Timing of Concurrent Events in Distributed Systems

A Framework for Reliably Testing A Computation for Data Dependencies

Karim Serhan
Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas
karim.serhan@utexas.edu

Nishanth Shanmugham
Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas
nishanths@utexas.edu

***Abstract*—Since concurrent events in a distributed computation may have data dependencies, out-of-order accesses of the shared data could lead to incorrect program execution. This paper outlines an algorithm that enables a developer to test all the possible execution orders that trigger these data dependencies by adding synchronization messages between events that could have dependencies.**

***Keywords*—software testing; distributed computing; controlled re-execution; dynamic programming**

I. Introduction

Distributed systems are inherently difficult to debug and test for several reasons, among which is the fact that in a true distributed system, only a partial order can be determined between events [1]. Concurrent events occurring on separate processes could access data in different orders in each execution of the program, possibly leading to different or unexpected outcomes. Thus, bug observation, and more so, bug reproducibility, is hard to achieve.

This paper proposes an offline algorithm that takes the trace of a distributed computation as input, and generates a list of traces with synchronization messages added between concurrent events, thus establishing a strict ordering on these concurrent events. If each of the output traces with strictly ordered events passes a target test suite, a developer can confidently conclude that the original trace will also pass the test suite irrespective of the runtime ordering of the concurrent events.

A developer can input their offline distributed execution trace, along with the specific data dependencies they are interested in testing. Our algorithm will generate computations corresponding to all the possible orders that satisfy these dependencies. We also supplement our platform with a GUI tool to visualize the results, and random trace generator to facilitate the testing of our algorithm for correctness and evaluation purposes.

The next section provides background information on distributed computations and introduces the terminology used in the paper. Section III explains the algorithm that generates the output computations with the synchronization messages added. Section IV explains the implementation details of our tool, including the GUI tool we used to generate the computations and verify our results, along with some examples. It also discusses performance results, detailing the time taken to execute our algorithm depending on the input computation characteristics.

II. Backgroud

A. Model of a distributed computation

Our model of a distributed computation is identical to that in [2]. It is based on message passing, without any assumptions on the upper bound of message delays. A distributed program consists of N processes denoted by $\{P_0, P_1, \ldots, P_{N-1}\}$, and a set of unidirectional channels, each of which connects two processes together. Note that a bidirectional channel can be modeled as two unidirectional channels. The channels are assumed to have reliable delivery, and an arbitrary but finite delay. Each process is defined as a sequence of events that transition the process from one state to another. Examples of events include the beginning of the execution of a function, the end of its execution, the sending of a message, and the reception of a message [2]. We also assume that events on the same process execute sequentially, that is, each process is single-threaded.

As argued in [1], a true distributed system can only have a partial order, dictated by the messages exchanged in the system. This order is called the *happens before* relation.

In the simplest case, if we consider two events occurring on the same process, then the one that occurs earlier is said to happen before the other. Note that this is well-defined since we assume processes are single-threaded. If we now consider two events occurring on different processes, where one event is the send event of a message, and the other is the receive event of the same message, then the first naturally happens before the second. The definition of *happens before* is transitively extended based on these two base cases. More formally, let $e \prec_{im} f$ denote that e is the event immediately preceding f on the same process, and let $e \leadsto f$ denote that e is the send event of a message and f is the receive event of that same message, then the *happens before* relation, denoted by $\rightarrow$, can be defined as:

1. $e \prec_{im} f \Rightarrow e \rightarrow f$
2. $e \leadsto f \Rightarrow e \rightarrow f$

3. $e \rightarrow f \text{ and } f \rightarrow g \Rightarrow e \rightarrow g$

As already mentioned and as discussed in [2], *happens before* can only ensure a partial order on the set of events in a computation. For two distinct events e and f, it may neither be true that $e \rightarrow f$ nor that $f \rightarrow e$. In this case, the events are said to be concurrent, and denoted by $e||f$. For two concurrent events e and f, e might execute at an earlier time than f in one execution of the computation, while in another independent execution, f might execute earlier.

B. *Data dependencies*

In this section, we formally define the data dependencies we are interested in. A data dependency is a situation in which a statement refers to data of an earlier statement [3].

a. Flow dependency, also known as true dependency or read-after-write (RAW) occurs when a statement reads from a variable that an earlier statement writes to. If the order of execution of these statements change, the reading statement might read the stale value of that variable [3].

b. Anti-dependency, also known as write-after-read (WAR), occurs when a statement writes to a variable that an earlier statement reads from. The concern here is, if the statements do not execute in this order, the reading statement might read the updated version of the variable [3].

c. Output dependency, also known as write-after-write (WAW), occurs when two statements write to the same variable. If the order of execution of the two statements change, the effective value of the variable that is persisted to memory might be different [3].

