AL-AZHAR ENGINEERING SECOND
INTERNATIONAL CONFERENCE

December 21-24, 1991

PARALLEL REPLAY OF CONCURRENT PROGRAMS

M.M. Najjar* and T. Elrad**

*Assistant Prof., Info. and Computer Science Dept.
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

**Assistant Prof., Computer Science Dept.
Illinois Institute of Technology
Chicago, Illinois, U.S.A.

ABSTRACT

Re-executing a concurrent program with the same input may result in a different output. This is because concurrent programs are nondeterministic. Re-executing a concurrent program many times until a desired execution path is traversed is unlikely to produce execution replay because the probability of a program traversing the same execution path is small. Several replay methods have been designed for specific programming languages most of which use a centralized approach. We think that the centralized approach is unnatural and inefficient for the replay of concurrent programs. We present in this paper a parallel replay method for message-based concurrent programs.

We have designed a specification approach for program replay, named the locality principle. Locality uses the local paths of processes to specify an execution path. As a result, the course of execution for each process can be observed individually. This result is then used to design our parallel replay method. The advantages are: eliminating the exhaustive re-execution of concurrent programs, producing fewer testable execution paths, and providing parallel replay of concurrent processes. The method can be implemented for a number of programming languages like Ada and Concurrent C.

KEYWORDS

Concurrent programming; deterministic execution; execution replay; testing and debugging; parallel replay.

INTRODUCTION

Testing and debugging techniques for concurrent programs are much more difficult than for sequential programs. This is due to the multiple interacting threads of data and control and the nondeterminism inherent in concurrent programs [1].

A sequential program is usually tested by executing it once with a test input, the results are then compared with the expected results. In case of an error, the program is usually executed again, with the same test input, to collect some debugging information about the erroneous execution. The program is finally executed once again after it is corrected to check that no new errors were introduced.

This testing and debugging cycle is based on the fact that the replay of the erroneous execution of a sequential program is achieved by re-executing the program with the same test input. In testing and debugging concurrent programs the replay of an execution is a major problem. The testing and debugging cycle presented above can be applied to concurrent programs provided that an execution can be replayed.

In the next section we give an overview of concurrent program replay. Then we present our locality principle and its application to concurrent program replay. Based on this principle, we present a parallel deterministic replay method which can be applied to concurrent languages like Ada [2] and Concurrent C [3]. The Communicating Sequential Processes (CSP) notation [4] will be used throughout this paper.

An Overview of Concurrent Programs Replay

Re-executing a concurrent program with the same input may give different results. This is due to the use of nondeterministic constructs- available now in many high-level languages- and the nondeterminism inherent in the program execution environment. The replay problem [5] involves executing a concurrent program with a test input for the purpose of duplicating a previously exercised execution path. This problem is known as the execution replay problem, more on this problem and different approaches of handling it can be found in [6-14].

It is important to be able to replay an execution of a concurrent program because errors need to be reproduced for it to be corrected. An error in a concurrent program may not appear for many executions due to its dependency on some ordering of events which may be nondeterministic [15]. That is why once an error is detected, we need to reproduce it by replaying the execution of the program.

There are basically two sources for the nondeterministic behavior of a concurrent program: one is the use of explicit nondeterministic constructs in a concurrent program. This type is under the programmer control and exists locally within each concurrent process. The other type is the nondeterminism in the execution environment of a concurrent program. This type is outside the scope of the programmer and is

inherent in the execution environment. An example of the first, is the use of the Alternative construct in CSP or the Select statement in Ada. An example of the second, is the nondeterminism imposed by a multiprocessor scheduler on the set of ready-to-run processes in a concurrent program.

Thus, there are basically two aspects of the replay problem: to replay the actions of the execution environment, and to replay the actions of the program as imposed by its semantics. Duplicating the actions of an execution environment is hopeless in most cases. This is due to the massive number of nondeterministic constraints in parallel systems, most of which are outside the control of a program. However, duplicating the actions taken by a concurrent program is possible.

There are some difficulties associated with the replay of a concurrent program: first is the difficulty to recognize and record actions which affect the replay process and second, the difficulty to minimize the degree of the side effect introduced by the replay process.

Related Work

Exhaustive re-execution is the most obvious method of replay. In this method a concurrent program is executed many times with the same test input until the desired execution is replayed. This method does not guarantee error reproduction because the probability of a program exercising the same execution when supplied with the same input may be small. The massive number of possible execution paths makes this method inefficient and expensive.

