A Race Detection Mechanism Embedded in a Conceptual Model for the Debugging of Message–Passing Distributed Programs*

Ana Paula Cláudio and João Duarte Cunha

[1]Faculdade de Ciências da Universidade de Lisboa
Departamento de Informática, Campo Grande, Edifício C5, Piso 1, 1700 LISBOA Portugal
apc@di.fc.ul.pt
[2]Laboratório Nacional de Engenharia Civil
Av. do Brasil, 101, 1799 LISBOA CODEX, Portugal
jdc@lnec.pt

Abstract. An object-oriented conceptual model for the debugging of message-passing distributed programs incorporates several debugging facilities: generation of a space-time diagram showing the progression of the execution being studied, detection of race conditions, detection of particular kinds of predicates and representation of causality cones. The focus of this paper is the race detection mechanism. The proposed mechanism comprises two steps: detection of pairs of receive events in the same process potentially involved in race conditions and verification of the legitimacy of the potential race condition. The mechanism relies on the analysis of two arguments, *process id* and *message tag*, in receive events and consume events, which are considered as distinct types of occurrences in the conceptual model.

1 Introduction

This paper describes a race detection mechanism embedded in a conceptual model for message-passing distributed programs. This model was conceived according to the object-oriented methodology and incorporates several debugging facilities.

Due to race conditions the internal work of a distributed program may be nondeterministic, that is, two successive executions of the program with the same input behave differently. Therefore, a bug observed in one execution may vanish in a later one and a mechanism for detecting these conditions is of utmost importance.

However, it should be stressed that the programmer may intentionally introduce race conditions in order to obtain improved performances. So, only the programmer is able to distinguish between intentional and non-intentional races.

The conceptual model is the core of MPVisualizer, a debugging tool for message-passing programs that is capable of operating either in reexecution mode or in *post-mortem* mode. Besides the race detection mechanism, the conceptual model incorporates the following debugging facilities which are available in the tool:

* This work has been partially funded by FCT(Portugal) under Project POSI/39351/SRI/2001 Mago2.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 57–65, 2003.

- Generation of a space-time diagram showing the progression of the execution being studied.
- Detection of local predicates as well as disjunction and conjunction of local predicates. Local predicates are defined by the user and, as explained in [1], a specific class must be rewritten in order to prepare the tool for the detection of each local predicate. Once the local predicates are defined no further programming is needed for the detection of disjunctive and conjunctive predicates.
- Representation of causality cones. The tool has the capability of adding to the space-time diagram a representation of the causality cone [2] or causal history [3] of an event previously selected by the user. The causality cone of an event contains all the events in the program, which can possibly have influenced the event. Conjunctive and disjunctive predicates are evaluated in the frontier of a causality cone. The frontier of a causality cone includes the last event belonging to the cone in each process.

Detection of disjunctive and conjunctive predicates as well as the representation of causality cones did not exist in the previous version of MPVisualizer [1]. Also, the race detection mechanism has undergone a major improvement in the current version.

In the next section a brief description of MPVisualizer is given. The race detection mechanism is thoroughly explained in section three. Section four is dedicated to the conclusions.

2 MPVisualizer

As mentioned above, the core of MPVisualizer is the conceptual model. Conceived according to the object-oriented methodology, this model comprises two groups of classes: kernel classes and graphical classes. Classes in the first group are independent from the message-passing software. Classes in the second group are subclasses of the classes in the first group and encapsulate all the programming details that depend on the graphical software.

MPVisualizer builds, either in reexecution or in *post-mortem* mode, a concrete model of the execution being studied. This model is an instance of the conceptual model and is called the functional model.

MPVisualizer has three components: the visualization engine, the reexecution mechanism and the graphical interface.

The visualization engine contains the functional model, the entity responsible for its management, called *manager*, and a set of processes with special tasks, the *collectors* and the *central process*. *Collectors*, one per machine, receive data from local processes in the execution being studied and forward them to the *central process*, which is in charge of sending to the manager all the information, needed to build the functional model. The code executed by *collectors* depends on the message-passing software used by the program being studied; everything else in the visualization engine is independent of it.

The reexecution mechanism is based on *Instant Replay* [4], a well known mechanism for shared memory communication that we have adapted for message-passing communication.

The reexecution mechanism includes two phases: trace and replay. In the trace phase, minimal information is stored to minimize the inevitable intrusion caused by

tracing activities. Although minimal, the stored information is sufficient to ensure that, during the replay phase, each process will consume the same messages, in the same order.