III. Algorithm

Our algorithm takes a distributed computation as input, annotated with the variables that are being accessed at each event. That is, each event is annotated with a list of the variables that have been read right before this event, and a list of variables that have been written right before this event. By right before, we mean that the variable has been read (resp. written) after the immediately preceding event and before the event in question (for the initial event, we assume that the variable has been read or written after the start of the execution and before the event).

The output of our algorithm is a list of computations, each of which identical to the input computation, but with some added synchronization messages to satisfy a certain ordering between the events that access the same variables.

To illustrate our algorithm, let us first consider the very basic example in Figure 1 (a), where we have only two processes and one event on each process. Suppose both of these events write on the same shared variable, say x_0. Note that both of these events are concurrent, meaning that in some executions e_1 might execute earlier than e_2, while in others, e_2 might execute earlier. In either of these two cases, there will be an output dependence, since the latter event will be writing to the same variable than the earlier event wrote to.

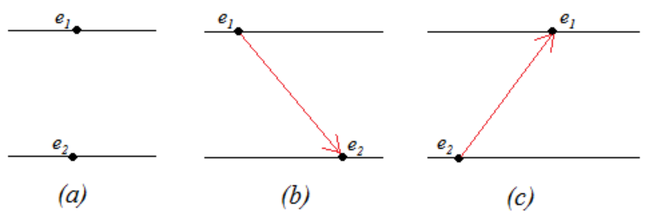


Figure 1: (a) Original computation, (b) first controlled computation, (c) second controlled computation

If the developer is interested in exploring output dependencies, then our output should be a list of computations that explore all the possible ordering. For this example, since there are only two possible orderings, we will output two computations, one of which has an added synchronization message from e_1 to e_2 (Figure 1 (b)), and the other has an added synchronization message from e_2 to e_1 (Figure 1 (c)).

The above describes a simplified version of the base case of our algorithm. Below we describe the full algorithm.

```
generateSynchronizedComputations(computation):
  concurrentEvents = getConcurrentEvents(computation)
  result = { computation }
  for each event1, eventsConcurrentTo1:
    generated = synchronize(event1, eventsConcurrentTo1, 0)
    result = mergeLists(result, generated)
  end for
```

```
synchronize(computation, event1, eventsConcurrentTo1, index):
   if index = size(eventsConcurrentTo1):
     return { computation }
   end if
   event2 = eventsConcurrentTo1[index]
   generated = {}
   message1Added = false
   message2Added = false
   if checkDependenciesSatisfied(event1 $\rightsquigarrow$ event2):
     newComputation, message1Added =
        addMessage(computation, event1 $\rightsquigarrow$ event2)
     if message1Added:
        generated = generated ∪ synchronize(newComputation,
            event1, eventsConcurrentTo1, index + 1)
     end if
   end if
   if checkDependenciesSatisfied(event2 $\rightsquigarrow$ event1):
     newComputation, message2Added =
        addMessage(computation, event2 $\rightsquigarrow$ event1)
     if message2Added:
        generated = generated ∪ synchronize(newComputation,
                event1, eventsConcurrentTo1, index + 1)
     end if
```

```
    end if
    if not(message1Added) and not(message2Added):
        generated = generated ∪ synchronize(computation,
                event1, eventsConcurrentTo1, index + 1)
    end if
    return generated
```

In the remainder of this section, we explain the specifics of this algorithm in more detail.

A. *generateConcurrentEvents(computation)*

The *generateConcurrentEvents* function returns a mapping between each event and the list of all events it is concurrent with. For performance reasons, if e_2 was included in the set of concurrent events for e_1, we don't include e_1 in the set of concurrent events for e_2, so as to not process the same pair of concurrent events twice.

To compute the list of events concurrent to an event e, we first find all the events that are reachable from e, subtract it from the set of all events in the computation, and then filter out all the events from which e is reachable. We obviously also need to remove e itself, as e is not considered concurrent with itself. Following this procedure, we will be left with all the events f such that $e \neq f, e \nrightarrow f$, and $f \nrightarrow e$, which is the definition of all events concurrent with e. In this part of the algorithm, we use dynamic programming, in order to find the set of events reachable from a given event.

