Chapter 5

Debugging of Parallel and Distributed Programs

José C. Cunha, João Lourenço and Vítor Duarte

Abstract

This chapter surveys the main issues involved in correctness debugging of parallel and distributed programs. Distributed debugging is an instance of the more general problem of observation of a distributed computation. This chapter briefly summarizes the theoretical foundations of the distributed debugging activity. Then a survey is presented of the main methodologies used for parallel and distributed debugging, including state and event based debugging, deterministic re-execution, systematic state exploration, and correctness predicate evaluation. Such approaches are complementary to one another, and the chapter discusses how they can be supported using distinct techniques for observation and control.

5.1 Introduction

The correctness debugging activity is first discussed within the general context of software development, and then the main characteristics of debugging of sequential programs are outlined. Finally, this section discusses the main dimensions involved in the debugging of parallel and distributed programs. Throughout this chapter, the term *distributed debugging* (DD) is used to refer to the debugging of parallel and distributed programs.

5.1.1 The debugging activity

Given an application, one typically reasons in relation to some specification of its intended behavior. Ideally, such specification would be expressed by a formal notation to ensure or assess the correctness of any implementation of the application in terms of a programming model. One would like to describe the application using a formal specification that would automatically generate correct (and efficient) program code. In such a case, bugs could only appear at the level of the application specification, in relation to its intended behavior (*specification bugs*). Given the general lack of automated code generation tools for high level formal specification languages, a programmer becomes responsible for the mapping from the formal specification to the program code. Even if the clarity

and expressiveness of each programming model can greatly contribute to ease the mentioned mapping, such an activity gives the opportunity to introduce another kind of bugs, at the level of program design and implementation (*programming bugs*). These ideas are shown in Figure 5.1 below.

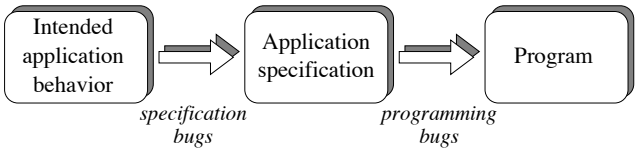


Figure 5.1: Specification and programming bugs

In common practice, however there is typically no complete application formal specification available so the correctness of the program can only be assessed in relation to the intended application behavior, which only exists in the mind of the developer (to be optimistic...). This makes the debugging activity extremely complex, as specification and programming bugs appear at the same level, embedded into the program code.

Below the program level one would also have to consider the mapping to the operating system and machine code levels as potentially contributing to the appearance of other kind of bugs. However, in general, such bugs are beyond the reach of the application developer and programmer.

Due to the above difficulties, debugging becomes a fundamental activity even if it is not a desired one.

In the past fifty years, there was a huge amount of work concerning the debugging of sequential applications. Several significant debugging techniques were developed, addressing both specification and programming bugs, depending on the abstractions and paradigms of the programming languages (e.g., imperative or declarative) [16]. A sequential application is executed by a sequential system so a state-based approach is adequate to analyze its behavior. Interactive debuggers allow the user to inspect the succession of states followed by a sequential computation. Due to the deterministic behavior of a sequential system, it is easy to re-execute the program under a given set of input conditions in order to examine its behavior in detail. Even if the programming language exhibits some form of internal nondeterminism in its computation strategy, e.g., like Prolog, it is possible to apply the cyclic state-based debugging technique in order to repeatedly examine the deterministic execution path followed by the sequential Prolog executor. Sequential debugging is also made simpler because one has only to think about one thread of control at each point during a debugging session. The large number of states that may be generated for a sequential computation is easily handled by placing breakpoints at desired conditions or regions of code and having the debugger stopping the execution. Such an external control by a debugger has no logical effect upon the sequential computation behavior.

The increased complexity of developing parallel and distributed applications makes it more difficult to use the above approach for debugging.

5.1.2 Distributed debugging

In this chapter, a *distributed program* (DP) consists of a collection of sequential processes which cooperate by using some distributed-memory communication model. The term *distributed* is emphasized because such distributed processes cannot rely on physically shared memory or global clock abstractions for synchronization purposes. Parallel applications are naturally included in this

concept, when distributed-memory systems are considered, such as the ones based on cluster computing platforms. A distributed program is based upon the semantics of the specific programming language that is used to express the concurrency, distribution and parallelism, and communication between processes. A distributed program is executed in the context of a *distributed system* (DS). A distributed system provides the computational mechanisms to support the execution of a distributed program, in terms of a virtual architecture defined by the operating system and the hardware platform.

Given a distributed program, one would like to be able to specify its correctness properties in term of predicates on the expected program behavior and then having a distributed debugger automatically comparing them to the observed program behavior (see Figure 5.2).

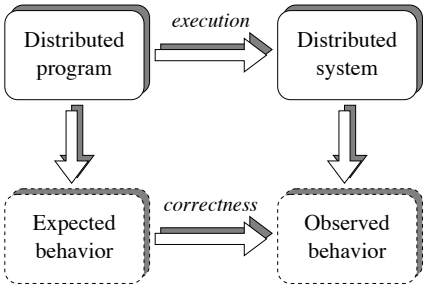


Figure 5.2: Expected and observed behaviors

Specification of desired properties would be possible, as well as detection of undesired or erroneous situations corresponding to programming bugs. This would require the distributed debugger to support three distinct aspects:

- A language to specify the correctness predicates.
- Algorithms to evaluate those predicates.
- Observation and control mechanisms to allow the user to observe the computation states corresponding to the detected erroneous situations.

Such an ideal picture is difficult to achieve in an asynchronous distributed system with no global clock, no global shared memory, and no bounds on message transmission times. The following dimensions make distributed debugging much more difficult than sequential debugging [36]:

1. The large number of parallel and distributed entities, with dynamic interactions in a distributed program.
2. The intrinsic non-deterministic behavior of a distributed program.
3. The difficulties of constructing accurate, up-to-date, and consistent observations of the global states of an asynchronous distributed system.
4. The intrusion effect due to the observation and control mechanisms.

The first dimension (*distributed dynamic interacting entities*) must consider a large number of computation states, and unforeseen dynamic interactions whose influence on the global program behavior is difficult to understand. To address this dimension, the distributed debugger must be

able to observe computations both at a global level, to understand the interactions, and at the level of the individual processes. This has implications concerning the debugging functionalities and its software architecture.

The second dimension (*nondeterminism*) makes the actual execution behavior dependent on actual process speeds (due to distinct processor speeds and to distinct operating system scheduling effects) and unpredictable communication delays. On one hand, this characteristic is related to the expected benefit of speeding up the computations through parallel processing. On the other hand, it may originate erroneous situations, when two concurrent actions involving distinct distributed processes are in conflict and may occur in distinct orderings depending on the above timing effects. Examples of such *race conditions* occur for shared-memory communication models, and for message-passing models [35]. This dimension requires the distributed debugger to provide facilities to detect those situations, and in general to evaluate program correctness properties, in a way that must be valid for all possible execution orderings. It also requires techniques to allow reproducible coherent observation of such error situations, without precluding user interaction.

The third dimension (*observation of global states*) must be considered because the evaluation of erroneous situations depends on accurate observations. These can only be approximately achieved, in a distributed system, by remote observation, based on message passing, so they face the difficulty of absence of a global system state. The distributed debugger must provide strategies for the observation of consistent computation states.

The fourth dimension (*probe- or intrusion-effect*) recognizes the unavoidable fact that any observation affects the system under study, so the distributed debugger must rely on techniques that ensure the lowest possible intrusion, and still allow user interaction, even knowing that this is a highly intrusive activity.

In order to understand how the above dimensions are addressed by a distributed debugging system, some formal concepts are necessary from the distributed systems theory.

The concept of a *distributed computation* (DC) represents all possible behaviors which result from executing a distributed program in a distributed system (see Figure 5.3).

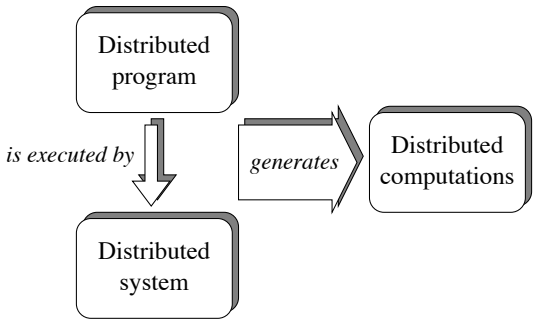


Figure 5.3: Distributed computations

In order to understand and ensure the correctness of a distributed program, one must observe and control the distributed computations which are generated when running the program. So, the debugging activity becomes an instance of the more general problem of observation and control of distributed computations because the above mentioned correctness predicates must be evaluated in meaningful computation states.

This chapter has two main goals. One is to explain the reasons why it is so difficult to develop

such an ideal distributed debugging framework addressing all the above dimensions. The other goal is to discuss the main methods and techniques that are actually being used by existing distributed debuggers, and their expected evolution.

5.1.3 Organization of the chapter

In the remaining of this chapter we discuss solutions to face the above difficulties in the context of the distributed debugging activity. The concept of distributed computation is presented in Section 5.2 and the observation problem is discussed in Section 5.3. The detection of global predicates is discussed in Section 5.4. Then, an overview of the main distributed debugging methodologies and techniques is presented respectively in Section 5.5 and Section 5.6. Finally, the main issues involved in the implementation of distributed debuggers are summarized in Section 5.7 and some conclusions are drawn in Section 5.8.

Significant references may be found in [1–3, 5, 43, 47].

5.2 Distributed Computations

The results of this section may apply to general distributed programming models based on processes and/or threads, and using communication models based on message-passing or shared-memory, with forms of synchronous and asynchronous interactions.