Other more efficient replay methods have been suggested based on handling the problems which arise from the nondeterminism within concurrent programs [6-8]. Deterministic execution replay methods have been devised to eliminate the need for exhaustive re-execution by first planning an execution path or collecting, at run-time, some information about a previously exercised execution. Then, some additional statements are added to the original program to eliminate the inherent nondeterminism. Finally, the extended program is executed and the desired execution path is replayed. Deterministic methods extend the concurrent program by adding one or more controllers for replay. Usually one of these controllers is in charge of the full replay process. This technique imposes a sequential order on a concurrent program replay which is unnatural and inefficient.

Our Approach

We have made a distinction between two types of concurrent actions at the level of specifying a concurrent program's execution. Based on this distinction we developed a method to specify a program execution path called the locality principle. We have developed a parallel replay method based on this principle. The method uses one controller per process to help replay its communications with other processes where each controller may be executing in parallel. This technique preserves the concurrent structure of the program which may be altered by conventional replay approaches [8, 10]. The method has a significant

impact on the time necessary to replay a program by reducing the number of testable paths, and performing the replay in parallel.

LOCALITY

Specifying an execution path for a concurrent program is essential for its replay. In the space-time view presented in [16] an execution is viewed as a partially ordered set of events. The partial ordering is determined by these two rules: all actions are totally ordered within each process, and sending a message comes before its arrival. A later work by Fidge [9] uses partial order for parallel program debugging. We have adapted these approaches in our work.

With respect to program replay, actions are generally classified into two categories: sequential and concurrent. Obviously, the replay of sequential actions is trivial. However, concurrent actions are the source of trouble in the replay process. We have distinguished two types of concurrent actions: preferred and communication actions. A preferred action is an action that has been selected within a nondeterministic construct over other actions. A communication action is either a send or a receive.

A process can be specified in terms of these two types of concurrent actions. Thus, we specify a concurrent program's execution as a number of totally ordered sequences. Each sequence is associated with a process and consists of a totally ordered sequence of preferred and communication actions. The sequence of these actions within each process allows us to observe the course of execution for each process individually. More specifically, we define a process's local path (LP) as the totally ordered sequence of all preferred and communication actions within that process with respect to a specific input. In terms of this local path, another definition is also essential: a program's local history (LH) is defined as the set of all local paths within that concurrent program with respect to a specific input. Thus, the number of local paths within a program's local history is equal to the number of processes in that program.

Using this representation for replay, the initial execution of a concurrent program with a specific input produces a local path for each of its processes which form its local history for that specific execution. Notice that a different input may result in a different local history.

A set of global paths can be constructed from a local history; for this purpose we define a program's global path (GP) with respect to a specific input as any possible merge [17] of actions applied to all actions within a program's local history. The distinction between local histories and global paths is very important for the replay process. The main difference is that a program's local history is a collection of local path sequences, where as a global path is any merge sequence of all the actions present in all local paths. It is possible to get more than one possible merge of the same local history, therefore, a number of global paths can be associated with one local history. We define two global paths as equivalent if they define the exact local history with respect to the same input. More on the relationship between global paths and local histories, will be presented later in the paper.

Locality in this context means using local histories to represent a program's execution path; it's the bases for our parallel replay method. There are many advantages for the locality principle which we will explore in the rest of this paper.

Locality for the Parallel Replay of Concurrent Programs

We redefine the replay of an execution path according to the locality principle as: the replay of a global execution path for a concurrent program P with respect to an input I is successful if the re-execution of P with I produces the exact or an equivalent global execution path.

The replay of a global path can further be defined in terms of its local history as: the replay of a global path is successful if the replay of its corresponding local history is successful. For example, let us assume that we are willing to replay the global path G1 which has the corresponding local history LH1. Replaying G1 is equivalent to the replay of LH1.

The replay of a local history is now defined in terms of the local paths it contains as: a replay of a program's local history is successful if the replay of each one of its local paths is successful. Obviously, replaying the local history can only be achieved by the replay of each local path. The only condition which is required in this case is that the execution path must be for a program that has terminated successfully. For example, the replay of LH1 above is equivalent to the replay of all local paths in LH1.