The monitoring code has been inserted in the standard libraries of the message-passing software. Program code is kept untouched but linking is different in the trace and replay phases. In the trace phase, the program code must be linked with one modified library, the trace library; for the replay phase, it must be linked with a second modified library, the replay library. The routines in this last library force the re-execution to respect the causal order of the communication events recorded during the original execution. They also force the processes to send blocks of information to the local *collector* process.

The graphical interface is composed of a menu bar and a drawing area where the space-time diagram is shown. Space-time diagrams are built in a bidimensional orthogonal referential: one of the axis corresponds to the continuous variable time and the other axis to the set of all the processes in the execution which is a discrete variable. Each process is represented by a line-segment parallel to the time axis and growing accordingly to the evolution of the process. Every kind of communication event has its own graphical symbol drawn over the line–segment of the corresponding process in order to signal its occurrence. An oblique line-segment connects each consume event to the corresponding send event. The diagram is consistent with the *happened-before* relation defined by Lamport [5].

By appropriate manipulation of the graphical symbols in the diagram, the user can profit from all the facilities offered by MPVisualizer.

3 Race Detection Mechanism

The race detection mechanism comprises two steps: detection of pairs of receive events in the same process potentially involved in race conditions and verification of the legitimacy of the potential race condition.

The original mechanism [1] included also the same two steps. However, the first step has been improved in order to enable the identification of a broader spectrum of race conditions.

The mechanism, embedded in the conceptual model, performs the detection of race conditions while the construction of the functional model is taking place.

The description of the mechanism is divided in three subsections: architecture, implementation and mechanism flaws.

3.1 Architecture

According to Netzer and Miller [6]: “Two messages race if either *could have* been accepted first by some receive, due to variations in message latencies or process scheduling”. Therefore, the entities involved in a race condition are: a couple of messages sent to the same process and at least one receive event that consumes the message arrived in the first place. Sometimes, there is a second receive event involved, which may or may not consume the message arrived in the second place.

The proposed mechanism relies on the analysis of two arguments, *process id* and *message tag*, in receive and consume events. In the conceptual model, receive and consume events are considered as distinct types of occurrences:

- A receive event occurs when a process intends to consume a message; it can be either blocking or non-blocking. A *process id* and a *message tag* are specified; both values are integers and one or both can have the value "–1" which means "any". We assume that this value is strictly reserved for this meaning.
- A consume event occurs when a process consumes a message. An event of this kind is always preceded by a receive event. It contains the tag of the message and its sender id. Both values are different from "-1".

A consume event is always preceded by a receive event but, it is possible that a receive event is not followed by a consume event. Therefore, for any pair of receive events, four distinct cases can be identified:

- RCRC: both receive events, either blocking or non-blocking, are followed by consume events.
- RCR-: the first receive event, either blocking or non-blocking, is followed by a consume event. The same does not happen with the second receive event which is not immediately followed by a consume event. If this receive event is a blocking one the execution of the process remains suspended.
- R-RC: the first receive event is not followed by a consume event while the second one is. This case is not possible when the first receive is blocking.
- R-R-: none of the two receive events is followed by a consume event. This case is not possible when the first receive is blocking. If the first receive event is non-blocking and the second receive event is blocking, the execution of the process is suspended.

Potential race conditions are considered as non-legitimate, *i.e.* spurious, when one of the following situations holds:

- Both messages have the same origin. Assuming that communication channels are FIFO, two messages sent by the same process to an equal destination can not be racing with one another.
- Consumption of the first message "happens before" the sending of the second one, in the sense introduced by Lamport [5]. These situations are called *false race condition* [6].

The race detection mechanism compares the pair of values (*process id*, *message tag*) of a receive event with the equivalent pair of a later receive event in the same process. This comparison deals with $16=2^4$ different combinations of values. Table 1 details all the sixteen different situations. For referencing purposes, each situation is identified in the first column by an integer ranging from 1 to 16.

The second and third columns correspond to the arguments *process id* and *message tag* of the first receive event; the fourth and fifth columns correspond to the same arguments of the second receive event.

The conceptual model assumes that every message carries its origin and its tag, thereafter called *message.pid* and *message.tag,* respectively. Every consume event contains the pair of values that corresponds to the consumed message. These values are useful in both detection steps; however they are essential for the second one, when the legitimacy of potential races is verified.