A distributed program is usually based on an abstract model of concurrency and communication. Its operational semantics can be defined in terms of events that correspond to process control and communication actions. Such high level entities (e.g., processes) and events are mapped into low level primitive events that are recognized and generated by the underlying distributed system. So, in this perspective a distributed system is defined as a collection of processes that communicate using a basic message-passing model with the classical *send* and *receive* primitives. A distributed system has the characteristics of an asynchronous system, so that one cannot reason in terms of an accurate global physical time reference in order to follow the chain of computation states that may lead to the cause of a bug in the distributed program.

Due to the arbitrary process speeds and message transmission delays that occur in a distributed system, distinct execution paths can be generated when repeatedly running a distributed program with a given set of input conditions, possibly leading to different results. Such nondeterminism makes it very difficult to evaluate correctness properties that should hold for all possible executions of a distributed program, and not only for a single observed execution.

The concept of distributed computation describes all possible execution runs of a program by a distributed system. See [5] for a detailed presentation of the fundamental issues concerning distributed computations and the observation of global states. It is defined in terms of two concepts. One is the concept of local history (LH) of each sequential process that is involved in the execution of a distributed program. The other concept is the cause-and-effect relationship due to local process ordering and to the event dependences originated by process interactions [29].

A *process* P_i is defined as a sequence of *events*, also called its local history LH_i. There are two types of events. *Internal* events represent local state transitions made by P_i alone, not involving any other processes in the distributed system. *Interaction* events represent process communications corresponding to message send and receive actions. The totally ordered events in P_i's local history represent the evolution of the values of all the P_i's variables and of the interactions involving P_i in a distributed program execution.

$$LH_i = e_i(0), e_i(1), \ldots, e_i(f)$$

In this sequence, $e_i(0)$ is the initialization event of P_i. It defines the process initial state, denoted by $LS_i(0)$. In general, the kth event in the process history, denoted by $e_i(k)$, produces the local state $LS_i(k)$, as the state immediately right after $e_i(k)$ occurrence. One can assume $e_i(f)$ is the termination event of P_i, and $LS_i(f)$ is P_i's final state. Discussion of perpetual processes is beyond the scope of this chapter, but similar conclusions may be drawn.

A prefix of LH_i, for example up to and including the kth event, is denoted by $LH_i(k)$ and it represents the partial history of P_i, up to a certain point in P_i's computation.

A *global history* (GH) is the set defined by the union of all local histories:

$$GH = LH_1 \cup LH_2 \cup \ldots \cup LH_N$$

A fixed number (N) of processes is usually assumed without loss of generality. Among all the event orderings represented by a global history, only some of them can possibly occur that are compatible with the causal precedence relationship ($\rightarrow$) as defined by Lamport [29]. One has relation $e \rightarrow e'$ iff e causally precedes e'. One has relation $e \parallel e'$ iff neither $e \rightarrow e'$ nor $e' \rightarrow e$. Although such relationship is only a potential causality dependence, it is generally used as the basis of distributed debugging to track causes of the errors.

A distributed computation is a partially ordered set (poset) defined by $(GH, \rightarrow)$. Intuitively, this reflects all physically feasible event combinations that must be obeyed by all possible executions of a distributed program in a distributed system. A distributed computation may be represented by a process-time diagram where the event causality chains replace the classical notion of instant physical time in a centralized system with a global clock (see Figure 5.4).

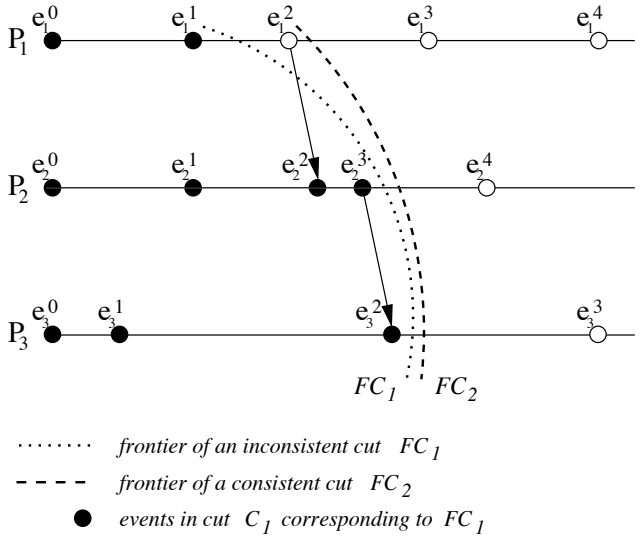


Figure 5.4: Process-time diagram

The observation problem in distributed debugging requires one to think about the global states of a distributed computation. A *global state* (GS) is a n-tuple of LS:

$$GS = (LS_1, LS_2, \ldots, LS_N)$$

where LS_i ($i \in \{1, \ldots, N\}$) is the local state of P_i corresponding to some prefix of P_i's local history. The initial global state (denoted by $GS(0)$) of a distributed computation is defined by the initial local

states of all processes i.e., $LS_i(0) : \forall i \in \{1,\ldots,N\}$. The final global state of a distributed computation (denoted by $GS(f)$) is defined by the final local states of all processes i.e., $LS_i(f) : \forall i \in \{1,\ldots,N\}$. The difficulty with the "intermediate" global states is that all combinations of local state tuples cannot occur in real executions of a distributed program ...

In relation to a process-time diagram, like the one shown in Figure 5.4, the concept of a *cut* (C) is defined as a subset of the global history. It represents a partial global history of a distributed computation because it is made of prefixes of all processes' local histories. The *frontier* (FC) of a cut C is the n-tuple of the last events in each prefix of LH_i for all $i \in \{1,\ldots,N\}$. The frontier FC seems to represent a view of the global progress of a distributed computation up to a certain point in the execution in terms of the last occurred events (see Figure 5.4). There is a well-defined unique global state corresponding to each frontier FC, that gives the last occurred local states for each process.

However, only *consistent cuts* (CC) are significant for the purpose of distributed debugging. A consistent cut CC is "left closed" under the $\rightarrow$ relationship, i.e.,

$$\forall e, e' \in GH : e \in CC \wedge e' \rightarrow e \Rightarrow e' \in CC$$

Intuitively, a consistent cut incorporates all the past of its own events. A cut that would include some event e and not all events causally preceding e, cannot correspond to a possible view of a distributed program execution.

A *consistent global state* (CGS) is the global state defined by the frontier FC of a consistent cut. A consistent global state represents a global state that can possibly occur during a distributed program execution because it represents a view of the global state that respects the causal precedence among events. In Figure 5.4 the global state corresponding to FC_2 is a consistent state, unlinke the state corresponding to FC_1.

The consistent cut and consistent global state concepts can be used as a basis to define observation models for distributed debugging purposes. An intuitive notion of the *current state* of a distributed computation can be visually caught by considering the events (and states) to the "left" of the frontier of a consistent cut, as equivalent to a past history, and the events to the "right", as the ones in the future. This suggests one could consider an incremental progression of the distributed computation, followed by the user under the control of a distributed debugger where "successive" consistent global states would be examined for evaluation of correctness predicates. Indeed this is an important research direction in distributed debugging, but it has several inherent difficulties that will be discussed in the following.

In order to understand the behavior of a distributed program one has to consider all intermediate consistent global states that can possibly occur starting by the initial state $GS(0)$ until the final state $GS(f)$. For each execution of a distributed program, a distinct set of consistent global states may be followed so each execution generates a distinct sequences of states, due to the nondeterminism of a distributed system. However, to ensure correctness, one needs to reason in terms of all such possible sequences of consistent global states.

The concept of *consistent run* (CR) represents a possible observation of a distributed computation where all the events appear in a total ordering that extends (i.e,. is compatible to) the partial ordering defined by Lamport's causal precedence relation. A consistent run can be obtained from the process-time diagram by building a sequence of events that respects the causality chains and additionally imposes an arbitrary ordering among the concurrent events. A consistent run defines a sequence of consistent global states such that

$$CR = GS_1, GS_2, \ldots, GS_k, GS_{k+1}, \ldots, GS_m$$

where $GS_1 = GS(0)$, $GS_m = GS(f)$, and GS_{k+1} only differs from GS_k in one local state in one of the processes. Intuitively, a consistent run defines a sequence of consistent global state where each new global state in the sequence is obtained by making a "single step" in a single process. One says GS_k *leads-to* GS_{k+1} in a consistent run *CR* if they are immediate neighbors in that consistent run. A *GS* is reachable from a GS' in a consistent run iff $GS' \mapsto GS$ in that *CR*, where $\mapsto$ is the transitive closure of the *leads-to* relation.

Although the consistent run concept artificially restricts the distributed computation to make progress one event at a time, its arbitrary event ordering represents the nondeterminism of a distributed program execution. In order to generate all possible sequences of consistent global states in a distributed computation, one has to consider the set of all possible consistent runs. The set of all consistent global states, ordered by the leads-to relation defines the *lattice* $\mathcal{L}$ of global states that characterizes all possible execution paths in a given distributed computation. $GS(0)$ and $GS(f)$ are respectively the *infimum* and *supremum* elements of $\mathcal{L}$, and the set of all consistent runs is the set of all paths from $GS(0)$ to $GS(f)$.