We can conclude from the above discussion the following result: the replay of a global path is successful if the replay produces its corresponding local history. This result has a significant impact on the replay process. If many global paths can correspond to the same local history then the replay of this local history is equivalent to the replay of all the corresponding global paths. More specifically, suppose G_1, G_2, ..., G_n are global paths corresponding to Local_History_1. Then the replay of Local_History_1 is equivalent to the replay of G_1, G_2, ..., G_n.

The relationship between local histories and global paths is explained through an example which uses the CSP notations.

Suppose that there are six processes in a CSP program C, where C uses the distributed termination convention and a SKIP statement has no effect.

```
C :: P1 || P2 || P3 || P4 || P5 || P6
P1 :: P3 ! 11
P2 :: P3 ? A
P3 :: *[   P1 ? A  ---> SKIP
       []  P2 ! 22 ---> SKIP
      ]
P4 :: P6 ! 44
P5 :: P6 ! 55
P6 :: *[   P4 ? B  ---> SKIP
       []  P5 ? B  ---> SKIP
      ]
```

In the above program, P3 communicates with P1 and P2, and the order of communications is nondeterministic (similarly for P4, P5, and P6). A list of all the possible local paths for each process is given below, where each local path contains one or more communication tuples. A communication tuple is represented by three entities: the name of the process, the type of command (Input or Output), and the communication parameter.

```
LP1  : ( (P1,O,11) )
LP2  : ( (P2,I,A) )
LP3a : ( (P3,I,A), (P3,O,22) )     -- P3 has two possible local paths
LP3b : ( (P3,O,22), (P3,I,A) )
LP4  : ( (P4,O,44) )
LP5  : ( (P5,O,55) )
LP6a : ( (P6,I,44), (P6,I,55) )    -- P6 has two possible local paths
LP6b : ( (P6,I,55), (P6,I,44) )
```

There are two possible nondeterministic actions in P3 and P6. According to our definition, this means that there are two possible local paths for P3 (LP3a and LP3b) and for P6 (LP6a and LP6b). Using the above list of local paths, a list of all possible local histories can be constructed as follows:

```
Local_History_1 : ( LP1, LP2, LP3a, LP4, LP5, LP6a)
Local_History_2 : ( LP1, LP2, LP3b, LP4, LP5, LP6a)
Local_History_3 : ( LP1, LP2, LP3a, LP4, LP5, LP6b)
Local_History_4 : ( LP1, LP2, LP3b, LP4, LP5, LP6b)
```

Note that LP1, LP2, LP4, and LP5 are unchanged in all local histories and the only change between two local histories is in the local paths of P3 and P6. A set of global paths is also associated with each one of the above local histories. Global_Path_1 and Global_Path_2 correspond to Local_History_1, on the other hand, Global_Path_3 correspond to Local_History_2.

```
Global_Path_1 : ((P1,O,11),(P4,O,44),(P2,I,A),(P5,O,55),
                 (P3,I,A),(P6,I,44),(P6,I,55),(P3,O,22))

Global_Path_2 : ((P1,O,11),(P2,I,A),(P4,O,44),(P5,O,55),
                 (P3,I,A),(P6,I,44),(P6,I,55),(P3,O,22))

Global_Path_3 : ((P1,O,11),(P4,O,44),(P2,I,A),(P5,O,55),
                 (P3,O,22),(P6,I,44),(P6,I,55),(P3,I,A))
```

The two global paths Global_Path_1 and Global_Path_2 are semantically equivalent, since both terminate with A=22, B=55. Local_History_1 represents the two global executions Global_Path_1 and Global_Path_2. In this case, Local_History_1 is said to be compatible with both Global_Path_1 and Global_Path_2. In general, one local history may be compatible with more than one global path, but all such global paths are semantically equivalent.

A global path and a local history are said to be compatible if and only if the global path preserves the order of actions within each local path of the

local history. Two actions are said to be independent if any total ordering of them preserves the same local history.

If a global path and a local history are compatible, then they are semantically equivalent. Obviously, changing the order of independent actions preserves the semantics of a program.

The above discussion suggests that all visible global concurrent program paths can be partitioned into their corresponding local histories, with the result that all global paths within one group are semantically equivalent. This result is important, because it reduces the number of testable global paths to the number of a local histories which belong to the program. If two global paths define the same local history, then it is enough to replay their local history once, because the two are semantically equivalent.

Another advantage of the distinction between global paths and local paths is that a program has many fewer local histories than global paths. Thus, significantly fewer test cases depend on local histories than global paths. Note that no global test cases are lost, but rather those that are semantically equivalent are eliminated. The reduction of the number of test cases may have a significant impact on the time needed to test a concurrent program.

