On Distributed Systems Debugging

Jiří Patera, Jiří Šafařík
Department of Computer Science and Engineering
University of West Bohemia, Univerzitní 8, 306 14 Plzeň, Czech Republic
e-mail: {jipatera, safarikj}@kiv.zcu.cz

Abstract

Software engineers have to face many problems when creating, testing and debugging their applications. One of the most challenging problems today is closely associated with the modification of distributed systems. Even a small modification of such a system can considerably change its behavior. This paper explores methods which play an important role in the research on debugging parallel and distributed systems. We will study the detection of global state predicates and the use of temporal logic operators. Finally, on-line and off-line approaches to the problem of debugging a distributed system will be discussed. These include real-time monitoring and deterministic replay.

1. Introduction

The importance of using distributed systems is becoming more and more important not only with the growth of the Internet, but also by their usage in real-time embedded systems. However, today's complex distributed systems are difficult to design without bugs (mismatches between expected and actual computations). Although, programmer's skills and intuition play a very important role during the design process, the existence of tools (debuggers), that provide observation and control of a computation, is essential.

Testing and debugging are two terms which are often mixed up. The purpose of *testing* is to show whether the program has bugs (faults), while the purpose of *debugging* is to locate and remove sources of the bugs, which might have lead to a failure followed by a potential error.

Research on distributed systems debugging has focused on two main areas: *detecting bugs* in a distributed computation, and *replaying* a traced distributed computation. The techniques of detecting bugs depend on the types of bugs. An important work in this area is a global snapshot algorithm [2] which is used to detect stable properties (e.g. deadlock or termination). Since then, many algorithms for different classes of bugs have been designed.

For example, race conditions (when two or more processes are competing for an access to one resource), or evaluating predicates on single global states [3]. Research in replaying traced computations focused mainly on reducing the size of the trace data [4]; where only messages, which are not causally related, are necessary for an exact reproduction of the execution.

One of the aims of this paper is to investigate the issues that make debugging of parallel and distributed systems so hard. Furthermore, this paper presents a survey of a state of the art in the field of distributed systems debugging. Finally, we point out areas for future research using the major part of the paper as a background.

2. Model of Computation

A distributed system consists of n sequential processes $p_1, p_2, \ldots, p_n$ which can send messages to each other over reliable or unreliable channels. The system is asynchronous and has no shared memory. Channels can have either FIFO or non-FIFO character.

Let us have n processes $p_1, p_2, \ldots, p_n$. As we know, each process consists of a series of

The research was supported by a grant of the Grant Agency of the Czech Republic – Research of methods and tools for verification of embedded computer systems, no. 102/03/0672.

events. Thus, we can describe (like in [1]) an execution of each process by its *history*. First k events in a process's history are called a *prefix* of the process's history. We can also form a *global history* of a distributed system as the union of individual process histories.

A *cut* of a system's execution is a subset of its global history which is formed as a union of prefixes of individual process histories. A set of events is called a *frontier* of the cut, if only the last events from all local prefixes are included there.

The cut is *consistent* if for each event it contains, it also contains all the events that happened-before that event. Otherwise, the cut is said to be *inconsistent*. A *consistent global state* is then such a global state that corresponds to a consistent cut.

An *execution* of a distributed system is a series of transitions between global states of the system. In fact, we can have as many global states as the number of events in the system is, some of these states are consistent, and some are not. A *run* is a total ordering in a global history of the system which is consistent with each local history's ordering. Apart from that, there is a *linearization* (also known as a *consistent run*) which is consistent with the global history's ordering.

To sum up, not all runs pass through consistent global states, but all linearizations do pass through consistent states only. We say that a state S' is *reachable* from a state S if there is a linearization that passes first through S and then S'.

3. Global State Predicates

Detecting a condition in a distributed system equals to evaluating a *global state predicate*. The global state predicate is a function that maps from a set of global states in the system to $\{true, false\}$. A *local predicate* is such a predicate whose truth value depends only on the state of a single process. The four well-known classes of global predicates are stable, observer-independent, linear, and semi-linear.

Stable predicates [2] are usually associated with distributed problems such as deadlock or termination. Once the system enters a state where the predicate is evaluated to *true*, it remains *true* also in all future states that the system can enter. On the other hand, when we monitor or debug an application, we are often interested in *non-stable* predicates. These predicates can evaluate to *true* only in certain states of the computation and they need not be evaluated to *true* in any future state.

Observer-independent predicates [10] are such predicates where *possibly* Φ and *definitely* Φ are equivalent. The name comes from a set of observers where each witnesses a different sequential execution of the system. Each observer can determine if the predicate became true in any of the cuts witnessed by them. If the predicate is observer independent, then all observers must agree on whether it was ever evaluated to true. Any stable predicate is also observer-independent.