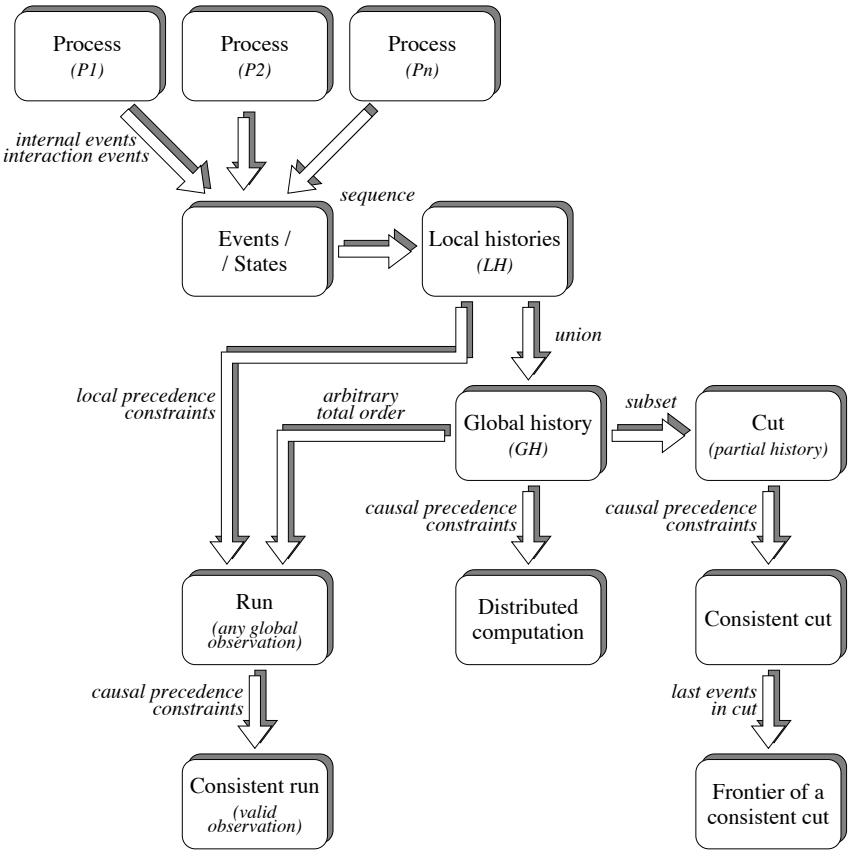


Figure 5.5: A summary of distributed computation concepts

In summary, the operational semantics of a distributed program can be described in terms of the distributed computations that are generated when the program is run by a distributed system (see Figure 5.5 for a summary and [5] for a deeper presentation). The distributed computation describes

all possible sequences of consistent global states that can occur, starting with the initial $GS(0)$ and leading to the final $GS(f)$ state. Such set of possible sequences is only constrained by the local process event ordering and by the Lamport causal precedence relation.

An exhaustive traversal of such paths would be necessary to verify or detect correctness properties of a distributed program. This approach is in general infeasible due to the large combinatory of consistent global states that would have to be examined. Moreover, the problem of constructing individual consistent global states poses itself additional difficulties.

5.3 Observation of Global States

The intuitive notion of global state of a distributed computation consists of a collection of local states that could be viewed by some ideal external observer. Note that the notion of global states, as presented above, implicitly includes the state of interprocess communication channels. In a distributed system, the only possible way for an external observer to build such a view is through message exchange with each remote individual process. There are several aspects related to the observation problem:

- The global state can be *obsolete* at the time the global view is actually constructed by the external observer. This occurs in case the observation is performed online, during actual distributed program execution. This aspect is particularly relevant in distributed reactive applications [5, 9]. In distributed debugging using online observation of a distributed computation, if the goal is to track a particular global state in order to detect bugs, mechanisms will be required to halt the execution in all processes and restore the n-tuples of local states that one wants to examine in detail. Techniques to tackle this problem will be reviewed in a section ahead. If the observation is performed off-line, in a postmortem analysis of the distributed computation global histories, this problem does not arise.

- The observed global state is a cut of the distributed computation. In the online observation approach, the constructed global state corresponds to a cut of the distributed computation, so it only allows one to reason about what happened so far. In the off-line observation mode, both the past and the future are known and may be accessed by the observer.

- The observed global state must be a consistent cut of the distributed computation. Observation of inconsistent cuts may occur due to the unpredictable message delivery orderings in a distributed system. An inconsistent sequence of events may be built by the observer that does not preserve the causal precedence relationship. Approaches to build consistent cuts are discussed in this section.

- Multiple independent observers may build distinct views of the same distributed computation. Even if consistent cuts are ensured, several independent observers may build distinct consistent cuts. The presentation of uniform views of a distributed computation to multiple concurrent and independent observers requires an adequate coordination between them, and has been discussed in the scope of distributed system research [5]. This is an issue that has not been considered in most of existing distributed debugging tools. However, its relevance is increasing with the emergence of integrated development environments where several concurrent tools act as observers (and sometimes controllers) of an ongoing distributed computation. Tool coordination is briefly mentioned in Section 5.7 and illustrated in several chapters in Part II of this book.

The practical methods for observation of distributed computation depend on the distributed debugging approach:

Off-line In this approach, it is possible to analyze global histories of a distributed computation that were generated by a previous distributed program execution or by a simulation of the distributed program model. These methods always deal with complete histories.

Online In this approach, it is necessary to develop algorithms to construct a global state or a consistent run of a distributed computation during an actual distributed program execution. These methods deal with partial histories.

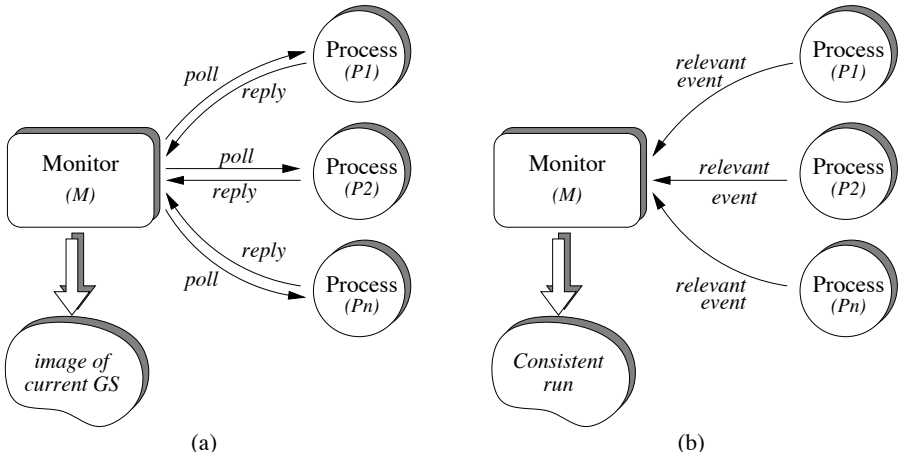


Figure 5.6: Two observation approaches: (a) Global snapshot (b) Active notification

The main approaches to construct online observations of a distributed computation rely on an external observer or monitor process (called $\mathcal{M}$ hereafter). In distributed debugging, they can be used to build snapshots of global states of the distributed computation in order to evaluate correctness properties at those points in a computation. They can also be used to build observations of consistent runs that could have been followed during execution of a distributed program. In all observation strategies it is necessary to consider the problem of the probe effect. Here an assumption is made that the probe effect has a negligible effect in the sense that the message exchanges between $\mathcal{M}$ and each process P_i, do not affect the event ordering of the distributed computation when compared to a unmonitored computation. A detailed discussion of the probe effect is presented in Chapter 6.

All existing approaches for online observation make specific assumptions on the message delivery rules that should be enforced by the distributed system, ranging from FIFO ordering between pairs of processes to causal delivery of messages. Such assumptions are important so that the external observer may be able to construct consistent global states. A discussion of such delivery rules, e.g., based on vector timestamps, is beyond the scope of this chapter. Here one should note the relevance of vector timestamps to support event relationships that exactly capture the causality. The reader may find complete surveys in [13, 34, 46].

The first strategy tries to build a *global snapshot* [7] that corresponds to a consistent global state of the distributed computation (see Figure 5.6 (a)). The observer $\mathcal{M}$ is responsible for polling all the processes and these must reply by sending their corresponding local states. The observer then

constructs a consistent global state, so that the method ensures there is a path in $\mathcal{L}$ that contains the constructed consistent global state. Successive polling requests by $\mathcal{M}$ can be used to try to build an image of the evolution of the consistent global states followed by the distributed computation. Unfortunately, this method may miss important consistent global states, and it exhibits either too much overhead due to the polling or a great delay in the observation.

The second strategy allows $\mathcal{M}$ to build an entire observation or consistent run of the distributed computation (see Figure 5.6 (b)). Each distributed program process is responsible for notifying $\mathcal{M}$ when each relevant[1] event occurs. $\mathcal{M}$ collects all information from all the processes and ensures it builds a consistent run. Such a consistent run corresponds to one of the possible paths that the distributed computation could actually have followed.

Each observation method has its limitations and uses as discussed in the following.

5.4 Detection of Global Predicates

A general method underlies the work by several authors [9, 19, 25, 51] to support the distributed debugging activity, according to the three following steps: specification of global correctness conditions, their (off-line or online) evaluation, and corresponding reaction of the distributed debugger, depending on the result of the evaluation [4]. A complete approach should allow:

1. Specification of behavior. There are several approaches to specify global predicates in terms of local and global states.

2. Detection of global predicates. There are several proposals for off-line and online global predicate detectors for distinct kinds of global predicates [6, 17].

3. Reaction of the distributed debugger. The distributed debugger must perform the (off-line or online) reconstruction of the consistent global state that satisfies the detected global predicate.

These issues underly the theoretical framework to implement several distributed debugging techniques for observation and control (see Section 5.6).

5.4.1 Global Predicate Specification

This step starts by the identification of desired or undesired distributed program properties corresponding to the program correctness criteria.

These properties are then expressed as global predicates (GP) which are boolean expressions involving conditions on the local variables of multiple processes or on the states of communication channels. Multiple authors have proposed distinct specification languages for global predicates in terms of global states or events, as well as in terms of sequences of states or patterns of events [6, 17].

The main issues in the design of such specification languages are:

(i) Expressiveness of the language, to be adequate to specify the desired conditions corresponding to correctness or erroneous situations.

(ii) Computational complexity, to be amenable to an efficient implementation. A highly expressive notation is of reduced practical interest if it implies a NP-complete evaluation.

[1] The specification of "relevant events" depends on the kind of state changes one wants to observe.

(iii) An adequate compromise between the abstraction level of the distributed program model and the observation level of the distributed debugger. This means allowing an easy mapping between the correctness conditions, the distributed program concepts (and the program code itself), and the actually observed entities by the distributed debugger, e.g., the low level events or global states.