Parallel Replay of Concurrent Programs

The first step in replaying a concurrent program's path is to determine its local history. The next step is to replay each local path within that local history.

The replay of a global path encourages the use of a centralized controller because a sequential list of actions is replayed. The controller stores a list of all global actions and controls communication actions among processes. Preferred actions are stored and replayed locally within each process. In this method, the case where two or more communication actions start simultaneously is excluded because only one communication action is permitted to start at any one time. On the other hand, if all local paths are to be replayed in parallel, then simultaneous replay of preferred and communication actions is not excluded.

The Parallel replay method presented here uses a set of controllers, and each controller is associated with a process. Each controller helps a process replay its communication actions. The controller simulates the communication environment of a process and makes sure that the sequence of calls coming to a process is in the predetermined order specified by the local path; preferred actions are replayed by the process itself.

This parallel replay method can be implemented using a two-phase approach similar to the one presented in [6] as shown in Fig. 1. The two phases are: a local path construction phase (CON), and a local path replay phase (REP). The purpose of the CON phase is to construct a local path for each process in a concurrent program. Constructing a local path is based on recording preferred and communication actions while a process is executing. The output of the CON phase plus the input to the

concurrent program becomes the input to the REP phase.

In the REP phase there are two major steps: the preparation of each process for its re-execution (extension step), and the actual re-execution of the program (replay step). The extension step transforms a nondeterministic program into a deterministic one according to its local history. This enables a debugger to re-execute an extended program as many times as required. The re-execution is done at the replay step, where results are also compared with the expected results of the program.

Processes follow the sequence of actions in their local paths by means of allowing only those actions at the beginning of their local paths to be selected at any one time. If an action at the beginning of a process's local path is not possible at some point of the execution, the process blocks itself until the action occurs. Specific details of how actions (communication and preferred) can be blocked depends on the language used.

The application of locality principle for program replay is useful because it partitions the replay problem into the replay of individual processes. One has to consider the concurrent characteristics of a programming language before applying this parallel method. Three main characteristics are identified: symmetric versus asymmetric naming conventions, synchronous versus asynchronous communications, and the control over nondeterminism within a process. We are currently working on applying this method on Ada and Concurrent C and a number of other concurrent languages.

CONCLUSIONS

In this paper we have presented a method for the use of local processes' paths as the method of specifying a concurrent program's execution path for the purpose of replay. Execution paths were represented in terms of their local histories and local paths. Based on this, a parallel replay method was presented which uses one controller per process for program replay. Controllers help processes to replay their communication actions. This eliminates the need for a centralized controller which can be a bottleneck to the replay process. The advantages are: reducing the number of testable execution paths, eliminating the need for exhaustive re-execution , making the replay process much easier to understand and to implement, and performing the replay in parallel. We think that this method is more efficient and more natural for the replay of concurrent programs. The implementation of this method on programming languages that has concurrent facilities like Ada and Concurrent C may differ slightly because of the difference in the concurrent facilities provided by these two languages.

ACKNOWLEDGEMENT

The first author acknowledges the support of King Fahd University of Petroleum and Minerals in presenting this paper.

REFERENCES

1- FRANCEZ, F., and HOARE, C.A.R. et al. (1979). Semantics of Nondeterminism, Concurrency, and Communication. Journal of Computer and System Sciences, 19: 290-308.

2- U.S.Dep. of Defence, (1983). The Ada Programming Language Reference Manual, ANSI/MILSTD 1815A Document, US Government Printing Office.

3- GEHANI, N.H., and ROOME, W.D., (1986). Concurrent C. Software-Practice and Experience, Vol. 16(9): 821-844.

4- HOARE, C.A.R. (1978). Communicating Sequential Processes. Communications of the ACM, Vol. 21(8): 666-677.

5- HANSEN, P., (1978). Reproducible Testing of Monitors. Software-Practice and Experience, Vol. 8: 721-729.

6- NAJJAR, M.M., and ELRAD, T. (1989). A Two-Phase Reproduction Method for Ada Tasking Programs. Proc. of the 7th Annual National Conference on Ada Technology. 197-206.

7- TAI, K-C., CARVER, R.H., and OBAID, E.E., (1991). Debugging Concurrent Ada Programs by Deterministic execution. IEEE Transactions on Software Engineering, Vol. 17(1): 45-63.

