A Taxonomy of Distributed Debuggers Based on Execution Replay [1]

Carl Dionne
Alex Informatique
Lachine (Québec)
Canada
dionne@alex.qc.ca

Marc Feeley
Université de Montréal
Montréal (Québec)
Canada
feeley@iro.umontreal.ca

Jocelyn Desbiens
INRS-Télécommunications
Iles-des-Soeurs (Québec)
Canada
desbiens@inrs-telecom.uquebec.ca

Abstract

This paper presents a taxonomy of parallel and distributed debuggers based on execution replay. Programming of distributed and parallel systems is a complex task. Amongst the many factors contributing to this complexity, the nondeterminacy of these systems is an important one. Execution replay is a technique developed to facilitate the debugging of nondeterministic programs.
Execution replay has very broad applications and not every algorithm is applicable in every situation. This taxonomy provides a precise classification of replay debuggers using nine criteria. From this classification, it is easier to determine a debugger's scope of application, outline its strengths and weaknesses and compare it with others. This taxonomy is illustrated and validated using a collection of existing replay debuggers.

Keywords: debugging, nondeterminism, execution replay

1 Introduction

It is well known that programming of distributed and parallel applications is a complex task. Furthermore, very few parallel programming tools are available to support the programmer. In particular, debugging tools are often not well suited or simply lacking.

Tools used in sequential programming environments may not scale appropriately to distributed and parallel environments. Mellor-Crummey cites four reasons that make debugging parallel systems more problematic: lack of global time, nondeterminism, multiple threads of control and complex patterns of interactions [10]. Innovative strategies have to be developed to build tools well adapted to these difficulties. Execution replay is a strategy proposed to facilitate debugging of nondeterministic programs.

[1] Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), Sunnyvale, California, August 9–11, 1996

1.1 Related work

Our taxonomy aims at providing a terminology of replay debugger characteristics. The need for a debugging terminology has been formally identified in the conclusion to the 1988 workshop on distributed and parallel debugging by Sopka and Redell. Both concluded that we lacked a theory and a terminology for debugging [10].

Regarding replay debugging more specifically, important work has been accomplished since, but we are still far from a general theory and terminology. Netzer proposed a taxonomy of race conditions [12], extended by Helmbold and McDowell in 1994 [7]. A better comprehension of race condition properties lead to replay algorithms optimal in terms of the quantity of logged information [11, 13].

Our taxonomy widens the scope of these race condition taxonomies, by caracterizing other aspects of a replay debugger. We cover the characteristics of the replayed system, the replay algorithm (where the type of race condition replayed is included), the integration of the replay debugger into the replayed system as well as the characteristics of the resulting replay itself.

2 Terminology

2.1 Execution replay

Execution replay aims at providing an effective method to debug nondeterministic programs. We say that a program P has a nondeterministic behavior if two executions with the same input may differ. With execution replay, information is gathered during an execution E and used to control another execution E', called replay, in such a way that E' is identical to E at some level of abstraction. Clearly, the replay E' is not identical to E at all levels: one produces information while the other uses it. However, this replay helps to locate the fault if:

- Information may be gathered during the replay. This information may be obtained from additional instructions in the code (e.g `printf`), using standard sequential debuggers or with more sophisticated tools. Further, the information should be gathered at a level of abstraction where executions are identical.
- The information gathered at this level of abstraction is sufficient to locate the fault.

We illustrate these requirements with two examples. The first example illustrates a useless replay debugger. At a very high level of abstraction, it simply reproduces the output of an execution of a program: if the execution outputs the result *5*, the replay simply returns *5*. This is a correct replay at a very high level of abstraction. At this level of abstraction, the only information that can be gathered is its output, and this information is useless to debug the program. Furthermore, even if some information could be gathered at a lower level of abstraction, this information would be useless since the executions differ at this lower level of abstraction.

