Tech. Rep. CU-CS-151-79. Univ. of Colorado at Boulder.

[BDV86] BAIARDI, F., DEFRANCESCO, N., AND VAGLINI, G. 1986. Development of a debugger for a concurrent language. *IEEE Trans. Softw. Eng. SE-12,* 4 (Apr.), 547–553.

[BW83] BATES, P. C., AND WILEDEN, J. C. 1983. High-level debugging of distributed systems: The behavioral abstraction approach. *J. Syst. Softw. 3,* 255–264. Also COINS Tech. Rep. #83-29.

[CL85] CHANDY, K. M., AND LAMPORT, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3,* 1 (Feb.), 63–75.

[Coo87] COOPER, R. 1987. Pilgram: A debugger for distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems.* IEEE, pp. 458–465.

[CP86] CORSINI, P., AND PRETE, C. A. 1986. Multibug: Interactive debugging in distributed systems. *IEEE Micro 6,* 3, 26–33.

[CS88] CALLAHAN, D., AND SUBHLOK, J. 1988. Static analysis of low-level synchronization. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 100–111.

[CW82] CURTIS, R. S., AND WITTIE, L. D. 1982. BugNet: A debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* ACM, pp. 394–399.

[Els88] ELSHOFF, I. J. P. 1988. A distributed debugger for amoeba. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 1–10.

[Fid88] FIDGE, C. J. 1988. Partial orders for parallel debugging. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 183–194.

[FLM88] FOWLER, R. J., LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. 1988. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 163–173.

[FO76] FOSDICK, L. D., AND OSTERWEIL, L. J. 1976. Data flow analysis in software reliability. *ACM Comput. Surv. 8* (Sept.), 305–330.

[For88] FORIN, A. 1988. Debugging of heterogeneous parallel systems. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 130–140.

[Gai85] GAIT, J. 1985. A debugger for concurrent programs. *Softw. Pract. Exper. 15*, 6, 539–554.

[GB85] GARCIA, M. E., AND BERMAN, W. J. 1985. An approach to concurrent systems debugging. In *Proceedings of the 5th International Conference on Distributed Computing Systems.* IEEE, pp. 507–514.

[GGK84] GARCIA-MOLINA, H., GERMANO, F., JR., AND KOHLER, W. H. 1984. Debugging a distributed computing system. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar.), 210–219.

[GH83] GENTLEMAN, W. M., AND HOEKSMA, H. 1983. Hardware assisted high level debugging. *SIGPLAN Notices 18*, 8 (August), 140–144.

[GK86] GAGLIANELLO, R. D., AND KATSEFF, H. P. 1986. The meglos user interface. In *Proceedings of Fall Joint Computer Conference.* ACM, pp. 169–177.

[GKY88] GOLDSZMIDT, G., KATZ, S., AND YEMINI, S. 1988. Interactive blackbox debugging for concurrent languages. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 271–282.

[Gol86] GOLDBERG, A. T. 1986. Knowledge-based programming: A survey of program design and construction techniques. *IEEE Trans. Softw. Eng. SE-12*, 4 (Apr.), 752–768.

[GR85] GEHANI, N. H., AND ROOME, W. D. 1985. Concurrent C. Tech. Rep., AT&T Bell Laboratories.

[Gri87] GRIFFIN, J. 1987. Parallel debugging system user's guide. Tech. Rep., Los Alamos National Laboratory.

[HC87] HOUGH, A. A., AND CUNY, J. 1987. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the International Conference on Parallel Processing.* Penn State University, pp. 735–738.

[HHK85] HARTER, P. K., JR., HEIMBIGNER, D. M., AND KING, R. 1985. IDD: An interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems.* IEEE, pp. 498–506.

[HL85a] HELMBOLD, D., AND LUCKHAM, D. 1985. Debugging ada tasking programs. *IEEE Softw. 2*, 2, 47–57.

[HL85b] HELMBOLD, D., AND LUCKHAM, D. 1985. TSL: Task Sequencing Language. In *Ada In Use, Proceedings of the Ada International Conference.* ACM, Cambridge University Press.

[HU75] HECHT, M. S., AND ULLMAN, J. D. 1975. A simple algorithm for global data flow analysis problems. *SIAM J. Comput. 4*, 519–532.

[HW88] HABAN, D., AND WEIGEL, W. 1988. Global events and global breakpoints in distributed systems. In *Proceedings of Hawaii International Conference on System Sciences.* IEEE, pp. 166–175.

[Int87] INTEL CORP. 1987. *iPSC Concurrent Debugger Manual.*

[JLSU87] JOYCE, J., LOMOW, G., SLIND, K., AND UNGER, B. 1987. Monitoring distributed systems. *ACM Trans. Comput. Syst. 5*, 2 (May), 121–150.

[Kar87] KARP, A. H. 1987. Programming for parallelism. *Computer 20*, 5, 43–57.

[Lam78] LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 558–565.