Table 1. Sixteen combinations of pairs (*process id*, *message tag*) in two receive events.

nº	recv1		recv2		Comments
1	pid1	tag1	pid2	tag2	There is no race condition
2	pid1	-1	pid2	tag2	There is no race condition
3	-1	tag1	pid2	tag2	There is a potential race condition if tag1=tag2
4	-1	-1	pid2	tag2	There is a potential race condition
5	pid1	tag1	pid2	-1	There is no race condition
6	pid1	-1	pid2	-1	There is no race condition
7	-1	tag1	pid2	-1	There is a potential race condition if both messages are consumed and both have a tag equal to tag1 or if the message consumed in the first place has a *process id* equal to pid2
8	-1	-1	pid2	-1	There is a potential race condition
9	pid1	tag1	-1	tag2	There is no race condition
10	pid1	-1	-1	tag2	There is no race condition
11	-1	tag1	-1	tag2	There is a potential race condition if tag1=tag2
12	-1	-1	-1	tag2	There is a potential race condition
13	pid1	tag1	-1	-1	There is no race condition
14	pid1	-1	-1	-1	There is no race condition
15	-1	tag1	-1	-1	There is a potential race condition if both messages have been consumed and if they have equal tags
16	-1	-1	-1	-1	There is a potential race condition

In the set of situations listed on table 1 four subsets can be identified:

- Subset 1: situations 1, 2, 5, 6, 9, 10, 13 and 14. In these situations there are no race conditions because the message consumed in the first place has to be the one sent by the process specified in the corresponding receive event parameter.
- Subset 2: situations 4, 8, 12 and 16. These situations correspond to potential race conditions. Since the first receive event does not specify either the *process id* of the origin or the tag of the expected message, the message consumed first is the one that arrived first. The second message may or may not be consumed by the destination process.
- Subset 3: situations 3 and 11. In these situations there are potential race conditions if message tags are the same in both receive events.
- Subset 4: situations 7 and 15. In these situations there are potential race conditions if both messages have been consumed and both verify *message.tag = tag1*. Additionally, in situation number 7, a potential race condition also occurs if the message consumed in the first place verifies *message.pid = pid2*.

3.2 Implementation

The following piece of pseudo-code explains the implementation of the first step in the mechanism. In the code, potential_race denotes a boolean variable, recv1.tag and cons1.pid correspond, respectively, to the arguments *process id* and *message tag* of the first receive event, ∃cons1 denotes a boolean expression that is true when the consumption of the first message has taken place; expressions recv2.tag, cons2.pid and ∃cons2 have similar meanings.

```
1  potential_race = false
2  if (recv1.pid = -1)
     { the condition is true in situations 3, 4, 7, 8, 11, 12, 15 and 16 }
3  then  if (recv1.tag = -1)
                { the condition is true in situations 4, 8, 12 and 16 }
4          then  potential_race = true
5          else  { recv1.tag≠ -1 }
6            if (recv1.tag = recv2.tag)  { the condition is true in 3 and 11 }
7            then  potential_race = true
8            else  { recv1.tag≠ recv2.tag }
9              if (recv2.tag = -1)  { the condition is true in 7 and 15 }
10             then   if (∃cons1)
11                      then  if (cons1.pid = recv2.pid)  { false in 15 }
12                              then  potential_race = true
13                              else { cons1.pid≠ recv2.pid }
14                                if (∃cons2)
15                                then   if (cons1.tag = cons2.tag)
16                                         then potential_race = true
```

Condition pid1 ≠ -1 is sufficient to guarantee that we are not dealing with a pair of receive events involved in a race condition. This condition excludes situations 1, 2, 5, 6, 9, 10, 13 and 14.

The condition on line 3 is evaluated for the remaining eight situations. It is true in situations 4, 8, 12 and 16, whose first receive events do not specify either the process id, or the tag of the expected message. These situations are signaled as potential race conditions, whether or not any of the corresponding consumptions have occurred.

The condition on line 6 is evaluated for situations 3, 7, 11 and 15. It is false for situations 7 and 15 and it can be true or false in situations 3 and 11. So, conditions 3 and 11 that satisfy the condition on line 6 are classified as potential race conditions.

Condition recv2.tag = -1, on line 9, is false for situations 3 and 11. This fact excludes these situations from the subsequent tests. The condition is true for situations 7 and 15.

Condition ∃cons1, on line 10, is true for situations 7 and 15 corresponding to cases RCR- or RCRC. Within this group, situations satisfying the condition cons1.pid=recv2.pid on line 11 are classified as potential race conditions. This condition is false for situation 15 and true for situations 7 similar to the one illustrated in figure 1, or when two messages with the same origin have been consumed. Both cases are marked as potential race conditions, but the last one is discarded during the verification process since it corresponds to a spurious race.