B. *addMessage(computation, e1 $\rightsquigarrow$ e2)*

The *addMessage* function creates a copy of the computation, possibly with the added message. As will be explained in the following paragraph, it is not always possible to add a message. This function returns that modified copy and a Boolean indicating whether the message was added successfully or not. Note that in terms of the pseudo-code above, *addMessage* does not modify the computation passed to it.

A message will not be added if it generates a cycle in the computation, as this would violate the asymmetric nature of the happens before relation. Note that even though we are only looking at pairs of concurrent events in terms of the original computation, and an added message between two concurrent events should not result in a cycle, the function *addMessage* might not be acting on the original computation. For example, if e_1 and e_2 are concurrent, and if we have already added a message from e_1 to e_2 into a copy of the computation, then in the recursive call, a message from e_2 to e_1 could not be added to this copy, as this would create a cycle in the copy of the computation (even though a message form e_2 to e_1 could be added to the original computation).

C. *checkDependenciesSatisfied(e1 $\rightsquigarrow$ e2)*

This function checks whether a synchronization message from e_1 to e_2 is actually needed, based on what variables are being read an written in each of these events. For example, if e_1 reads x, and e_2 write on x, and if the developer is only interested in flow dependencies, then we do not have to explore the order of events where e_1 happens before e_2, since this order does not result in a flow dependence. However, if the developer is also interested in anti-dependencies, then a message from e_1 to e_2 should be added, as the resulting order would cause e_1 to read from x before e_2 writes to it.

D. *mergeLists(list1, list2)*

This function takes two lists of computations, and generates a list where each computation in *list1* is merged with every computation in *list2*. In other words, the size of the resulting list will be up to $|list1| * |list2|$. Note that the size might be actually smaller because some computations could be impossible to merge (if the merging would result in a cycle). The pseudocode for this function is given below.

```
mergeLists(list1, list2):
  result = {}
  for each comp1 in list1
    for each comp2 in list2
      mergedComp = merge(comp1, comp2)
      if mergedComp is not NULL
        result = result U { mergeComp }
      end if
    end for
  end for
  return result
```

E. *merge(computation1, computation2)*

We finally describe the function that merges two computations mentioned above. In this function, the two computations are exactly the same in terms of processes, events, and ordinary messages, but they might differ in the synchronization messages that they have. One might initially consider simply merging the synchronization messages of each of these two computations, and ignoring the ones that could not be added due to a cycle. While this is correct, the resulting computation created will often have redundant synchronization messages that are undesirable.

Consider the two controlled computations to be merged in Figure 2 (a) and (b) below, where the red arrows denote the synchronization messages. The red variable names indicate the variables being written, and the blue variable names indicate variables being read. All events are concurrent in the original computation.

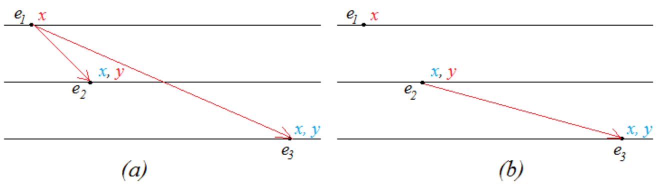


Figure 2: Synchronization messages added (a) when considering e_1 and its concurrent events, (b) when considering e_2 and its concurrent events

Suppose the developer is interested in exploring flow dependencies. When considering e_1 and its concurrent events, a synchronization message will be added from e_1 to e_2, and one from e_1 to e_3 (Figure 2 (a)). When considering e_2 and its concurrent events, a synchronization message will be added from e_2 to e_3 (Figure 2 (b)).

If we were to follow the simple approach of merging the message, we would obtain the computation in Figure 4 (a) below.

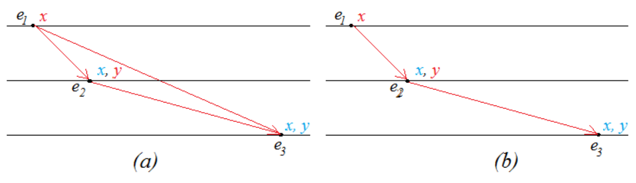


Figure 4: (a) sufficient but unnecessary merging, (b) sufficient and necessary merging

While this is valid, the synchronization message from e_1 to e_3 is not needed because the relation $e_1 \rightarrow e_3$ is already satisfied from the other two messages.