The main obstacle to the full adoption of this predicate based approach in distributed debugging has been the complexity of the evaluation of the global conditions in a distributed system.

5.4.2 Evaluation of global predicates

This step is responsible for the detection of global predicates using off-line or online approaches. The problem of evaluating general forms of global predicates has been studied and found NP-hard [8], so several authors have focused on the evaluation of restricted forms of global predicates, such as conjunctive [18, 24] and disjunctive global predicates. Although restricted, such global predicates are still useful in distributed debugging. An important distinction is established among so-called stable and unstable properties of a distributed program. Once a stable property becomes true in a specific global state *GS*, it remains true in all the following ones that can be reached from *GS*. Deadlock and termination are examples of stable properties. On the other hand, an unstable property may dynamically change its truth value during the distributed execution. Both types of properties are important for analyzing the correctness of a distributed program. Unstable properties reflect many situations resulting from the dynamic behavior of a distributed program, e.g., distributed mutual exclusion.

The evaluation of stable and unstable properties poses different requirements. The former properties can be caught by online observations based upon the global snapshot approach, provided the polling requests are repeated until a consistent global state is found where the required stable property holds. Unfortunately, the detection of unstable properties is more difficult. It cannot be ensured by online observations based on the global snapshot approach, as the constructed global state may miss the point of the distributed computation where that property temporarily holds, due to the uncertainties arising in a distributed system. Concerning the online construction of consistent runs, based on the active notification approach (see Figure 5.6), even if the property holds for a certain consistent global state in that constructed run, this does not gives information about how the property behaves in other possible consistent runs.

Extended forms of global properties have been proposed by several authors that try to express the distributed program behavior in terms of the entire distributed computation, instead of a single consistent global state. These have the forms:

$Definitely(GP)$: for all consistent runs of a distributed computation, there exists a consistent global state that satisfies *GP*.

$Possibly(GP)$: there is a consistent run of a distributed computation such that it contains a consistent global state satisfying *GP*.

The second form is particularly useful for distributed debugging, for a property expressing an undesirable or erroneous condition, it would be sufficient to find a consistent global state where the corresponding GP holds. This may also require the search along the lattice of consistent global states but only until such a consistent global state is found.

For a correctness property expressing a condition that must hold in all possible executions of a distributed program, the corresponding GP must be true in all the paths in the lattice of consistent global states defined by a distributed computation.

Several authors have exploited approaches for building and traversal the entire lattice of consistent global states [9], which are adequate for evaluation of both stable and unstable predicates, of the possibly and definitely kinds. Other authors have tried to exploit specific and simplified forms of GP, e.g., conjunctive or disjunctive, in order to avoid an exhaustive search of $\mathcal{L}$. A summary of these approaches can be found in [17].

A distinct approach, albeit with the same objectives of evaluating correctness properties, concerns specific forms of race conditions and their detection. Such mechanisms are usually embedded as implicit functionalities of a distributed debugger [23, 41].

5.4.3 Reaction on detection of a global predicate

Depending on the user interpretation of the logical condition that was evaluated, a particular reaction may be necessary. For example, if the detected global predicate corresponds to a bug situation, a distributed debugger should be able to execute the following actions:

(i) Stop the distributed program execution.

(ii) Restore the local states of all processes in a meaningful consistent global state that satisfies the detected global predicate.

(iii) Allow the user to examine individual local states of that consistent global state as well as of its past.

The first action is obviously only required when the predicate detection is made online. Its implementation is open to multiple interpretations because of the asynchronous evolution of independent computation processes. The above actions are simpler for a post-mortem approach that accesses the complete computation histories. Such approach may even allow the user to traverse the execution paths into the past and the future of the reconstructed global state. In Section 5.6 a review of existing implementation techniques is presented.

5.5 Debugging Methodologies

In this section three main criteria are used to classify distributed debugging methodologies: (i) What steps of distributed debugging activities are supported in the development cycle? (ii) At what time in the application development cycle do such distributed debugging activities take place? (iii) What is the observation model used?

5.5.1 The steps in the cycle of distributed debugging activities

Distributed debugging methodologies can be classified according to the level of support they provide to the user concerning the activities of global predicate specification and detection, and the search for the causes of the distributed program bugs (see Figure 5.7).

In the following, these approaches are successively discussed, starting from the simpler approaches to the more complex ones. These approaches are complementary to one another, in the sense that each approach tries to overcome a limitation of the previous approach in the sequence.

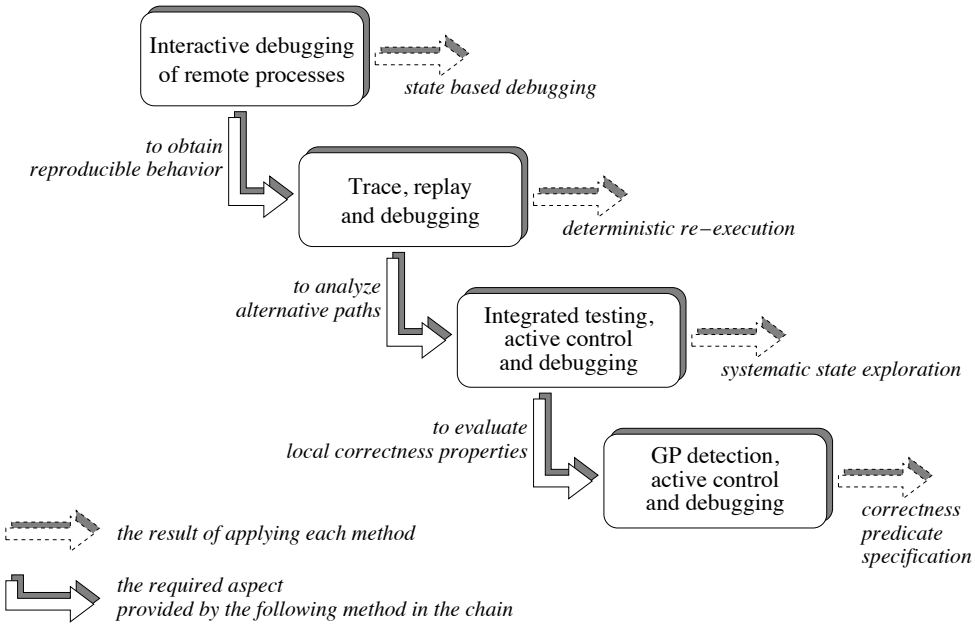


Figure 5.7: Distributed debugging methodologies

Interactive debugging of remote sequential processes

This is based on an extension of conventional sequential debugging commands to allow individual online observation and control of the execution of remote sequential processes. This is a limited approach that only allows to examine local histories of individual processes of a distributed program. As each local history describes only the evolution of each process in terms of its internal and interaction events, it is the programmer's task to build the global picture of the corresponding distributed computation. However, as such basic remote debugging mechanisms are required to enable more sophisticated approaches, they are supported by almost all existing commercial or academic distributed debuggers. The main distinction between existing distributed debuggers of this kind is related to the functionalities and design of their architectures.

In particular, this approach does not handle the nondeterminism behavior that is exhibited by a distributed program on a distributed system.

Trace, replay and debugging

In order to address the non-reproducibility issue, this approach is based on collecting a trace of the relevant events generated by a distributed computation, during a first run of the distributed program. The trace describes a computation path (a consistent run) that can be analyzed at a post-mortem stage. If erroneous situations are found, the distributed program can be re-executed under the control of a supervisory mechanism. This mechanism uses the traced sequence of events to force the distributed computation to follow the same path as the one executed by the previous run. This allows the user to examine the behavior of that path within a cyclic interactive debugging session, in a reproducible way. In such a session, the user may use the observation and control functionalities provided by the previous approach. The trace and replay technique has been the focus of intensive

research in the past decade, mostly concerning the reduction of the probe effect and of the volume of the traced information (see Section 5.6). However, not all commercial debuggers include such a facility. This approach is also used in monitoring systems, as discussed in Chapter 6.

From the view point of distributed debugging, there is a limitation in this approach if it gives no support to analyze other distributed computation paths besides the traced one. If the first run which is used to collect the trace is a "free" run i.e., under the control of no supervisory mechanism, the resulting trace describes only a randomly occurring path from the large set of possible paths. This gives no guarantee that such is an (the) interesting path to consider for analysis. Indeed, it is highly unlikely this will be the case.

Integrated testing, active control and debugging

This approach tries to overcome the above mentioned limitation of a simple passive trace and replay approach. Multiple authors have proposed approaches for the active control of distributed program execution for distributed debugging purposes. They all share a similar goal, namely to provide a facility to enforce the execution of specific runs of a distributed computation in order to ease the location of erroneous situations. They differ in the way they generate and specify the desired consistent run that a controlled execution should follow. In the following, one of these approaches is briefly described for illustrative purposes.

The approach considers two separate phases in the distributed debugging activity. It is based on the integration of a static analysis and testing phase (hereafter called the T phase) and a dynamic analysis and debugging stage (called the D phase). The goal of the T phase is to assist the user in the generation of interesting consistent runs that may exhibit violations of correctness properties. In general it is not feasible (or even possible) to provide a completely automated T phase. An interactive testing tool is useful to cooperate with the user to specify and refine the conditions and regions of distributed program code that should be considered for analysis. The T phase is then used to generate a sequence of commands that will be used to drive a distributed program run, in order to exercise the paths defined by the above testing scenarios. Such a distributed program run can then be the subject of a trace and replay approach, and integrated in a cyclic debugging session.

The main advantage of this methodology is that it allows the user to interactively "walk" through the T and D phases, until one is convinced about the satisfaction of the correctness properties that are being investigated. Another advantage of this approach is that it combines the benefits of static and dynamic analysis in order to help the user to understand distributed program behavior.