The second example illustrates a situation where replay is useful. Replay debuggers generally operate at a level of abstraction corresponding to that of the primitive operations of the language. A regular sequential debugger is usually available to gather information. Functions like `printf` could also be added to the code to gather information. If the replay is correct at the language level, information is gathered at a level of abstraction where executions are identical. This information is useful to debug the program. Of course, nondeterminism is not the only difficulty of parallel and distributed debugging: lack of global time, multiple threads of control and complex patterns of interaction might make it complex to track down the fault using the information gathered. Other debugging techniques should then be used in conjunction with replay, but further analysis is beyond the scope of this paper.

2.2 Other terms related to replay

Execution replay is applied to a particular language or system. We call that language or system the *replayed system*.

We also distinguish between *replay algorithm* and *replay debugger*. A replay algorithm describes how a particular set of instructions is replayed. A replay debugger incorporates one or more replay algorithms, as well as more practical aspects such as the integration into the replayed system.

A replay debugger is also often a component of a more complete *debugging system*: execution replay is helpful in debugging if additional information is given during the replay, this information being gathered by the other components of the debugging system.

3 Existing Replay Systems

Before proceeding with the taxonomy itself, we first present a survey of existing replay debuggers. These debuggers will be used to illustrate the various classification criteria. Their classification is summarized in table 1.

RD 1 (BugNet [1]) BugNet is a replay debugger developed by Curtis and Wittie. It is used to replay applications running on a workstation cluster. The application's tasks communicate through interprocess communication (IPC), where unpredictable communication delays cause nondeterminism. All I/O and IPC data exchanges are captured, their data recorded and used to create the replay.

RD 2 (Recap [14]) Recap is a debugger that provides the illusion of reverse execution. It logs and replays the results of systems calls and shared-memory read, as well as the time when asynchronous events (signals) occur. Given (1) an initial execution passing through states t_c, t_p and t_n, (2) a checkpoint of the execution state at t_c and (3) a replay stopped at t_n, it creates the illusion of reverse execution to time t_p by replaying (forward) the execution from t_c to t_p.

RD 3 (Instant Replay [9]) Instant Replay is an algorithm developed by Leblanc and Mellor-Crummey to replay shared-memory parallel programs. Instant Replay assumes that all accesses to shared variables are guarded. The replay is guaranteed by recording the order in which the variables are accessed, rather than recording the data that is accessed. Several adaptations of Instant Replay were proposed to debug other types of nondeterministic constructs (RD 8, 9).

RD 4 (SYN-Sequence [16]) Tai, Carver and Obaid proposed a replay debugger for concurrent Ada programs. Given a program P, source-to-source transformations produce the programs P' and P'', such that (1) the execution of P' records a sequence of synchronization events (SYN-Sequence) and (2) P'' uses this SYN-Sequence to produce a correct replay. Their debugger is concerned with all nondeterministic Ada constructs, except accesses to shared-variables: it is assumed that these accesses will be guarded using other Ada constructs.

RD 5 (Optimal Tracing of Shared Memory Accesses [11]) Netzer proposed an enhancement to the original Instant Replay algorithm. This enhancement is based on the observation that it is usually more expensive to record all log entries than to compute a subset of them that is sufficient to guarantee replay. With Instant Replay, all accesses to shared variables are logged. Netzer proposes an algorithm that computes, dynamically, a subset of accesses that is sufficient to guarantee a correct replay, and shows that this subset is minimal.

RD 6 (Optimal Tracing of Messages [13]) Based on the previous observation, Netzer and Miller proposed an algorithm applicable to message-passing programs that dynamically computes the minimal subset of events that needs to be recorded.

RD 7 (Mostly Functional Language [6]) Halstead and Krantz proposed a replay algorithm adapted to mostly functional parallel programs. The algorithm has been implemented for Multilisp, a parallel variant of Scheme. Like Instant Replay, their algorithm reproduces the order in which shared-variables are accessed. But it exploits characteristics of mostly functional languages to reduce the intrusiveness of the debugger: no tracing overhead is imposed on read accesses to variables (even shared-variables), only a small amount of tracing overhead is imposed to Multilisp's touch operation, but greater overhead is added to side-effect operations, that are presumed to be rare.