The following merge algorithm avoids this problem.

```
merge(computation1, computation2):
    output = copy either computation1 or computation2 without
             any synchronization messages
    for each synchronized message $e_1 \rightsquigarrow e_2$ in computation1:
        if not (isReachable(computation2, $e_1$, $e_2$)):
            add synch message from $e_1$ to $e_2$ to result
        end if
    end for
    for each synchronized message $e_1 \rightsquigarrow e_2$ in computation2:
        if not (isReachable(computation2, $e_1$, $e_2$)) or
            computation2.containsMessage($e_1 \rightsquigarrow e_2$):
            add synch message from $e_1$ to $e_2$ to result
        end if
    end for
    return output
```

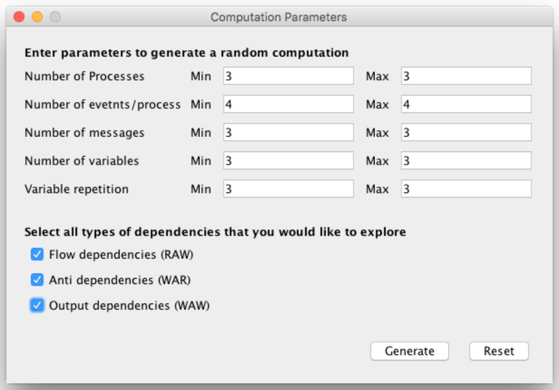
This approach will add all the synchronization messages from computation1 into result as long as the corresponding happens before relation is not already satisfied in computation2, and then do the same thing for the synchronization messages in comptation2. Note the additional call to computation2.*containsMessage*($e_1 \rightsquigarrow e_2$), which is needed in order to not skip adding a synchronization message if it is present in both computaiton1 and computation2. The resulting computation is the one in Figure 4 (b).

IV. Results

The algorithm is available as Java package to which input traces can be provided and a list of output traces obtained. In addition, the accompanying GUI application can be used to visualize computations and generated random traces according to some parameters.

A. *GUI application*

As input to the GUI application, the computation parameters and the types of dependencies to be explored are specified. The example configuration is Figure 5, a 3-process system with 4 events in each process is chosen. There are 3 messages and 3 variables in total in the system, with each of the variables being accessed exactly 3 times across all events. Synchronization messages in the output will be added for all 3 types of dependencies, as necessary.



Computation Parameters

Enter parameters to generate a random computation

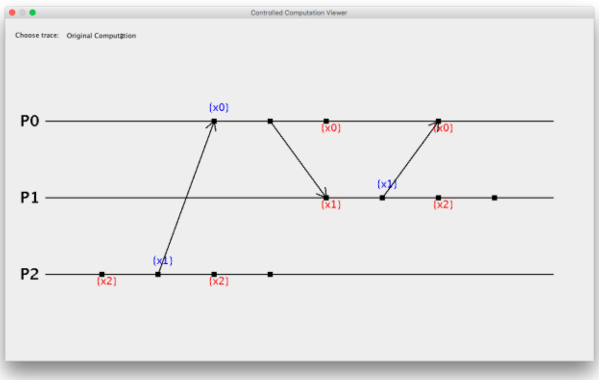
Number of Processes	Min	3	Max	3
Number of evetnts/process	Min	4	Max	4
Number of messages	Min	3	Max	3
Number of variables	Min	3	Max	3
Variable repetition	Min	3	Max	3

Select all types of dependencies that you would like to explore

- ✓ Flow dependencies (RAW)
- ✓ Anti dependencies (WAR)
- ✓ Output dependencies (WAW)

Figure 5: Configuration parameters to generate traces in the GUI application

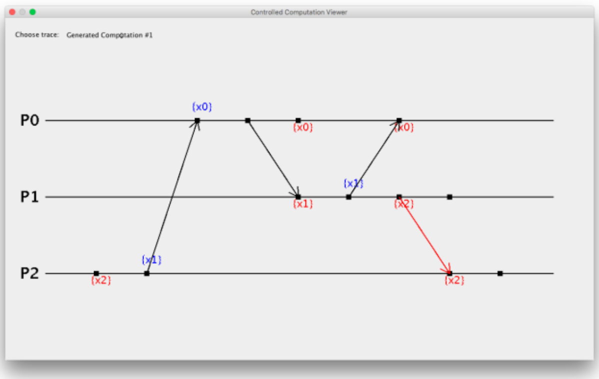
The original 3-process computation generated is shown in Figure 6. Each event in the diagram has a list of variables read at the event (in blue) and written at the event (in red) displayed next to it. The original messages between processes are represented as black arrow with the arrowhead pointing in the direction of the message.