The main problem with this approach is that it basicly relies upon the user conviction that all relevant scenarios were specified and generated, tested and analyzed, so that one gets confidence on distributed program correctness. There is no full guarantee that no important situations went unnoticed (this is a classical characteristic of testing approaches: *How do you specify a complete test suite?* See Chapter 9).

Automated detection of global predicates, active control and debugging

This approach is an attempt to help the user increasing the confidence on the results of the previous approach, by allowing the specification of the correctness criteria in terms of global predicates. Such global predicates are then automatically evaluated by detection algorithms, working off-line or online. A summary of the main goals of this approach was given in a previous section.

As the efficient evaluation of global predicates is limited to restricted classes of predicates, this approach may be seen as complementary to the testing and debugging approach. Their integration seems a promising research direction to improve distributed debugging.

5.5.2 The times of the distributed debugging activities

Distributed debugging approaches can also be classified according to the phases of the development cycle (see Figure 5.8).

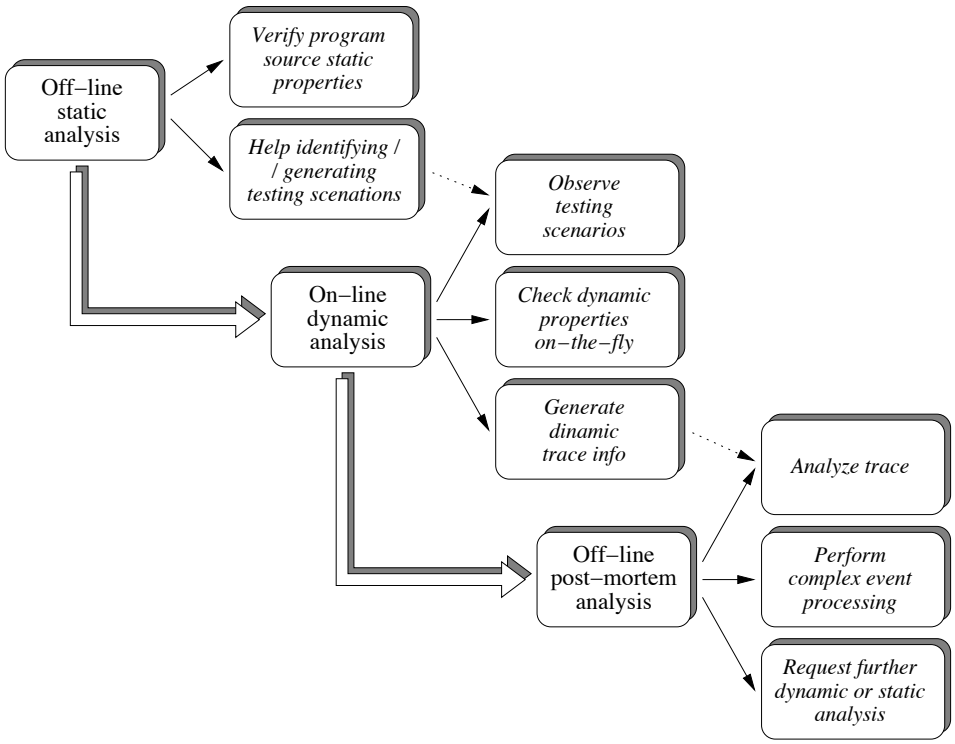


Figure 5.8: Program analysis and debugging at distinct phases of development

Off-line distributed debugging based on static analysis

This approach uses the program code as a basis and it does not require an actual program execution. It relies on formal models of program behavior that can be used to check certain kinds of properties, usually expressed as temporal logic formulas. However, model checking techniques can only be used to analyze certain properties and do not give information on dynamic properties that depend on actual runtime program behavior, e.g., termination. Also, they usually incur great computational costs in their search for all allowable state transitions in the modeled computation space.

Still, static analysis of the program code is one approach that can be of great importance for distributed debugging, when adequately combined with complementary approaches.

Online distributed debugging based on dynamic analysis

Due to the mentioned limitation of static analysis, one needs to use online approaches that help evaluating the actual program behavior on-the-fly. Such approaches rely upon online observation techniques so they must deal with the difficulties of accurate construction of consistent global states.

Once a specific program behavior pattern was detected, these approaches also require adequate control mechanisms to help the user inspecting the individual computation states of interest. This approach must deal with the probe effect, in order to ensure that the observed computation path exhibits the same logical behavior as the original computation would, when running under no observation mechanisms ... (see Section 5.6 and Chapter 6).

Dynamic and static analysis approaches can be combined in order to provide the distributed debugging with functionalities as the ones illustrated by the mentioned integrated testing, active control and debugging approach.

Off-line distributed debugging based on post-mortem analysis

Post-mortem analysis approaches provide an effective way to analyze program behavior because they rely upon previously collected traces of the processes' local histories. On one hand, it becomes easier to construct a consistent global state, out of these local histories, by regenerating the causal precedence chains. This reduces the runtime overhead incurred by online approaches. It also enables facilities for analysis of complete computation histories, with the help of a diversity of event analysis and visualization tools. On the other hand, post-mortem techniques can be integrated with online techniques, in order to exploit tracing, replay and debugging methods, to address the non-reproducibility issue. Incremental methods consisting of online and postmortem stages also allow to handle the potentially large volume of traced information. A first run is used to collect only the minimum amount of information to ensure reproducible re-execution, and further post-mortem analysis can determine the need to collect further information on successive runs (see also Chapter 6, on the use of this approach for performance debugging purposes).

The summary of how such approaches are complementary to one another is illustrated in Figure 5.8.

Off-line: verify certain properties using static analysis and help identifying relevant scenarios for testing.

Online: check dynamic properties on-the-fly, and observe testing scenarios, under actively controlled execution.

Post-mortem: analyze traces of complete global histories, perform more complex event processing (e.g., high level event abstractions) and visualization. Use the results of such analysis to determine further runs and dynamic analysis.

5.5.3 The observation models

Finally, distributed debugging approaches can be classified according to the observation model of distributed computations (see Figure 5.9).

State-based views in distributed debugging

This approach aims to provide the equivalent functionalities to state-based sequential debugging but must address consistency issues. It considers state exploration at two distinct levels.

The "component level" considers processes and threads as individual computation units, whose sequential state transitions must be examined.

The "distributed program level" considers component interactions at a global level. Typically, in a distributed-memory model, global process interactions are of the message-passing or RPC types.

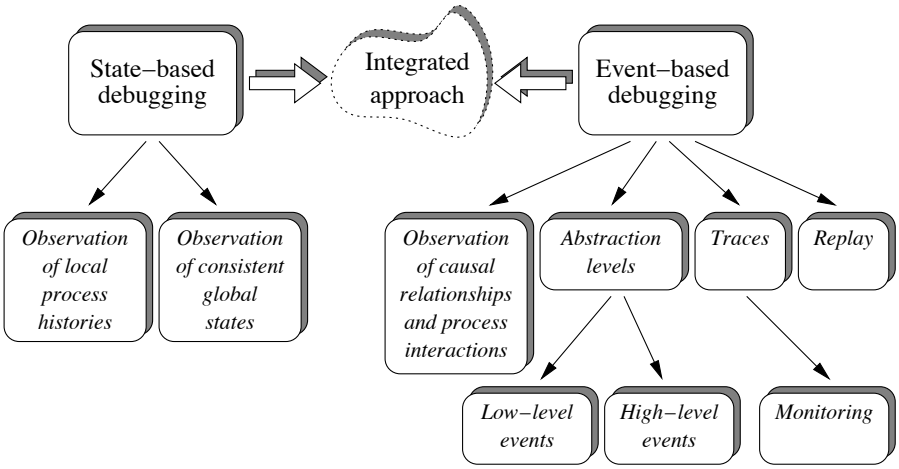


Figure 5.9: Debugging approaches according to the observation model

In a computational model including distributed processes and multi-threading within each process, shared-memory interactions are also considered among the threads in each process.

Formally, a state based view interprets process communication as a way of having one process affecting the state of the other. For example, some authors interpret a message receive operation as equivalent to an assignment of the value of a variable of the sender process (the send buffer, indeed) to a variable in the receiver buffer. This allows to directly express dependencies and correctness conditions in terms of state variables.

State-based approaches for distributed debugging try to explicitly handle the observation problem by describing distributed program evolution in terms of global states. While this view is necessary as an important way of identifying errors, it makes it difficult to relate logical correctness conditions, distributed program code locations, and actually observed computation states.

An explicit event based approach seems more adequate, both to describe the dynamic distributed program evolution, and to provide a transparent interpretation of correctness properties.

However, the state based view has also an important role in distributed debugging to allow to examine local states and global states.

Event-based views in distributed debugging

Event-based approaches for distributed debugging are important for a number of reasons.

Events are the natural concepts to track causality relationships as they underly the theoretical concepts proposed by Lamport's precedence relations. They are also at the root of a large diversity of work, both on the theory of distributed computations, and on the support of causality mechanisms like the vector timestamps.

Event-based models enable distinct levels of interpretation of a distributed computation, at distinct levels of abstraction. They ease the mapping from the high level abstractions of a distributed programming model into the low level abstractions of a distributed system. So, it becomes possible to use an event-based model to specify the desired program behavior, in terms of user-level abstractions, and check if it matches the observed program behavior. Concerning practical interactive distributed debugging tools, event abstractions allow to provide a high-level view to the user, in

terms of the application level model, e.g., graphical notation or a high level semantics.

Events are the adequate concepts to handle the reproducibility issue through deterministic replay techniques as discussed in a section ahead.

Events allow to bridge the gap from theoretical concepts to the practical tools that support the distributed debugging activity. Namely, event-based models ease the combination of trace-based monitoring, distributed debugging, and visualization techniques.