RD 8 (Concurrent Logic Language [15]) Shen and Gregory extended Instant Replay to concurrent logic programs. Their replay debugger is applied to KLIC. Committed choice concurrent logic programming languages have several properties that they exploit in order to simplify the replay algorithm. The most important property is that KLIC's processes communicate through single-assignment shared-variables. Nondeterminism of KLIC's executions occurs because the clause to which each goal (or process) commits during an execution is nondeterministic. Tracing this information is sufficient to create a correct replay.

RD 9 (Actor Language [3]) Dionne proposed a replay debugger applied to CLAP[2], an actor extension of C++. CLAP's multi-threaded actors communicate through both shared-variables and messages. This replay debugger reproduces some nondeterministic instructions of CLAP with an extension of Instant Replay, and combines the logging of data to reproduce other nondeterministic instructions such as clock accesses or user input.

RD 10 (Distributed Training System [4]) Dionne proposed a debugger to replay executions of a distributed training system composed of a user interface, a simulation engine and a collection of artificial agents. The replay debugger is applied to messages sent across the system: their data is recorded and used to replay a process by simulating its interactions with other processes.

Debugger	Instructions	Task	Alg	Inst. traced	Int.	Time	Proc.
(RD 1)	$1b$, input	$2a$	$3a$	$4a$	$5b$	$8b$	$9c$
(RD 2)	$1b$, input	$2a$	$3a$	$4a$	$5b$	$8b$	$9c$
(RD 3)	$1a$	$2a$	$3b$	$4a$	$5a$	$8a$	$9a$
(RD 4)	$1a$, $1b$	$2a$	$3a$	$4a$	$5a$	$8a$	$9a$
(RD 5)	$1a$	$2a$	$3b$	$4b$ (races)	$5a$	$8a$	$9a$
(RD 6)	$1b$	$2a$	$3b$	$4b$ (races)	$5b$	$8a$	$9a$
(RD 7)	$1a$	$2b$	$3b$	$4b$ (side effects)	$5b$	$8a$	$9a$
(RD 8)	goal commitment	$2b$	$3b$	$4a$	$5b$	$8a$	$9a$
(RD 9)	$1a$, $1b$, input	$2b$	$3b$,$3a$	$4a$	$5a$	$8a$	$9a$
(RD 10)	$1b$	$2a$	$3a$	$4a$	$5b_p$	$8a$	$9c$

Table 1: Classification of some replay debuggers. Two criteria were omitted: the failure of the replay (criterion 6) is not a concern in most debugger papers, but we believe that most debuggers are of the type $6b$, and all debuggers are of type $7a$ regarding the class of instrumented instructions (criterion 7).

4 Taxonomy

The taxonomy we are proposing is composed of nine criteria, describing a debugger from four different perspectives: the characteristics of the replayed system, the characteristics of the replay algorithms used by the debugger, the integration of the replay debugger into the replayed system and the characteristics of the resulting replay.

We consider that the integration of the replay debugger into a larger scale debugging system is not a characteristic of the replay debugger, but rather of the debugging system. Thus, aspects such as the interface to visualization packages, user interface and so on are not relevant in this classification.

The replay debuggers previously described illustrate the classification criteria.

4.1 Characteristics of the replayed system

This first perspective captures the aspects of the replayed system relevant to the classification of a replay debugger.

Criterion 1 (Type of instruction) *We first classify the replay debuggers according to the type of instructions they replay. The value set defined by this criterion is not formally introduced. Typically, either accesses to shared-variables (*$1a$*) or message exchanges (*$1b$*) are the instructions that provoke nondeterminism, and thus the instructions replayed.*

Not every instruction of the replayed system has to be considered. We distinguish three classes of instructions[2]:

- *Deterministic instructions*: instructions that always produce a deterministic effect (do not require replay at all);
- *Weakly nondeterministic instructions*: instructions that produce a deterministic effect if they are executed in a particular order;
- *Strongly nondeterministic instructions*: instructions that do not produce a deterministic effect.