Condition ∃cons2, on line 14, is true for all RCRC cases in situation 15 and for situation 7 when condition cons1.pid = recv2.pid does not hold. In all these situations, there is a potential race condition when the consumed messages have the same tag, that is, when condition cons1.tag = cons2.tag, on line 15, holds.

A brief explanation of how this code has been incorporated in MPVisualizer follows. The mechanism is embedded in specific methods of class *Process*, the kernel class modeling any individual process in the execution. In situations 4, 8, 12 and 16, the detection of potential race conditions takes place when the first receive event is notified; in situations 3, 11 and some situations of type 7, the detection takes place

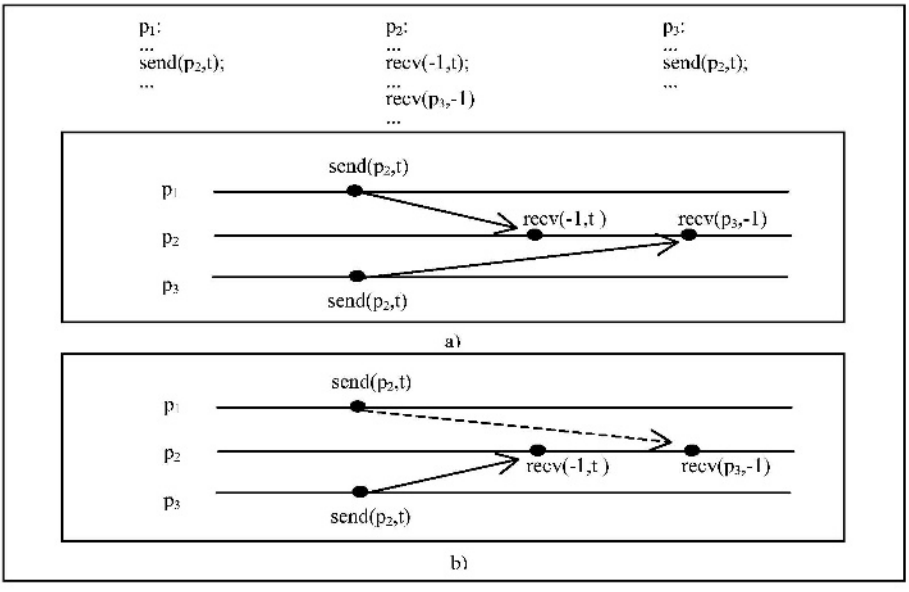

Fig. 1. Situation 7 of table 1. Dashed arrows correspond to messages that were not consumed.

when the second receive event is notified; in situations 15 as well as a different set of situations of type 7, the detection occurs when the second consume is notified. Therefore, detection code is embedded in methods b_recv, nb_recv and m_consumed, which are invoked respectively when a blocking receive, a non-blocking receive and a consume event occurs.

Whenever any of these methods detects a potential race condition, it invokes a private method of *Process* called notify_potential_race, which then verifies if the condition is a legitimate race.

Two receive events in the same process are classified as *adjacent* when no receive event occurred between them. In the current version of MPVisualizer, for efficiency reasons, detection of adjacent race conditions is performed automatically while detection of non-adjacent race conditions is performed only on demand, when the user selects the graphical symbol of a recv(-1,-1) event.

3.3 Mechanism Flaws

The mechanism is embedded in a conceptual model and therefore limited by the amount of information contained in the model. Since the model does not contain information about messages that arrived at a process but were not consumed, legitimization is not always feasible and it is possible to identify race conditions that are not detectable by the mechanism.

The mechanism is not able to legitimate potential races, namely when the second consumption does not occur, either because the receiver process did not execute the second receive event that could lead to the consumption of the late message, or because it did execute the second receive but the late message could not be consumed since its characteristics did not match the arguments of that receive event.

However, even when legitimization is not feasible, the tool signals the occurrence of a potential race condition.

Potential race conditions not detectable by the tool are those that involve messages that were not actually consumed. For instance in a situation of type 7, similar to the one illustrated in figure 1a) but without message consumptions because both receive events are non blocking and both messages arrived too late to be consumed.

4 Conclusions

A race detection mechanism is a valuable debugging facility. Nevertheless, it is relatively unusual to have this facility available in debugging tools. Out of 12 debugging tools compared with MPVisualizer, only 4 claim to detect race conditions.