In summary, event-based models provide the adequate concepts to capture the semantics of distributed programs, and they ease the development of practical distributed debugging tools:

- They relate to the theory: causal precedence and "poset" of a distributed computation.
- They specify behavior at distinct abstraction levels.
- They enable reproducibility using traces.
- They allow runtime detection of relevant conditions, related to breakpoints.
- They relate to analysis and visualization tools so they bridge the gap from theory to practice.
- They allow unification with monitoring tracing approaches.

5.5.4 Integrating event and state based distributed debugging

An event-based model can be used to specify program behavior and to detect the occurrence of a particular program behavior. This will then require the examination of the corresponding global computation states. So it becomes natural to associate event with state based distributed debugging techniques. Inspection of local histories of individual process can be based on a state-based debugging technique, as provided by typical sequential debuggers. Global state views should be triggered by the detection of significant events, so that state examination may enlight the reasons for the errors.

The main difficulty of this integration is due to the need of (re-)constructing a consistent global state of a distributed computation, whenever some particular event is detected. Only then will the user be able to inspect meaningful individual process histories, using a state-based debugging technique. This problem is discussed in the following section.

5.6 Debugging Techniques

The main techniques supporting distributed debugging can be classified according to the main goal: observation or control of the distributed computations. In the following sections, the main characteristics of existing techniques are surveyed.

5.6.1 Observation

Here, a wide interpretation of the observation concept is assumed. It encompasses the low level observation of actual running computations, up to the consistent observation of global computation states, and including the logical observation of distributed program behavior that is the realm of high level visualization tools. In distributed debugging, all those observation dimensions must be considered so that logical program properties can be studied.

The general goals of observation techniques are:

(i) To collect the required information [33] (see Chapter 6).

(ii) To evaluate (re-)construct consistent global states in order to evaluate correctness conditions.

(iii) To interpret, analyze, visualize and animate distributed program behavior, at multiple levels of abstraction (see Chapters 2 and 8).

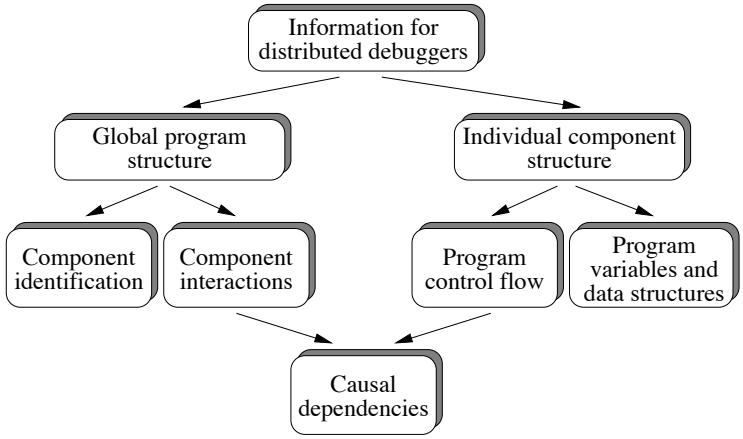


Figure 5.10: Information for distributed debugging

In general a distributed debugger should provide information of distinct types, as illustrated in Figure 5.10. The component and global program level views were already explained. The Figure 5.10 puts the focus of the distributed debugging activity on the observation of significant information in order to understand the causal dependencies that explain program behavior.

This information can be examined at several phases during program development. Minimal information on process interactions and program control flow is usually obtained using a trace-based approach while the remaining information on the global computation states and individual process states is usually obtained dynamically, on user demand, using an interactive debugging cycle. This is not the case when debugging distributed real-time applications where interactive debugging is not used due to the real-time constraints. Real-time distributed debugging is not discussed here (see [10, 53]).

Due to its practical importance, trace based techniques are briefly discussed in the following, from the perspective of distributed debugging.

Using tracing techniques for correctness distributed debugging

Event based techniques rely on monitoring to collect the relevant information and generate the traces describing the distributed computation histories (see Chapter 6).

Tracing provides the following main functionalities for distributed debugging (see Figure 5.11):

(i) To enable deterministic re-execution [30, 54].

(ii) To gather sufficient information to allow further event analysis by other tools, aiming at the interpretation, visualization, and animation of distributed program behavior, at adequate levels of abstraction.

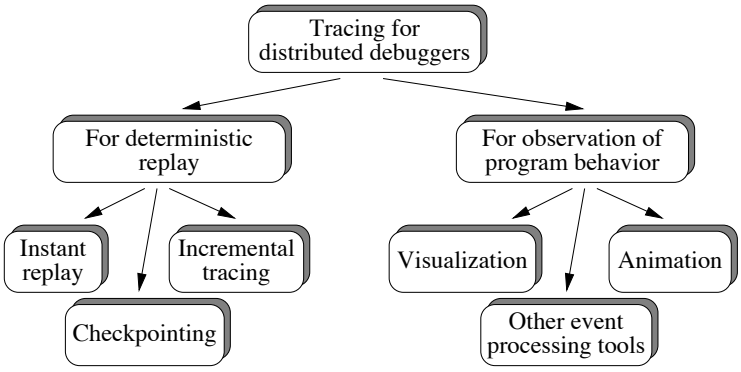


Figure 5.11: Tracing for distributed debugging

A critical aspect is how to determine the required information that should be traced.

Concerning deterministic re-execution, a conservative approach (called the event log approach) saves the contents of all the process state information, including values of variables and contents of exchanged messages. It also includes information related to the possible sources of internal non-determinism in a process, such as consulting real clock values, communication with external I/O devices, and internal nondeterministic choices. This allows deterministic re-execution in the more general case and it has the advantage of allowing to repeat single (or groups of) processes in isolation, i.e., just by providing the traced input information from their external environment. However, such an approach often generates a large volume of traced information. The main risks of such approach are the increased requirements in memory and disk space, and the increased perturbation due to the monitoring activity.

So, in applications where each individual process exhibits a deterministic behavior, it is sufficient to register the ordering of the relevant distributed computation events, namely the interaction events, instead of all their associated state information. This is called the *Instant Replay* approach [30] that is a landmark in the distributed debugging area. In its simplest form, this requires all processes to be re-run, during a replay session, but allows one to examine further detailed state information on each process and event of the observed distributed computation path, using interactive debugging commands.

For long running applications, a checkpointing technique may be needed to allow the replay to start from intermediate points in the computation, instead of from the beginning. Such a facility requires a mechanism to perform consistent global checkpoints (corresponding to consistent global states).

Concerning the use of tracing for high level observation and analysis of program behavior, it seems one would need to collect much more information in order to enable the post-processing tasks related to program behavior analysis and visualization. However, if there is an adequate event modeling abstraction, the user may be able to specify the desired level of abstraction that is required at each point during application development. This allows to filter, cluster or ignore certain kinds of events or/and processes, thus contributing to reduce the trace volume. If further analysis reveals the need for further details, it is possible to (deterministically) re-run the distributed program with the trace facility enabled to gather such additional information. This incremental tracing facility has been used in a diversity of distributed debuggers [38]. A good compromise between the event logging and the instant replay approaches can be reached by following incremental tracing techniques,

driven by user demands, during an interactive debugging cycle based on deterministic replay.

For off-line or post-mortem processing tools, clearly all needed information had to be collected, in a single run or in a series of incrementally traced runs. For online processing, e.g., visualization, such tools may actually consume the generated events during execution, so there is less need to save large event traces.

Even for the instant replay technique, one cannot avoid the probe effect so there is the risk of a logical distinction between the instrumented and non instrumented program behaviors. For software monitoring approaches, some authors propose to keep the instrumentation probes integrated into the application all the time, so that there is no such distinction. This is only feasible if the corresponding intrusion is minimal and compatible to the required application performance (see Chapter 6).

In the past decade there was intense research concerning the reduction of the tracing time and memory space overheads [12, 39, 40, 45]. An example of an optimization is to avoid tracing unnecessary event orderings corresponding to program regions that are known to be deterministic.

State exploration in distributed debugging

One of the main requirements of a distributed debugger is a facility to allow the exploration of computation states of a distributed program (Figure 5.12).

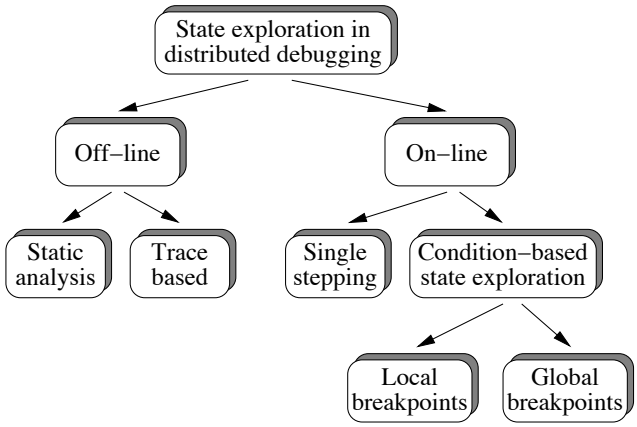


Figure 5.12: State exploration in distributed debugging

In classical sequential debugging an instant observation technique has been used since the early days of the Von Neumann machine: the memory dump. In a sequential debugging context, the main criticism results from the low level abstraction and the amount of information that is provided by this technique. This kind of technique is, however, also the basis of state-based sequential debugging, although the information is shown at a higher level and is structured according to the program entities of interest at each point (e.g., stack frames, etc.). In a distributed debugging context, the corresponding technique of making such kind of complete global snapshots of the ongoing distributed computation is not easy to achieve as previously discussed. One can also argue that the technique is of limited use in distributed debugging, due to the existence of too many states.

A single-stepping facility, as found in sequential debugging, is not much useful in distributed debugging, although it is conceptually possible to achieve, in terms of a succession of consistent cuts (see Section 5.2), corresponding to a consistent run. The problem is that there is a large space

of alternative consistent runs that would have to be searched, step-by-step, using this method. Even if such an approach is used in association with high level abstractions, that try to focus on interaction events, the approach is still generally infeasible for realistic programs, except for a few toy programs.