Replay debuggers are not concerned with deterministic instructions. Strongly nondeterministic instructions occur both in parallel and in sequential programs: access to a timer, user input, etc. Some debuggers do consider them (RD 1, 2, 9), but they are often ignored. Weakly nondeterministic instructions are generally the ones that make debugging parallel programs such a hard task. Replay debugging has mostly been applied to this class of instructions. Typically, replay debuggers deal with instructions related to shared-variables accesses (RD 3, 5, 7, 9) or to messages exchanges (RD 1, 2, 4, 6, 9, 10). Some systems consider a different set of instructions: KLIC's (RD 8) reproduces the commitment of a goal to a clause, CLAP's (RD 9) deals with message-passing and shared-variable access, but also with instructions related to control of the actor's behavior.

As more complex distributed applications are developed, execution replay may have to be applied to systems where a wider set of instructions cause nondeterminism. Access to more complex shared ressources (databases, files) and complex user interactions are examples of instructions that may have to be dealt with in the future.

Criterion 2 (Task creation model) *The replayed system's task creation model is an important classification criterion. It describes whether the replay debugger requires a static task creation (2a), or handles a dynamic task creation (2b).*

We distinguish two task creation models: *static* and *dynamic*. In systems where task creation is static, the number of tasks is known when the program is started. In systems where it is dynamic, that number may change during the execution. The implementation of a replay debugger that handles the dynamic creation of tasks is more complex.

Replay debuggers for CLAP (RD 9), Multilisp (RD 7) and KLIC (RD 8) support dynamic task creation. It is a fundamental aspect of the systems that they replay: in CLAP, an actor processes messages with new threads, in Multilisp, parallelism is expressed with an expression resulting in the creation of a new task (a short execution could result in tens of thousands of tasks being created) and in KLIC, every goal corresponds to a task and additional tasks are created as subgoals are found.

The replay debuggers proposed by Netzer (RD 5, 6) rely on a fixed number of tasks: every shared ressource is associated with a vector of size t, where t is the number of tasks. If tasks are dynamically created, the value of t changes. Thus, debuggers (RD 5, 6) require a static task creation model.

[2]A formal definition of these three classes of instruction may be found in [3]

The replay debugger (RD 10) does not handle dynamic task creation. It was not required since the underlying communication system does not permit the attachment or creation of tasks. If dynamic task creation was allowed, the event identification method would have to be adapted accordingly. In general, an event is uniquely identified within a specific context. This context is usually the task: events of different tasks are logged into different files, or the event identification within the log file describes the task from which the event originated. The identification of every task or the association of every task to a log file is usually easy when tasks are statically created. But this might be nondeterministic when tasks are dynamically created. This adds to the complexity of the debugger.

In the case of the other debuggers (RD 1, 2, 3, 4), it is not clearly indicated whether they support dynamic task creation or not. They do not attach a vector of information to the shared ressources, but dynamic task creation is not fundamental to the replayed system. A fair assumption would be that the previous discussion about (RD 10) holds true for these debuggers.

4.2 Type of algorithm

We have discussed the characteristics of the replayed system. We now focus on the replay algorithm itself. A replay debugger could combine many replay algorithms. Such a replay debugger would be described by the characterization of every algorithm it employs.

Criterion 3 (Data or synchronization replay) *Replay algorithms are either based on the data (*3*a) or the synchronization (*3*b) of the instructions.*

Data replay was the first technique used, by BugNet (RD 1). It is also the technique used in (RD 2). Since synchronization replay is usually more efficient, it is the technique used by most recent algorithms (RD 3, 4, 5, 6, 7, 8, 9). One of these debuggers (RD 9) also employs a data replay algorithm to replay nondeterministic instructions (that, by definition, require data replay).

Unfortunately, synchronization algorithms require that all tasks be replayed, a condition that is not always feasible. For example, in (RD 10), a component of the system (a simulation engine) could not be replayed. Thus a data algorithm was required.