[LM87] LEBLANC, T. J., AND MELLOR-CRUMMEY, J. M. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput. C-36*, 4 (Apr.), 471–482.

[LP85] LEDOUX, C. H., AND PARKER, D. S., JR. 1985. Saving traces for ada debugging. In *Ada In Use, Proceedings of the Ada International Conference.* ACM, Cambridge University Press, pp. 97–108.

[LP86] LAZZERINI, B., AND PRETE, C. A. 1986. Disdeb: An interactive high-level debugging system for a multi-microprocessor system. *Microprocess. Microprogram. 18*, 401–408.

[LR85] LEBLANC, R. J., AND ROBBINS, A. D. 1985. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems.* IEEE, pp. 515–522.

[Man87] MANNING, C. R. 1987. Traveler: The apiary observatory. In *Proceedings of European Conference on Object Oriented Programming.* pp. 97–105.

[MC88a] MILLER, B. P., AND CHOI, J.-D. 1988a. Breakpoints and halting in distributed systems. In *Proceedings of International Conference on Distributed Computing Systems.* IEEE.

[MC88b] MILLER, B. P., AND CHOI, J.-D. 1988. A mechanism for efficient debugging of parallel programs. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM, pp. 141–150.

[McD88] MCDOWELL, C. E. 1988. Viewing anomalous states in parallel programs. In *Proceedings of the International Conference on Parallel Processing.* Penn State University, pp. 54–57.

[McD89] MCDOWELL, C. E. 1989. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing 6*, 3 (June), 515–536.

[MCR85] DI MAIO, A., CERI, S., AND REGHIZZI, S. C. 1985. Execution monitoring and debugging tool for ada using relational algebra. In *Ada In Use, Proceedings of the Ada International Conference.* ACM, Cambridge University Press.

[MMS86] MILLER, B. D., MACRANDER, C., AND SECHREST, S. 1986. A distributed programs monitor for Berkeley UNIX. *Softw. Pract. Exper. 16*, 2, 183–200.

[Pet77] PETERSON, J. L. 1977. Petri nets. *ACM Comput. Surv. 9*, 3 (Sept.), 223–252.

[PL88] PAN, D. Z., AND LINTON, M. A. 1988. Supporting reverse execution of parallel programs. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM. Published as *SIGPLAN Notices 24*, 1 (January 1989). pp. 124–129.

[RRZ88] RUBIN, R. V., RUDOLPH, L., AND ZERNIK, D. 1988. Debugging parallel programs in parallel. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM. Published as *SIGPLAN Notices 24*, 1 (January 1989). pp. 216–225.

[SBN88] SOCHA, D., BAILEY, M. L., AND NOTKIN, D. 1988. Voyeur: Graphical views of parallel programs. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM. Published as *SIGPLAN Notices 24*, 1 (January 1989). pp. 206–215.

[Seq86] SEQUENT CORP. 1986. *Dynix Pdbx Parallel Debugger User's Manual.*

[SG86] SCHEIFLER, R. W., AND GETTYS, J. 1986. The X window system. *ACM Trans. Graph. 5*, 2 (Apr.).

[Sno84] SNODGRASS, R. 1984. Monitoring in a software development environment: a relational approach. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments.* SIGPLAN, ACM SIGSOFT.

[ST83] SEIDNER, R., AND TINDALL, N. 1983. Interactive debug requirements. *SIGPLAN Notices* 9–22.

[Sto88] STONE, J. M. 1988. A graphical representation of concurrent processes. In *Proceedings of Workshop on Parallel and Distributed Debugging.* ACM. Published as *SIGPLAN Notices 24*, 1 (January 1989). pp. 226–235.

[Sun86] SUN MICROSYSTEMS. 1986. *NeWS Preliminary Technical Overview.*

[Tan81] TANENBAUM, A. S. 1981. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J.

[Tay83] TAYLOR, R. N. 1983. A general-purpose algorithm for analyzing concurrent programs. *CACM 26*, 5, 362–376.

[Tay84] TAYLOR, R. N. 1984. Debugging real-time software in a host-target environment. Tech. Rep. 212, Univ. of California at Irvine.

[TO80] TAYLOR, R. N., AND OSTERWEIL, L. J. 1980. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. Softw. Eng. SE-6*, 3 (May), 265–278.

[Vic77] VICTOR, K. E. 1977. The design and implementation of DAD, a multiprocess, multimachine, multilanguage interactive debugger. In *Proceedings of Hawaii International Conference on System Sciences.* IEEE, pp. 196–199.

[Web83] WEBER, J. C. 1983. Interactive debugging of concurrent programs. *SIGPLAN Notices 18*, 8, 112–113.

[Wer88] WERNER, L. L. 1988. Fault detection in production programs by means of data usage analysis. Ph.D. dissertation UCSD.

[YT86] YOUNG, M., AND TAYLOR, R. N. 1986. Combining static concurrency analysis with symbolic execution. In *Proceedings of Workshop on Software Testing.* pp. 10–178.

Received July 1988; final revision accepted January 1989.