The definition of a *linear* predicate [11] is based on a "forbidden" state S (the predicate Φ in the cut containing state S must remain false until a successor to S is reached). A predicate is linear if for any cut, in which the predicate is false, at least one of the states is forbidden. In addition, there is a unique first cut where every linear predicate is true. [11] also gives explanations why linear predicates are not necessarily stable or observer-independent. A special class of linear predicates are *regular* predicates. A predicate is regular if the set of global states that satisfy the predicate forms a sub-lattice of the global state lattice.

The class of *semi-linear* predicates, first proposed in [11], contains all the three previous classes. Its definition is based on a "semi-forbidden" state S, which is irrelevant to the truth-value of the predicate. While we are looking for a cut where the predicate is true, we can disregard S in favor of its successor.

From another point of view, there are two classes of predicates, *conjunctive* and *disjunctive* predicates. In the former class, local predicate formulas are connected using $\bigwedge$ operator only to form a global predicate. In the latter class, it is $\bigvee$ operator instead. As [11] proves, a conjunction or disjunction of stable predicates is also a stable predicate. A disjunction of observer-independent predicates is again an observer-independent predicate, but this does not hold for a conjunction. Regarding linear predicates, the opposite is true. A conjunction of linear predicates is a linear predicate, but a disjunction is not. Semi-linear predicates are not closed neither under a conjunction, nor under a disjunction.

3.1 Predicate Detection

Next, we examine the problem of finding out whether a *transitory state* occurred in an actual execution (a non-stable predicate was evaluated to true). This is what we require when debugging a distributed system. The aim is to determine the cases where a global state predicate was *definitely true* at some point of the execution, and cases where it was *possibly true*.

Detection of a stable predicate is trivial. It is sufficient to detect it in any one consistent state of the system and we know for sure that it will remain true until the final state. Apart from that, the detection of non-stable predicates is more complicated.

Next, we describe operators *possibly*, *definitely* [1], *controllable*, and *invariant*. For a predicate Φ in terms of linearizations L of the global history of the system's execution H.

possibly Φ means that there is a consistent global state S through which L of H passes such that Φ is true in S.

definitely Φ means that for every L of H, there is a consistent global state S through which L passes such that Φ is true in S.

controllable Φ means that Φ is true in every consistent state along some L of H.

invariant Φ means that Φ is true in every consistent state along every L of H.

Let us note that we may conclude *definitely*$(\neg\Phi)$ from $\neg$*possibly*(Φ). However, we may not conclude $\neg$*possibly*(Φ) from *definitely*$(\neg\Phi)$ (predicate $\neg\Phi$ holds at some state on every linearization, nevertheless, Φ may hold at the other states).

Generally, there are three approaches to the detection of global predicates. In the first one, global snapshot algorithm [2] is used repeatedly until a consistent state where predicate Φ holds is found. This approach can be used for detection of stable predicates only.

Secondly, the construction of a global state lattice [12] is used. The lattice captures the reachability relation between consistent global states of a distributed system. Nodes denote consistent global states, and edges denote possible transitions between the nodes. This lattice is then explored and evaluated in order to detect stable and unstable predicates. This detection can be very expensive. In the system with n processes where each process has m local states, this approach requires exploring of $O(m^n)$ global states in the worst case.

In the third approach, the whole lattice is not constructed. Instead only a subset of global states, based on the structure of the predicate, is identified. For example, the predicate may depend only on the states which are related to a particular variable and so the detection is more efficient than in the second approach.

When the global lattice is created, its exploration checks whether a predicate is satisfied for all possible computations (linearizations) of a program. This approach is being referred as a *model checking*. Next approach to predicate detection checks only the global states that occurred in one particular computation. The latter approach is close to a distributed systems debugging where only a single execution trace is captured. Even if the model checking were used on a single computation, the complexity of detecting a predicate would be still proportional to the size of the lattice which is exponential in the number of processes.

In the *predicate detection* approach, we are only detecting whether a given predicate ever became true. Apart from that, the *predicate control* problem states that given a distributed computation and a global predicate, it is possible to add synchronization arrows (messages) to the computation such that the predicate always stays true.

4. Observing Consistent Global States

In general, a monitor process lies outside the system [1]. The monitor must assemble consistent global states at which it evaluates predicates. It does so from *timestamped state messages* which it receives from the cooperating group of processes. The state message carries a new value of the process's local variables and also an actual vector time-stamp of the sending process.

The monitor keeps incoming state messages in per-process queues (one separate queue for each process). The state messages in the queues are ordered by their sending order, which can be established immediately by examining the i^{th} component of the p_i's message vector time-stamp. Obviously, the monitor cannot order the states of different processes from their arrival order due to variable message latencies. Instead it has to examine the vector timestamps of the state messages.