Criterion 4 (Class of traced instructions) *This criterion describes the relationship between the instrumented instructions and the traced instructions: whether information is traced upon the execution of every instruction instrumented (*4*a), or only upon the execution of a subset of these instructions (*4*b).*

Run-time knowledge of the execution may drastically reduce the size of the log. Two approaches are illustrated in the debuggers we use as examples: (RD 5, 6) do not log unecessary information upon the execution of instructions inherently ordered, while (RD 7) optimizes the replay of mostly functional programs by considering that most accesses are read accesses, and that only information about variables that were written needs to be logged. The other debuggers log information upon the execution of every instruction that is instrumented.

4.3 Integration into the replayed system

The next family of criteria describes the integration of the replay debugger into the replayed system.

Criterion 5 (Integration method) *This criterion caracterizes the method used to integrate the replay debugger into the system: the integration is manual (*$5a$*) or automatic (*$5b$*). We further distinguish between complete (*$5b_c$*) and partial (*$5b_p$*) automatic integration.*

Integration is manual when the user is involved in the instrumentation of the program. In this case, it is the responsibility of the programmer to identify nondeterministic instructions in the program. When integration is manual, it is not possible to guarantee that the replay is always correct: the user could have forgotten to identify an instruction, for example. This type of integration is paradoxical in the sense that the debugger relies on the premise that the program is free of a certain type of bug.

Integration is automatic when the user is not involved in the instrumentation process. In this case, the integration might be done during the compilation (by a preprocessor, the compiler or the interpreter), or at run-time (by instrumenting the library). With automatic integration, programs will be replayed without extra effort from the user. In the case where integration is automatic, we further distinguish between complete and partial automatic integration. If integration is complete, then replay is guaranteed for any valid program of the given language.

Debuggers of the type $1a$ usually require manual integration. In (RD 9), shared-variables have to be instances of a particular class, and standard C++ access methods are overloaded and instrumented. In (RD 3, 4, 5), shared-variable accesses are guarded with particular instructions. As an exception to this rule, (RD 7) is automatically integrated into the language.

Debuggers of type $1b$ usually provide automatic integration. This is true of (RD 1, 2, 6, 10). (RD 4, 9) belong to both types $1a$ and $1b$. Their integration is automatic with respect to the message exchange instructions, but manual with respect to shared-variable accesses. Overall, their integration is manual since the user's intervention is required. (RD 8) is of neither type and is automatically integrated.

Most papers on replay debuggers are concerned with the instructions that they replay, and do not mention those that they do not replay. Therefore, the classification of debuggers into the types $5b_c$ and $5b_p$ is usually not possible. We believe that most debuggers, if not all, are only partially integrated.

Criterion 6 (Failure of the replay) *Only a replay debugger with complete automatic integration (of type* $5b_c$*) may guarantee a correct replay of any program. Generally, there exists programs and situations such that the replay may fail. This criterion characterizes the failure detection: whether it is never detected (*$6a$*), sometimes signaled to the user (*$6b$*) or always signaled to the user (*$6c$*).*

The failure of the replay debugger is a concern that very often remains unanswered. Most papers on debuggers describe the replayed instructions, how the replay is achieved and what type of replay is achieved. Sometimes, the integration of the replay debugger into the replayed system is described. But most of the time, the behavior of the debugger when the replay fails is not described.

The failure may occur in many situations. Where integration is manual, the user could have forgotten to identify a nondeterministic instruction. Where integration is automatic but incomplete, the execution of a particular instruction might cause the replay to fail. This includes simple cases such as inputs that are not taken care of, but also less trivial cases, such as memory exhaustion occuring only during the replay.

A failure of the replay in (RD 9, 10) will sometimes be signaled to the user, but this is not guaranteed. The failure is detected when an access to the log is performed, and if the event parameters do not match the original ones (for example, event types do not match).