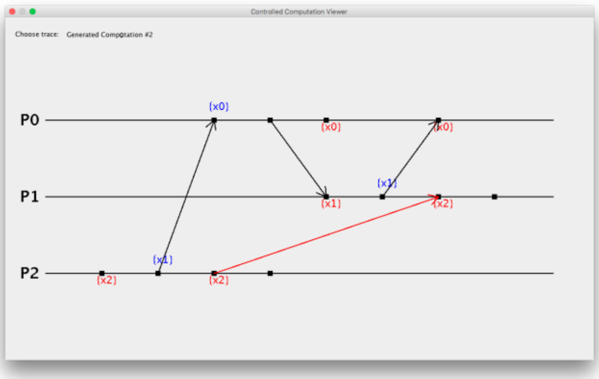
Controlled Computation Viewer
Choose trace: Original Computation
P0
{x0}
{x0}
{x0}
P1
{x1}
{x1}
{x2}
P2
{x2}
{x1}
{x2}

Figure 6: The original computation 3-process, 4-event, 3-message computation

The generated computations can be viewed using a dropdown menu. The first generated computation, shown in Figure 7 (a), adds a synchronization message (shown in red) from the third event on the second process to the third event on the third process, thus establishing a *happens before* relationship between these originally concurrent events. In effect, this enforces an order on the output dependency of the *x2* variable. Besides this message, no other messages are required in this example because the original computation enforces the necessary ordering on the access of other variables. Figure 7 (b) shows the second possible generated computation. It adds a single message (shown in red) between the same events, but in the opposite direction. The effect is similar: an order on the output dependency of the *x2* variable is created.


Controlled Computation Viewer
Choose trace: Generated Computation #1
P0
P1
P2
{x0}
{x0}
{x0}
{x1}
{x1}
{x1}
{x2}
{x2}
{x2}

(a)


Controlled Computation Viewer
Choose trace: Generated Computation #2
P0
P1
P2
{x0}
{x0}
{x0}
{x1}
{x1}
{x1}
{x2}
{x2}
{x2}

(b)

Figure 7: (a) First generated computation, (b) Second generated computation

B. *Execution time*

The algorithm's execution time for different number of processes, number events, and number of messages was recorded. The number of variables used and the number of events that were sharing these variables was 3 or 5 as indicated. All 3 types of dependencies were explored. The elapsed times were recorded using *System.nanoTime()* and averaged over 100 runs. The average number of generated computations over the 100 runs is also provided. Table 1 presents the results. The tests were performed on a late-2013 MacBook Pro with the following specifications:

- Processor: 2.4 GHz Intel Core i5
- Memory: 8 GB 1600 MHz DDR3
- Operating System: OS X version 10.11.4

Processes	**Events/ process**	**Messages**	**Variables-repetition**	**Avg. time (ms)**	**Avg. list size**
3	3	3	5-5	5.12	143
3	3	5	5-5	2.64	49
3	3	7	5-5	2.0	25
3	5	3	5-5	67.9	949
10	5	5	3-3	173.9	295
10	10	10	3-3	1896.6	937

Table 1: Average execution time and output list size for various input computations

When the total number of processes and total number of messages are held constant, the average execution time and average size of the generated computations list increase with the number of events per process. This is expected, since more synchronization messages need to be added, and more valid possible orderings can be generated.

When the number of processes and the numbers of events per process is held constant, the average execution time and the average size of the generated computations list decreases for higher number of messages. This is expected, since the existing messages would already establish a large fraction of the happened-before relationships.

V. Conclusion

In this project, we implemented a tool that helps the developers of a distributed computation test their applications for data dependency bugs. The tool works by generating a list of similar computations that explore all the different orders in which concurrent events of interest can execute. Our tool also allows the developer to specify which types of dependencies they are interested in, in order to restrict the search space and the number of generated computations to be tested. A possible direction for future work is to consider data dependencies across multiple shared variables that are being accessed, and to offer the developer more control over the way the computations are generated, in order to further restrict the number of generated computations.

VI. References

[1] Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." Communications of the ACM 21.7 (1978): 558-565.

[2] Garg, Vijay K. Elements of distributed computing. John Wiley & Sons, 2002. 11-25. Print.

[3] Krothapalli, V. Prasad, and P. Sadayappan. "An approach to synchronization for parallel computing." Proceedings of the 2nd international conference on Supercomputing. ACM, 1988.