Off-line visualization tools, based on an interpretation of traces, can contribute to identify interesting patterns of program behavior. Interactive, user-driven visualization tools allow the user to select the abstraction levels and the areas of the program that should be considered.

Condition based state exploration is a technique that allows the user to specify logical conditions such that, once satisfied in a certain state, force the program execution to be suspended in that state. The goal is to allow the user to perform state exploration selectively, as a strategy to avoid an exhaustive, albeit systematic, search of the computation states. This is clearly an important approach that recognizes the impossibility of completely automating the distributed debugging task, so it opens a way for the user to direct the state search.

In sequential debugging those conditions usually involve the program variables, status, or instruction locations. Such breakpoints or watchpoints have a simple and well defined semantics in a sequential program but allow several possible interpretations in distributed programs. Furthermore breakpoints and interactive debugging techniques are highly intrusive and affect program behavior in an unpredictable way. So, the combination of breakpoints with deterministic replay-based techniques becomes of the utmost importance in distributed debugging.

The global conditions can be expressed in several forms. They can be expressed in terms of variables, code locations or local states of the individual processes of the distributed program. A local predicate is one which refers to a single process, so it corresponds to a local breakpoint that may be easily detected by the distributed debugger. A global predicate is a boolean expression involving conditions that are local to multiple processes. Common forms are conjunctions and disjunctions of local predicates, respectively called conjunctive and disjunctive global predicates. Other kinds of global conditions express temporal relations between abstract events, aiming at easing the task of relating the occurrence of errors to the logical abstractions in individual processes.

Most existing distributed debuggers only allow the specification of local breakpoints. This is due to the difficulty of evaluating general forms of global predicates, as previously discussed. Indeed, efficient detection algorithms exist only for very restricted forms of global predicates.

However, even for simple local breakpoints, one has to decide what to do when a local condition is detected during execution in a specific process. For an online approach, the question is how to stop the distributed program at a consistent global state that is meaningful for the examination of the detected condition. Typically, one finds two main approaches [21, 37]:

(i) To send stopping messages to all other processes.

(ii) Just wait until all other processes "naturally" block, waiting for input or for synchronization with the initial process.

In both cases there is a delay in the detection of the stop condition in other processes. As this will be an unpredictable delay, such approaches should only be used when there is a guarantee that the other processes cannot possibly affect the investigation of the detected conditions. This obviously depends on the types of conditions. For local predicates, the above approaches seem reasonable.

For global breakpoints, their associated conditions are global predicates, so in general there is a detection algorithm that forces the involved processes to stop while the remaining processes continue running as above. For example, consider a distributed computation consisting of 4 processes P_1, P_2, P_3, and P_4, and a global breakpoint involving variables in P_1 and P_2, but not in P_3 and P_4. Processes P_1 and P_2 are forced to stop on the global breakpoint, but P_3 and P_4 continue. In general, P_3 and

P_4 may affect the global system state, e.g., by sending and receiving messages, in a way that may change the conditions that explain a bug situation.

Reconstruction of a meaningful global state may require the aid of rollback and checkpointing techniques. It may become a difficult problem if a non-replay based approach is followed.

In a replay based approach, breakpoints are only set during replay. In order to handle such situations, replay-based distributed debuggers have been proposed [32] that are able to reconstruct a consistent global state such that each process is at its last local state where it could cause the involved processes (P_1 and P_2, in the example) making the global predicate true. The reconstruction of this global state relies upon the trace to identify the desired halting local states. In some proposals a new annotated trace is generated so that one may replay the distributed program to the last consistent global state defined by such directives.

Besides reconstructing the "last" consistent global state for a given global breakpoint, one now needs to be able to inspect the local states of individual processes. This can be achieved by integrating an event-based distributed debugger that allows global breakpoints and halting in consistent global states, with individual state-based sequential debuggers that may be applied to each application process in isolation.

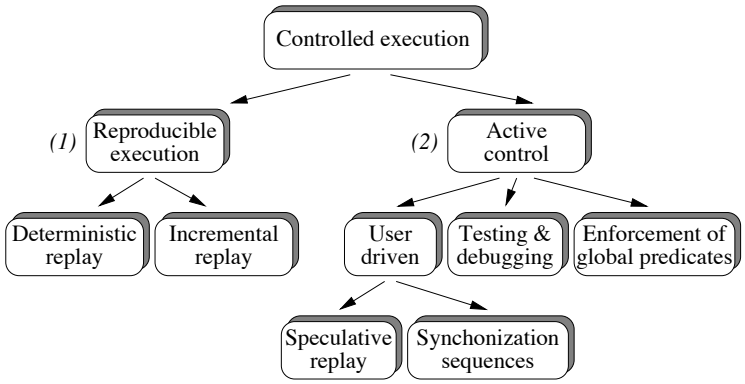
5.6.2 Control

Control of distributed computation is necessary for distributed debugging purposes in order to achieve the following objectives (Figure 5.13):

(i) Handle nondeterminism.

(ii) Enforce specific distributed computation paths.

(iii) Enforce specific global predicates.



(1) Follows / mimics a previously executed path
(2) Enforce a new generated path

Figure 5.13: Controlled execution

Due to the nondeterminism of distributed programs, multi-execution testing cannot be achieved by simply trying to repeat "free" uncontrolled program runs. This requires a deterministic re-execution. This technique is based on a form of passive control because the re-execution just blindly mimics the behavior of a previous run.

Additionally, more active forms of control are required (called *controlled execution* or *active control* [50]) because specific alternative distributed computation paths need to be pursued in order to examine the dynamic program behavior. A supervisory process controls event ordering, for re-execution, e.g., by exercising distinct message delivery orderings, or alternative process schedules, in order to investigate the consequences of nondeterministic execution.

Desired event orderings may be specified as event expressions or as command scripts that are interpreted by the distributed debugger. The goal is to generate and test all/some possible permutations of events. Such forms of active control allow a distributed debugging to perform a complementary role to static analysis and testing tools, in order to ease the evaluation of correctness properties through the exploration of the space of distributed computation paths. The specification of such paths can be:

(i) Given by the user.

(ii) Generated by an automated testing tool.

(iii) Automatically enforced by a global predicate evaluator/controller.

An example of approach (1) is given by the *speculative replay* technique [48]. During speculative replay, the user can select any of the potential consistent global states and ask the distributed debugger to re-execute the program using a message log. So the user can detect how different event orderings may originate different observable behaviors. The user may then add additional interprocess dependencies, and re-run the program under new synchronization constraints.

Another example is given by the use of *synchronization sequences* in an approach called *deterministic execution testing* [49]. This forces the execution of a concurrent program, with a given input, according to a user-defined sequence of synchronization commands, corresponding to a different test of the program, for a given input. Another use of a similar technique was mentioned in a previous section, where an annotated trace is generated by a distributed debugger in order to force the computation to follow a specific path leading to a consistent global state associated with a global breakpoint.

An example of approach (2) was discussed in a previous section, corresponding to the testing, active control and debugging methodology. This approach is the basis of the work done in the SEPP projects [31] on integration of testing and debugging tools which are further discussed in Chapters 9, 13 and 16. This approach also uses a partial specification from the user.

Approach (3) is related to the automated evaluation of global predicates that was also discussed in the section on distributed debugging methodologies. Its integration into a distributed debugging cycle only(!) requires the user to specify the correct program behavior in terms of global predicates. Then it is up to the distributed debugger to monitor the execution in order to detect or enforce the satisfaction of those logical conditions. The user can then inspect consistent global states and get more information on the causes of errors. At this point in the debugging cycle, the user may have new information on program behavior that allows to assert certain global properties that should be enforced by the global predicate controller [50, 52]. Then an active control phase begins where the controller is responsible for enforcing those conditions, thus allowing the investigation of other potential error situations. The third approach is the more ambitious one. Conceptually, it would allow having a distributed debugger controller automatically imposing the required synchronization constraints so that some high level specification of a distributed application would be enforced. Only a few distributed debuggers are able to enforce specific global predicates or assertions, and for very restricted expressions [52].

5.7 The Architecture of Distributed Debugging Systems

The design of a distributed debugging system must meet the requirements for basic observation and control functionalities, as discussed in previous sections. The architectures of most existing parallel and distributed debuggers have followed the approach of the p2d2 architecture [22] which is based on a client — server model. Additionally, existing debuggers provide very sophisticated visual and graphical user interfaces, and cover a wide range of functionalities for interactive debugging of individual distributed processes and threads [11].

In the recent past, some distributed debuggers have been developed aiming at being integrated into complete parallel software development environments. However, there are still many open issues concerning a successful tool integration [44].

Instead of an architecture providing a fixed set of functionalities, the software architecture of a distributed debugger should be extensible and allow adequate integration with other parallel software development tools. This is necessary so that the multiple methodologies and approaches for distributed debugging may be supported within a common architectural framework. In the SEPP projects, such an approach has been followed for the design of the DDBG debugger (see Chapter 13) and its results are discussed in related chapters in this book.

5.8 Conclusions

The distributed debugging activity still faces enormous difficulties to increase its impact upon the users. As mentioned in [20] referring to a PTOOLS report, about 90% of the parallel and distributed application developers using PVM still relied on classical "`print`" based approaches to debug their programs. Furthermore one doesn't known if the remaining 10% are really happy with currently existing distributed debugging tools (both commercial and academic). In this chapter, the main dimensions of distributed debugging have been discussed. The foundations of the distributed debugging activity were related to the theory of distributed computations in order to show the difficulties of distributed debugging, and explain why naive "`print`" based approaches do not work for distributed debugging.