Criterion 7 (Class of instrumented instructions) *This criterion describes the relation between the program and the instrumented instructions. It indicates whether all instuctions of a given type are instrumented (7a), or if only a subset of these instructions are instrumented (7b).*

Run-time information analysis leads to debuggers of type 4*b*, that do not trace every instrumented instruction. Static analysis of the program may reduce the number of instructions that need to be instrumented. This analysis is particularly important when a high proportion of the instructions of a given type do not have to be traced (but not all of them, since we assume that some of these instructions are nondeterministic). A good example of such instruction types are accesses to variables: instances of accesses to shared-variables are weakly nondeterministic and need to be traced, while instances of accesses to variables that are not shared are deterministic, and do not need to be traced. A static analysis of the program could be used to determine which variables are shared and which are not, and enable the restriction of the instrumentation to only the subset of accesses performed to a shared-variable.

Of the debuggers we use as examples, none use static information to reduce the proportion of instrumented instructions. This is not an important feature for debuggers that replay instructions that almost always have to be instrumented (message-passing instructions, for example). But it is interesting to see how instrumentation of shared-variables accesses is performed. Most debuggers operating on shared-variables require them to be protected (RD 3, 5, 9, 4). Thus the responsability of identifying shared-variables belongs to the user. In (RD 7), all variable accesses are instrumented, but run-time information is used to distinguish those that really need to be traced.

To our knowledge, static information has not yet been used to distinguish instructions that need to be instrumented. But we believe that as automatically integrated debuggers replaying shared-variable languages are built, debuggers using static information will start to appear.

4.4 Resulting Replay

The last family of criteria categorizes the type of replay obtained with the debugger. A replay occurs in two different dimensions: in time and in task space. These two dimensions are used to define two classification criteria.

Criterion 8 (Range in time) *This criterion indicates whether replay occurs from a predetermined point in time (static time range, 8a) or if the start of the replay may be changed (dynamic time range, 8b).*

Replay from a predetermined point in time is usually provided with a mechanism to create checkpoints whereupon the program may be restarted. Debuggers characterized with a dynamic time range may be used as a support for *reverse execution* (RD 2). Reverse execution gives the illusion that the flow of execution is reversed, which might be a more natural way to track down the reason of a failure. There is no direct relation between dynamic time range and reverse execution: (RD 1) is characterized with a dynamic time range but does not support reverse execution, and IGOR, although not a replay debugger, supports reverse execution of deterministic programs [5]. We do not discriminate replay debuggers according to this criteria (whether they support reverse execution or not): we consider that it is rather a characteristic of the debugging system to which the replay debugger belongs.

Of the debuggers we use as examples, (RD 1, 2) are characterized with a dynamic time range while the others are characterized with a static time range.

Criterion 9 (Range in task space) *This last criterion describes the set of tasks collaborating during the replay: all tasks (9a), a subset (9b) or only one task (9c).*

Our definition of collaboration is based on Lamport's *happened before* relation [8]. We say that tasks collaborate if and only if, for every pair of events ordered in the initial execution, the ordering relation also holds true in the replay. For example, if two tasks (t_1 and t_2) initially exchanged a message (m is sent from t_1 to t_2), and if, during the replay, m cannot be received by t_2 before t_1 sends it, we say that the ordering relation holds true for events $send(m)$ and $receive(m)$. If this relation holds true for every pair of events of the replayed tasks, we say that they collaborate. Synchronisation replay requires that all tasks collaborate during the replay. Most data replay algorithms do not require collaboration between tasks: since the content of m was recorded, there is no need to wait for $send(m)$ before $receive(m)$ is replayed.

The replay of a set of collaborative tasks enables information gathering about global states (distributed accross many tasks). Replay of a single task requires less resources, but observation of global states is not feasible.

(RD 1, 2, 10) are examples of debuggers that replay non collaborative tasks. We plan to extend (RD 10) in order to replay a subset of collaborative tasks. The other debuggers are based on synchronization, and require that all tasks be replayed.

5 Discussion

The juxtaposition of the debuggers survey with the taxonomy enables us to identify areas of potential research.

5.1 Automatic integration

Debuggers with manual integration (5a) present an interesting paradox: they guarantee a correct replay if the user correctly identified the sources of nondeterminism in the code, that is, the replay is correct only if the program is free of a certain type of bug. Because of this paradox, we believe that debuggers should aim for automatic integration (5*b*).