Let $S = (s_1, s_2, \ldots, s_N)$ be a global state assembled by the monitor process from received state messages. Let $V(s_i)$ denote the vector time-stamp of the state s_i received from p_i. Then S is a consistent global state iff the following equation holds [1].

$$V(s_i)[i] \geq V(s_j)[i] \quad \forall i, j \in \{1, 2, \ldots, N\}$$

It says that the number of p_i's events known at p_j is not greater than the number of events that had occurred at p_i until it sent s_i. In other words, if a state of one process depends on the state of another process, then the global state includes also the state upon which it depends. This is a method which the monitor process can use to distinguish whether the state messages form a consistent or inconsistent state of the system.

5. Temporal Logics and CTL*

Temporal logic [13] is a kind of *modal logic*. It provides *temporal logic operators* for description of how the truthfulness of logic formulas (predicates) changes with time. There are many classes of temporal logics: propositional, first-order, global, compositional. They can be further divided on (according to the used model of time): linear, branching, continuous, and discrete.

The most often used temporal logic is CTL (*Computation Tree Logic* [14]) and its extension CTL* [15], which is a union of CTL and PLTP (Propositional Linear Temporal Logic).

While several temporal logics may be used for description of the system, the one we will use is CTL*, which has two types of formulas: *state*, and *path*. State formulas represent the properties of a specific state, whereas path formulas specify the properties of a specific path. CTL* formulas are composed of *path quantifiers*, *temporal operators*, and *atomic propositions*.

Path quantifiers are **A** (*all paths*) and **E** (*there exists a path*). The following five temporal operators are used in CTL*: **X** (*next time*), **F** (*future*), **G** (*global*), **U** (*until*), and **R** (*release*). A finite set of atomic propositions is consisted of CTL* formulas that represents some properties of a global state.

We use the following abbreviations in writing CTL* formulas:

- $\mathbf{AF}(p) \equiv \mathbf{A}[\text{true}\ \mathbf{U}\ p]$ intuitively means the same as *definitely*(p),
- $\mathbf{EF}(p) \equiv \mathbf{E}[\text{true}\ \mathbf{U}\ p]$ intuitively means the same as *possibly*(p),
- $\mathbf{EG}(p) \equiv \neg\mathbf{AF}(\neg p)$ intuitively means the same as *controllable*(p),
- $\mathbf{AG}(p) \equiv \neg\mathbf{EF}(\neg p)$ intuitively means the same as *invariant*(p).

5.1 Predicate Detection Algorithms

To our best knowledge, predicate detection algorithms have been designed for the above operators of CTL*, when p belongs to a specific predicate class. A review [18] of references to existing algorithms is summarized in Table 1.

Predicate	**Algorithm**			
p	**EF**(p)	**AF**(p)	**EG**(p)	**AG**(p)
conjunctive	[16]	[17]	[16]	[17]
disjunctive	[17]	[16]	[17]	[16]
stable	[2]	[10]	trivial	trivial
linear	[11]	open	[18]	[18]
obs.-indep.	[11, 10]	[11, 10]	[18]	[18]
regular	[19, 20]	open	[19, 20]	[19, 20]

Table 1: Predicate Detection Algorithms

6. On-line and Off-line Debugging

Generally, on-line and off-line approaches to debugging a distributed system are divided on *observation* and *control* [21].

In observation there are two processes responsible for observing a global state of the system. A *non-checker process* has direct access to local predicates and channels of a particular process, whereas the *checker process* (the monitor) is responsible for detection of predicates in a global state. In fact, this is called a *real-time monitoring.* When observing off-line, the entire computation is given to us, whereas in the on-line approach we are given only the past and we have to make observations as the computation is unfolding.

Control is the next natural step after observation. A *supervisory process* observes the underlying user process. Moreover, it controls it by delaying (or disabling) some events or changing the order of incoming or outgoing messages in the process. A supervisory process is performing on-line control if it does not know about the future of the computation. For example, it is impossible for an on-line controller to meet disjunctive specifications without avoiding deadlocks. In the off-line control model, the supervisor knows about the future. If some monitored computation had unexpected final results and all the messages sent during the computation were timestamped and logged, then the computation can be replayed under the supervisory control. During the computation replaying, the messages can be delayed or their order can be changed so that we got the expected results. This is called as *deterministic reply.* Note that delaying and changing the order of messages can also be applied onto an on-line computation. Due to the lack of knowledge about the future, we are unable to avoid deadlock. [22] shows that off-line predicate control for general boolean predicates is NP-hard.

