Designing and Running Test Suite for a Distributed System Controlled Re-execution Program

David, Rachelle
Electrical and Computer Engineering
UT Austin
Austin, TX
rd33366

Lopez, Daniel
Electrical and Computer Engineering
UT Austin
Austin, TX
lopezd

Castillo, Jason
Electrical and Computer Engineering
UT Austin
Austin, TX
jzc248

***Abstract*—This paper provides a very brief overview of an implementation of the Predicate Control: Synchronization in Distributed Computations with Look-ahead paper that adds a novel implementation of a controlled re-execution visualization model inspired by the paper. Then goes over how software testing was added to account for any undesirable behaviours that made be present in the code.**

I. Introduction

Predicate control is a problem of synchronization across an offline distributed computation in order to maintain a global predicate. There are two types of interactions between an application its run-time environment: observation and control. While observation is typically events like monitoring load and detecting failures, control covers topics like dynamically balancing load, recovering from failures, and resetting variable values when debugging. Since we are focusing on distributed control, the real difficulty lies in ensuring the global conditions via local control actions. Furthermore, we are specifically interested in synchronization, controlling the relative timing among processes. The main difficulty in predicate control is maintaining the given property without causing deadlock with the existing synchronizations.

The model presented here visualizes a distributed system passing messages between nodes with a ball being passed among nodes arranged in a grid. There is a specific thread responsible for capturing the global state, and then visualizing the result. Inevitably, errors will occur in which the ball, or message, is no longer being passed or is no longer trackable. This paper will discuss the inspiration for this controlled re-execution of passing the ball and the algorithms used.

II. Problem Overview

A. Predicate Control

If we model a computation as a partially ordered set of events, then we can re-frame the predicate control problem as to determine how to add edges to a computation so that it maintains a global predicate. Essentially, this is a question of how to make the partial order stricter. However, the predicate control problem should be able to handle any kind of predicate.

The source paper analyzes three classes of predicates. The first is "disjunctive predicates," which express a global condition in which at least one local condition has occurred. An example of this is "at least one server is available," or in our model, "at least one node has the ball." The second class of predicates is "mutual exclusion predicates," in which no two processes are in the critical sections at the same time. From the perspective of the ball-passing model, this could be "no two nodes have the ball at the same time." These two classes form basic interpretations of the predicate control problem.

The third class is a generalization of mutual exclusion to include two additional properties. One of these types of predicates are "readers writers predicates," which specifies that only "write" critical sections must be exclusive but "read" critical sections are not exclusive. The second generalization is of "independent mutual exclusion predicates," in which critical sections have "types" associated with them so that no two critical sections of the same type can execute simultaneously. Thus, the class of "generalized mutual exclusion predicates" allows both read and write critical sections as well as multiple types of critical sections. In terms of our model, we can think of these as "read: ask a node if they have the ball; write: pass the ball." Our focus will be on this third class of predicates.

B. Computations and Intervals

Before we discuss the algorithms in depth, we must first present supporting frameworks for predicates, processes, and problems. A distributed computation, which we will just call a **computation**, is a tuple $\langle E_1, E_2, ...E_n, \rightarrow\rangle$ where the E_i's are disjoint finite sets of "events" or "processes" and $\rightarrow$ or "precedes" is an irreflexive partial order. We abbreviate this computation to be $\langle E, \rightarrow\rangle$ with the understanding that the size of the partition of E is n. In this partial ordering, if $e \rightarrow f$ then we say e "casually precedes" f. Furthermore, a computation is a *run* if $\rightarrow$ is a total order.

Predicates represent our understanding of properties on local or global states. Properties local to a process might be whether the process is in the critical section, whether the process has a token, and so on. Global states include properties across multiple processes, such as whether two processes are in the critical sections, whether at least one process has a token, and comparison of variables on different processes.

Each E_i has a corresponding special "dummy" event $\bot_i$ such that $\bot_i \notin E$ and it initializes the state of process E_i. Additionally, for each i, we let $\prec_i$ be the smallest relation on $E_i \cup \{\bot_i\}$ such that $\forall e \in E_i : \bot_i \prec_i e$, and $\forall e, f \in E_i : e \rightarrow_i f \Rightarrow e \prec_i f$.

An **interval** I is a non-empty subset of an $E_i \cup \{\bot_i\}$ corresponding to a maximal subsequence in the sequence of events in $\langle E_i \cup \{\bot_i\}, \prec_i\rangle$, such that all events in I have the same value for α_i, in which α_i's comprise a set of local predicates.

Given a set of intervals, we use $I_1 \mapsto I_2$ to represent the notion that "I_1 must enter before I_2 can leave."

In order to formally state the problem, we need one more concept, that of a "controlling computation," which intuitively is a stricter computation for which all consistent cuts satisfy the predicate. More formally, given a computation $\langle E, \rightarrow\rangle$ and a global predicate ϕ, a computation $\langle E, \rightarrow^c\rangle$ is called a **controlling computation** of ϕ in $\langle E, \rightarrow\rangle$, if: (1) $\rightarrow \subseteq \rightarrow^c$, and (2) for all consistent cuts C in $\langle E, \rightarrow^c\rangle : \phi(C)$.