The tools compared with MPVisualizer were: Atemp (the debugging tool of MAD) [7]; DDBG (a debugging tool previously used by GRADE) [8]; DETOP (the debugging tool of TOOL-SET) [9]; DIWIDE (the debugging tool of WINPAR and P-GRADE) [10]; p2d2 [11]; Panorama [12]; the debugging tool of Parascope [13]; PDBG (the debugging tool of DAMS) [14]; POET [15]; TotalView [16]; Xmdb [17] and XPVM [18].

Among these tools and environments only MAD, POET, XMDB, and the debugging tool of Parascope are able to detect race conditions.

The debugging tool of Parascope detects data races during execution and the compiler in the same environment is able to signal potential races.

MAD is able to detect race conditions based on the analysis of trace files produced by the trace/replay mechanism.

In POET the detection is performed by a component in charge of the trace/replay mechanism.

Both MAD and POET allow the user to test a different order of communication events and to conclude if the reordering of communication events causes different results. We consider that this is a useful feature to include in a future version of MPVisualizer, combined with the possibility of incremental reexecution.

Users of Xmdb can, *a priori,* list the race conditions classified as harmless; these are not signaled by the tool. MPVisualizer opens a pop-up window every time a race condition involving two adjacent receptions is detected. Depending on the program being studied, an overwhelming volume of information can be displayed. Therefore, future versions, will include filtering mechanisms to avoid this kind of effect.

The race detection mechanism described is based on an exhaustive study of all possible combinations of the arguments *process id* and *message tag* in two receive events in the same process. The mechanism is not perfect since it is possible to identify race conditions that are not detectable. However, this flaw is due to the fact that the mechanism is embedded in a conceptual model and therefore limited by the amount of information contained in the model. Without a thorough reformulation of the conceptual model itself this has to be considered as a feature.

References

1. Cláudio, A.P., Cunha, J.D. Carmo, M.B.*: Monitoring and Debugging Message Passing Applications with MPVisualizer.* Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society, pages 376-382, Rodhes, January 2000.
2. M. Raynal. *About Logical Clocks for Distributed Systems*. ACM Operating System Review, 26(1): 41-48, 1992.
3. C. Fidge. *Partial Orders for Parallel Debugging*. Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices: 24(1), pages183-194, 1989.
4. T. Leblanc, J. Mellor-Crummey. *Debugging Parallel Programs with Instant Replay.* IEEE Transactions on Computers, C-36(4), pages 471-482, April 1987.
5. L. Lamport. *Time, Clocks and the Ordering of Events in a Distributed System.* Communications of the ACM, 21(7): 558-565, July 1978.
6. R. Netzer, B. Miller. *Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs.* Proceedings of Supercomputing'92, Minneapolis, USA, pages 502-511, November 1992.
7. D. Kranzlmüller, R. Hügl, J. Volkert. *MAD - A Top Down Approach to Parallel Program Debugging*. Proceedings of HPCN'99, LNCS 1593, Springer-Verlag, pages 1207-1210, April 1999.
8. J.C. Cunha, J. Lourenço, T. Antão. *An Experiment in Tool Integration: The DDBG Parallel and Distributed Debugger*. Euromicro Journal of Systems Architecture, 45(11):897-907, Elsevier Science Press, 1999.
9. R. Wismuller et al.. *The Tool-Set Project: Towards an Integrated Tool Environments for Parallel Programming*. Proceedings of the 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, Koblenz, Germany, September 1997.
10. http://www.lpds.sztaki.hu.
11. http://science.nas.nasa.gov/Groups/Tools/Projects/P2D2.
12. J. May, F. Berman. *Designing a Parallel Debugger for Portability*. Proceedings of the International Parallel Processing Symposium, 1994.
13. K. Cooper et al.. *The Parascope Parallel Programming Environment*. Proceedings of the IEEE*:* 81(2), February 1993.
14. J.C. Cunha, J. Lourenço, J. Vieira, B. Moscão, D. Pereira*. A Framework to Support Parallel and Distributed Debugging*. Proceedings of HPCN'98, LNCS 1401, Springer-Verlag, pages 707-717, April 1998.
15. http://styx.uwaterloo.ca/poet/.
16. http://www.pallas.de/pages/totalv.htm.
17. http://www.lanl.gov/orgs/cic/cic8/para-dist-team/mdb/mdb.html.
18. http://www.netlib.org/utk/icl/xpvm/xpvm.html.