We saw that debuggers for languages where shared variables are the source of nondeterminism are particularly hard to integrate automatically, therefore most debuggers of type 1*a* are also of type 5*a*.

Debuggers of the type 4*b* start to enable the automatic integration of shared variables. An example was shown with (RD 7): (1) this replay debugger is concerned with shared-variables (1*a*), (2) an algorithm of type 4*b* was designed, based on the asumption that the programs are mostly functional, and (3), the debugger is automatically integrated to Multilisp programs (5*b*).

Replaying shared-variable programs becomes more and more important with the advent of new multi-threaded operating systems. We have also seen that automatic integration should be aimed for. These two observations imply that future replay debuggers will be integrated with the compilation of the program, rather than using specially instrumented libraries. Shared-variable accesses might be difficult to distinguish from nonshared-variable accesses, and implementation of debuggers of type 7*b* is an alternative that needs to be explored. Since shared-variable accesses are usually short and frequent, the debugger should also be of the type 4*b*.

5.2 Type of instruction

Replay systems have mostly been proposed in parallel computing environments. However, as the usage of clusters of computers increases, effective solutions will also be required to debug nondeterministic distributed systems. We believe that these distributed systems will require the replay of a wider range of instructions than those typically addressed: access to distributed databases, complex interaction with users and so on.

It might not be feasible to integrate all instructions that potentially cause nondeterminism. If complete automatic integration ($5b_c$) is not feasible, then the replay might differ from the initial execution. In this case, the planning of the failure of the replay becomes an essential feature of the debugger. We believe that debuggers should aim for criterion 6*c*, where a failure of the replay is always detected and signaled to the user. Using such a debugger, the user will be notified if the replay fails, and will not make false deductions.

6 Conclusion

In this paper, we have proposed a taxonomy of replay debuggers. This taxonomy uses four major categories of criteria: the characteristics of the replayed system, the integration of the replay debugger to the replayed system, the characteristics of the algorithm as well as the characteristics of the replay itself. A review and classification of existing replay debuggers demonstrates the proposed taxonomy.

Nondeterminism is an important problem in parallel and distributed computing and an effective replay debugger certainly proves itself very useful. This taxonomy is a classification tool to compare debuggers and measure their effectiveness. As such, it is an important indicator of the work accomplished to date and it reveals research directions that will lead to debugging tools that achieve their real objective: to find *real* bugs in *real* systems.

References

[1] R. Curtis and L. Wittie. BugNet: A debugging system for parallel programming environments. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 394–399, 1982.

[2] J. Desbiens, M. Lavoie, S. Pouzyreff, P. Raymond, T. Tamazouzt, and M. Toulouse. CLAP: An object-oriented programming system for distributed memory parallel machines. In *Proceedings of the PARCO'93 Conference*, pages 601–604, Grenoble, France, 1994.

[3] C. Dionne. Un dévermineur permettant la réexécution d'un langage de programmation parallèle de type acteur. Master's thesis, Université de Montréal, 1996.

[4] C. Dionne, M. Nolette, and D. Gagné. Message passing in complex irregular systems. In *Proceedings of the MPI Developers Conference and Users Group Meeting (MPIDC'96)*, July 1996.

[5] S. I. Feldman and C. B. Brown. IGOR: A system for program debugging via reversible execution. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):112–123, January 1989.

[6] R. H. Halstead, Jr. and D. A. Kranz. A replay mechanism for mostly functional parallel programs. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 119–130, Tokyo, Japan, April 1991.

[7] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, Computer Research Laboratory, September 1994.

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

[9] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[10] T. J. LeBlanc and B. P. Miller. Workshop summary. In *Proceedings of the Workshop of Parallel and Distributed Debugging*, pages ix–xxi, Madison, Wisconsin, 1988.

[11] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, San Diego, California, May 1993.

[12] R. H. B. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 251–253, Santa Cruz, California, May 1991. [Extended abstract].

[13] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.

[14] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):124–129, January 1989.

[15] K. Shen and S. Gregory. Instant Replay debugging of concurrent logic programs. *New Generation Computing*, 14(1):79–107, 1996.

[16] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.