Several methodologies for distributed debugging were presented which illustrate the complementary roles played by static analysis, testing, and dynamic analysis approaches, towards the understanding of the behavior of distributed programs. The Figure 5.14 summarizes the main dimensions of this view of distributed debugging. The main observation and control techniques were described, with emphasis to the interactive replay-based distributed debugging and how it promotes a cycle where the user repeatedly investigates and incrementally collects information to improve the analysis on the distributed program behavior. An ideal approach was also outlined for distributed debugging, starting with a specification of program correctness properties, followed by attempts to match them to the observed program behavior. Such approach was related to the interactive cyclic replay based distributed debugging approach, in order to allow the user to examine the relevant global computation states.

The research and development efforts in distributed debugging in the past decade have increased our belief that the distributed debugging activity only makes sense when adequately integrated with other complementary tools, for analysis, visualization, and program observation and control. So, besides highly sophisticated stand alone and autonomous distributed debugging tools [11], the future generation of distributed debuggers will require more and more flexible and extensible distributed debugging architectures, to ease the integration of distributed debugging functionalities with other tools. This requirement is even more important due to the current developments in homogeneous

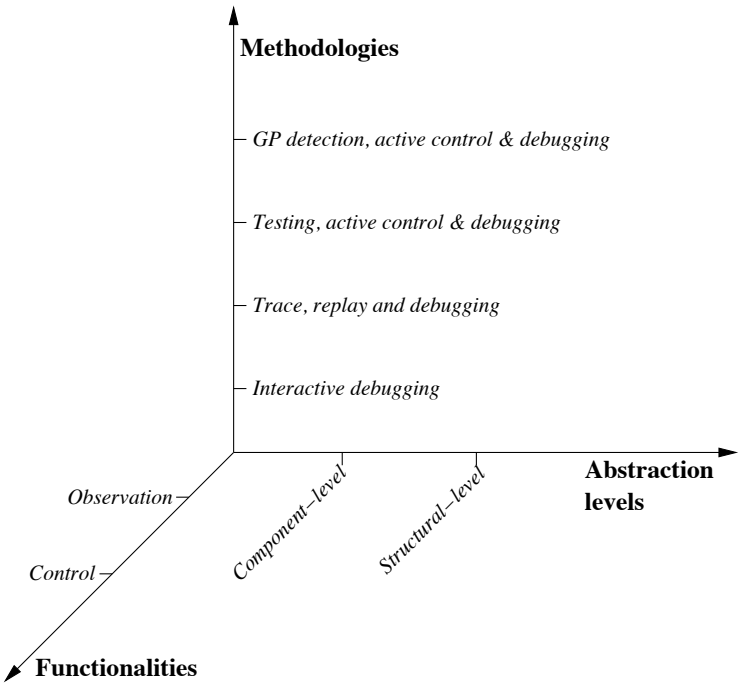


Figure 5.14: The user's view

and heterogeneous parallel and distributed computing platforms, including the cluster computing systems.

In the forthcoming years we will witness the appearance of standard definitions for distributed debugging application interfaces, along the line of the HPDF initiative [14]. We will witness an increasing trend towards distributed debugger's designs and architectures that can be adapted to the emerging parallel and distributed programming models, with component based and mobile computation abstractions.

Even with such high expectations for the near future, the impact of such approaches critically depends on how successfully they are integrated into easy to learn and to use tools, with friendly user interfaces, that meet practical user demands. Such topics have been increasingly discussed among users, tool developers and software engineers. This trend is also confirmed by current awareness towards considering the importance of the human factors for the design of the user interfaces and the functionalities provided by parallel development tools [15, 42, 44].

Acknowledgements

This work was partially supported by the Centre for Informatics and Information Technologies (CITI) and the Department of Informatics (DI) of FCT/UNL, by the PRAXIS XXI SETNA-ParComp (Contract 2/2.1/TIT/1557/95), and by the French Ambassy — INRIA/Portuguese ICTTI and the Hungarian/Portuguese Governments cooperation protocols.

References

[1] *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*. ACM Press, January 1988.

[2] *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, volume 26 of *ACM SIGPLAN Notices*. ACM Press, 1991.

[3] *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*. ACM Press, 1993.

[4] O. Babaoğlu, E. Fromentin, and M. Raynal. A unified framework for the specification and runtime detection of dynamic properties in distributed computations. Technical Report UBLCS-95-3, University of Bologna, Italy, June 1995.

[5] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. Addison-Wesley, second edition, 1993.

[6] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of ACM Workshop on Parallel and Distributed Debugging* [1], pages 11–22.

[7] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[8] C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[9] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* [3], pages 167–174.

[10] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software—Practice and Experience*, 22(10), 1992.

[11] Etnus Inc., Framingham, MA. *TotalView User's Guide (v3.9.0)*, June 1999. http://www.etnus.com/.

[12] A. Fagot and J. Chassin-de Kergommeaux. Optimized execution replay mechanism for rpc-based parallel programming models. Technical report, LMC-IMAG, Grenoble, France, 1995.

[13] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of ACM Workshop on Parallel and Distributed Debugging* [1], pages 183–194.

[14] J. Francioni and C. M. Pancake. High performance debugging standards effort. http://www.ptools.org/hpdf/.

[15] J. M. Francioni. *Debugging and Performance Tuning for Parallel Computer Systems*, chapter Determining the Effectiveness of Interfaces for Debugging and Performance Analysis Tools, pages 127–142. IEEE Computer Society Press, 1996.

[16] P. Fritzson, editor. *Automated and Algorithmic Debugging*, volume 749 of *LNCS*. Springer-Verlag, May 1993.

[17] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson. On the fly testing of regular patterns in distributed computations. In K. C. Tai, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 73–76, Boca Raton, FL, USA, August 1994. CRC Press.

[18] V. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 423–430, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press.

[19] V. Garg, C. Chase, J. R. Mitchell, and R. Kilgore. *Tools and Environments for Paralell and Distributed Systems*, chapter Efficient Detection of Unstable Global Conditions Based on Monotonic Channel Predicates, pages 195–226. Kluwer Academic Publishers, 1996.

[20] G. A. Geist. *Debugging and Performance Tuning for Parallel Computer Systems*, chapter Visualization, Debugging, and Performance in PVM, pages 65–77. IEEE Computer Society Press, 1996.

[21] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In Bruce D. Schriver, editor, *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, volume II (Software Track), pages 166–175. IEEE Computer Society Press, January 1988.

[22] R. Hood and D. Cheng. *Tools and Environments for Parallel and Distributed Systems*, chapter Accomodating Heterogeneity in a Debugger – a Client– Server Approach, pages 175–194. Kluwer Academic Publishers, 1996.

[23] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of IEEE/ACM Supercomputing'90*, pages 74–82. IEEE Computer Science Press, 1990.

[24] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates. Technical Report TR-967, IRISA, 1995.

[25] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* [3], pages 32–42.

[26] P. Kacsuk. Macrostep-by-macrostep debugging of message passing parallel programs. In *Proceedings of Tenth IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 527–532, Las Vegas, Nevada, USA, October 1998.

[27] P. Kacsuk. Systematic debugging of parallel programs based on collective breakpoints. In *Proc. of International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 83–96, Los Angeles, California, USA, May 1999.

[28] P. Kacsuk, R. Lovas, and J. Kovács. Systematic debugging of parallel programs based on collective breakpoints and macrosteps. In *Proc. of 5th International Euro-Par Conference*, pages 90–97, Toulouse, France, August/September 1999. Springer-Verlag.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[30] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[31] J. Lourenço, J.C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.

[32] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15(1):62–69, May 1992.

[33] D. C. Marinescu, J. E. Lumpp Jr., T. L. Casavant, and H. J. Siege. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, June 1990.

[34] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[35] C. E. McDowell. *Debugging and Performance Tuning for Parallel Computing Systems*, chapter Race Detection - Ten Years Later, pages 101–126. IEEE Computer Society Press, 1996.

[36] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comupting Surveys*, 21(4):593–622, December 1989.

[37] B. P. Miller and J.-D. Choi. Breakpoints and halting in distributed systems. In *Proceedings of International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 1988.

[38] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, volume 23 of *ACM SIGPLAN Notices*, pages 135–144. ACM Press, July 1988.

[39] R. H. B. Netzer. Optimal trace and replay for debugging shared-memory parallel programs. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging* [3].

[40] R. H. B. Netzer. Trace size vs. parallelism in trace-and-replay debugging of shared-memory programs. *LNCS*, 768, 1994.

[41] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the Symposiun on Parallel and Distributed Tools SPDT'96*, pages 31–40. ACM Press, 1996.

[42] C. M. Pancake. *Debugging and Performance Tuning for Parallel Computer Systems*, chapter Collaborative Efforts to Develop User-Oriented Parallel Tools. IEEE Computer Society Press, 1996.

[43] C. M. Pancake and S. Utter. A bibliography of parallel debuggers. *ACM SIGPLAN Notices*, 26(1):21–37, January 1991.

[44] D. A. Reed, J. S. Brown, A. H. Hayes, and M. L. Simmons. *Debugging and Performance Tuning for Parallel Computer Systems*, chapter Performance and Debugging Tools: A Research and Development Checkpoint. IEEE Computer Society Press, 1996.

[45] M. A. Ronsse and D. A. Kranzlmuller. Rolt-MP: Replay of Lamport timestamps for message passing systems. In *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*, pages 87–93, 1998.

[46] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[47] M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, editors. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Science Press, 1996.

[48] J. M. Stone. A graphical representation of concurrent processes. *ACM SIGPLAN Notices*, 24(1):226–235, January 1989.

[49] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. Software Eng.*, 17(1):45–62, January 1991.

[50] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 763–769, Los Alamitos, March 30–April 3 1998. IEEE Computer Society Press.

[51] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In Miller and McDowell [3], pages 21–31.

[52] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed systems. In *International Conference on Computer Systems and Education*, 1994.

[53] J. J. P. Tsai and S. J. H. Yang, editors. *Monitoring and Debugging of Distributed Real-Time Systems*. IEEE Computer Society Press, 1995.

[54] L. Wittie. Debugging distributed C programs by real time replay. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging* [1], pages 57–67.