Finally, [21] gives descriptions of problems in observation, which are open for research. For example, a problem of *detecting exactly-k predicates*, or a *detection of 2-SAT predicates.*

7. Conclusions

We have investigated main issues that make debugging of parallel and distributed systems so hard and presented a brief survey of a state of the art in the field of distributed systems debugging.

Global state predicates and their detection have been studied. They were expressed using operators of CTL* temporal logic. References to existing methods for detection of predicates of particular classes have also been presented.

In addition, we discussed on-line and off-line approaches to the debugging of distributed systems (particularly, observation and control of distributed computations).

Finally, we pointed out areas of future research as finding out efficient algorithms for regular and linear predicate detection under **AF** operator of CTL*, detection of exactly-k predicate, detection of 2-SAT predicates, and detection of conjunction of 2-local predicates.

References

[1] Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design, 3rd edition, 672 pages, Addison Wesley, ISBN 0-201-61918-0, 2000.

[2] Chandy, K. M., Lamport, L.: Distributed

Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems 3:1, 195, pp.63–75, 1985.

[3] Babaoglu, O., Marzullo, K.: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, Distributed Systems, Addison-Wesley, editor S. Mullender, pp.55–96, 1993.

[4] Netzer, R. H. B., Miller, B. P.: Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs, CS-94-32, http://citeseer.nj.nec.com/ netzer92optimal.html, 1994.

[5] Helary, J. M.: Observing Global States of Asynchronous Distributed Applications, In Proceedings of the 3rd International Workshop on Distributed Algorithms, number 392 in LNCS, pp.124–135, Springer, 1989.

[6] Mattern, F.: Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation, Journal of Parallel and Distributed Computing, volume 18:4, pp.423–424, 1993.

[7] Chow, R., Johnson T.: Distributed Operating Systems & Algorithms, 569 pages, Addison Wesley, ISBN 0-201-49838-3, 1997.

[8] Bhargava, B., Lian, S. R.: Independent Checkpointing and Concurrent Rollback for Recovery – An Optimistic Approach, In Proceedings of the International Conference on Data Engineering, pp.182–189, 1988.

[9] Wang, Y. M., Fuchs, W. K.: Lazy Checkpointing Coordination for Bounding Rollback Propagation, Symposium on Reliable Distributed Systems, pp.78–85, 1993.

[10] Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and Temporal Predicates in Distributed Systems, ACM Transactions on Programming Languages and Systems, 17(1):157–179, 1995.

[11] Chase, C., Garg, V. K.: Detection of Global Predicates: Techniques and Their Limitations, Distributed Computing, 11(4):191–201, 1998.

[12] Cooper, R., Marzullo, K.: Consistent Detection of Global Predicates, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices, 26(12):167–174, 1991.

[13] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer Verlag, 442 pages, ISBN-0387976647, 1991.

[14] Clarke, E. M., Emerson, E. A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, In Proc. of the Workshop on Logics of Programs, volume 131 of Lecture Notes in Computer Science, Yorktown Heights, 1981.

[15] Clarke, E. M., Emerson, E. A., Sistla, A. P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal-Logic Specifications, ACM Transactions on Programming and Systems, 8(2):244–263, 1986.

[16] Garg, V. K., Waldecker, B.: Detection of Weak Unstable Predicates in Distributed Programs, IEEE Trans. on Parallel and Distributed Systems, 5(3):299-307, 1994.

[17] Garg, V. K., Waldecker, B.: Detection of Strong Unstable Predicates in Distributed Programs, IEEE Trans. on Parallel and Distributed Systems, 7(12):1323-1333, 1996.

[18] Garg, V. K., Sen, A.: Detecting Temporal Logic Predicates on the Happened-Before Model, Technical Report TR-PDS-2001-003, PDSL, ECE Dept. Univ. of Texas at Austin, 2001.

[19] Garg, V. J., Mittal, N.: Computation Slicing: Techniques and Theory, In Proc. of the 15th Intl. Symposium on Distributed Computing (DISC), pp.78–92, 2001.

[20] Garg, V. J., Mittal, N.: On Slicing a Distributed Computation, In Proc. of the 15th Intl. Conference on Distributed Computing Systems (ICDCS), pp.322–329, 2001.

[21] Garg, V. K.: Observation and Control for Debugging Distributed Computations, Proc. 3rd Intl. Workshop on Automated Debugging (AADEBUG'97), Linkoping, Sweden, pp.1–12, 1997.

[22] Tarafdar, A., Garg, V. K.: Predicate Control for Active Debugging of Distributed Programs, Proc. of the IEEE 9th Symposium on Parallel and Distributed Processing (SPDP), pp.763–769, Orlando, USA, 1998.