8- CARVER, R.H., and TAI, K-C., (1991). Replay and Testing for Concurrent Programs. IEEE Software, Vol 8(2): 66-74.

9- FIDGE, C., (1989). Partial Orders for Parallel Debugging. Proceedings of the Workshop on Parallel and Distributed Debugging, SIGPLAN Notices, Vol. 24(1): 183-194.

10- LEBLANC, T.J., and MELLOR-CRUMMEY, J.M. (1987). Debugging Parallel Programs with Instant Replay. IEEE Trans. on Computers, Vol. C-36(4): 471-482.

11- FIDGE, C., (1987). Reproducible Testing in CSP. The Australian Computer Journal, Vol. 19(2): 92-98.

12- PAN, D.Z., and LINTON, M.A. (1989). Supporting Reverse Execution of Parallel Programs. Proceedings of the Workshop on Parallel and Distributed Debugging. SIGPLAN Notices, Vol. 24(1): 124-129.

13- GOLDSZMIDT, G.S., and YEMINI, S. (1990). High-level Language Debugging for Concurrent Programs. ACM Transactions on Computer Systems, Vol. 8(4): 311-336.

14- GORDON, A.J., and FINKEL, R.A. (1988). Handling Timing Errors in Distributed Programs. IEEE Transactions on Software Engineering, Vol. 14(10): 1525-1535.

15- LAMPORT, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. Comm. of the ACM, Vol. 21(7): 558-565.

16- ALFORD, M.W., ANSART, J.P. et al. (1985). Distributed Systems: Methods and Tools for Specification. An Advanced Course, Lecture Notes in Computer Science, No. 190,Springer- Verlag, Berlin, Heidelberg.

17- PNUELI, A., (1981). The Temporal Semantics of Concurrent Programs. Theoretical Computer Science, Vol 13: 45-60.

NOMENCLATURE

LP : local path
LH : local history
GP : global path
CON : construction phase
REP : replay phase

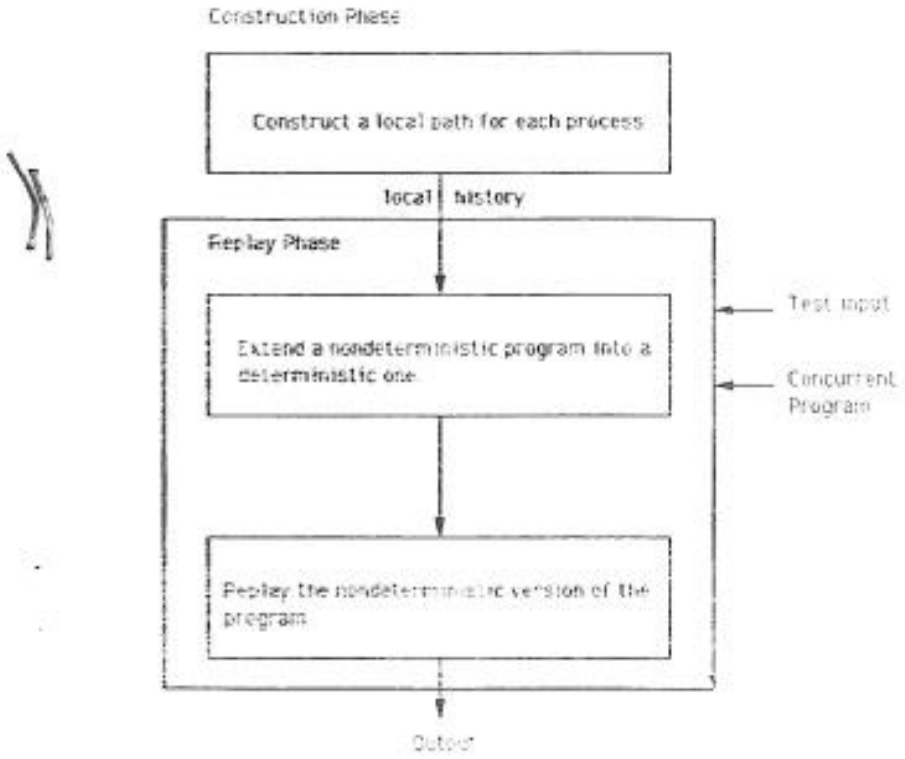

Fig. 1 Implementation of the Parallel Replay Method