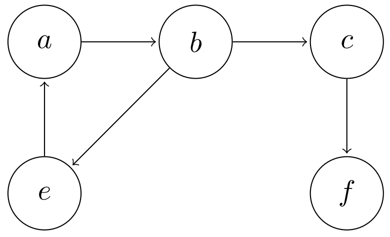


Fig. 1. A Directed Graph used to demonstrate SCCs

Now, we can define the **Predicate Control Problem** as: Given a computation $\langle E, \rightarrow\rangle$ and a global predicate ϕ, is there a controlling computation of ϕ in $\langle E, \rightarrow\rangle$?

Before we discuss approaches to solve this problem, we need to define one more key feature: strongly connected components. A directed graph is considered strongly connected if there is a path between all pairs of vertices. A **strongly connected component (SCC)** of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the figure 1, the subgraphs are as follows: $SCC = \{\{a, b, e\}, \{c\}, \{f\}\}$.

Now, we have all the tools necessary to solve the predicate control problem.

C. Generalized Mutual Exclusion Predicate Algorithm

In order to design an algorithm to solve the generalized mutual exclusion predicate control problem, we first start with the simplified classes of predicates, starting with Disjunctive Predicates. The central idea is that the algorithm inputs n sequences of intervals for each of the processes and produces a sequence of added edges. This sequence of added edges links true intervals into a continuous "chain" from the start of the computation to the end. Any cut must either intersect this chain in a true interval, in which case it satisfies the disjunctive predicate, or in an edge, in which case the cut is made inconsistent by the added edge.

When we consider Mutual Exclusion Predicates, we make use of the fact that the critical sections in a process are totally ordered. The key idea used in the algorithm is to maintain a frontier of critical sections that advances from the start of the computation to the end. Instead of finding a minimal critical section of the whole interval graph, we merely find a minimal critical section in the current frontier. It is guaranteed to be a minimal critical section of the remaining critical sections in the interval graph at that point.

Utilizing the previously described algorithms, we could design a simple algorithm based on determining the strongly connected components in the critical section graph and then topologically sorting them. Instead, we use the concept of a "general interval" or a sequence of intervals in a process that belong to the same strongly connected component of the interval graph.

The algorithm maintains a frontier of general critical sections that advances from the beginning of the computation to the end. In each iteration, the algorithm finds the strongly connected components (scc's) of the general critical sections in the frontier. Then, it picks a minimal strongly connected component, a *candidate*, from among them. However, the *candidate* is not necessarily a minimal scc of the entire critical section graph. In fact, it need not even be an scc of the entire graph. To determine if it is, we find the *mergeable* set of critical sections that immediately follow the general critical sections and belong to the same scc. If *mergeable* is not empty, the critical sections in *mergeable* are merged with the general critical sections in candidate to give larger general critical sections and the procedure is repeated. If *mergeable* is empty, then it can be shown that *candidate* is a minimal scc of the graph. Therefore, we check that it meets the sufficient conditions of validity, and then append it to the chain. After the main loop terminates, the scc's in the chain are connected using added edges which define the controlling computation, and the algorithm is complete. A simple implementation of the algorithm will have a time complexity of $O(n^2p)$. However, a better implementation of the algorithm would be $O(np)$ by avoiding redundant computations.

III. Applications

The Predicate Control Problem has a few general applications, and we will focus on two specific domains: software fault-tolerance and distributed debugging. We first discuss the challenges with applying predicate control in the real-world, then briefly observe fault-tolerance.

When mapping the abstract concept of predicate control, some issues arise. For the computation, tracing is a prerequisite for applying predicate control, and tracing takes time and memory. From a global predicate perspective, there always must be a shared resource that interacts with the system. Finally, in order to implement the controlling computation, there must be a method of replaying the traced computation and overlaying the synchronizations. This takes time and memory, and is a point of research in itself.

Distributed systems require fault tolerance, since programs often experience failures, such as races, due to synchronization faults. Some recovery methods include:

1) simple re-execution: simply re-execute and hope that the race does not re-occur
2) locked re-execution: apply file-system locks to each access during re-execution
3) controlled re-execution: add synchronizing messages to implement predicate control

Our implementation leverages specifically controlled re-execution as the motivation for our model.

IV. Our Implementation

When analyzing the predicate control problem and related solutions previously presented, we found it most useful to visualize the system and processes sending messages from one to another. As previously discussed, in order to synchronize and maintain that global perspective, there sometimes occurs a fault in which the global state loses track of this passing of a message. As a result, we implemented a controlled re-execution application of the predicate control problem.

A. Model

In order to visualize a distributed system, we chose to implement a two-dimensional array of nodes, or processes, arranged in a grid. Each node is connected to at most 8 of its available neighbors: columns, rows, and diagonals. Figure 2 demonstrates the arrangement of this model.

In our implementation, each node is a two dimensional array of $16x16$ integers (i.e. `int[16][16]`) initialized to 0's and runs its own thread.

In this visualization of a distributed system, we use a red ball to represent a change in state within that node, or process, and we notify the "next" node of that change via a message. As a result, the ball essentially demonstrates a message being passed between nodes.

The ball is a 32-bit integer, greater than 0, in which the lower 4 bits indicate the direction of movement.

In this model, what we visualize is the global state as captured by an observation thread. This thread is responsible for finding the ball within the network and drawing it to the screen as a red dot on a canvas. This canvas is the visual representation of our grid system, so that we can observe the current global state as well as controlled-re-execution as faults occur.

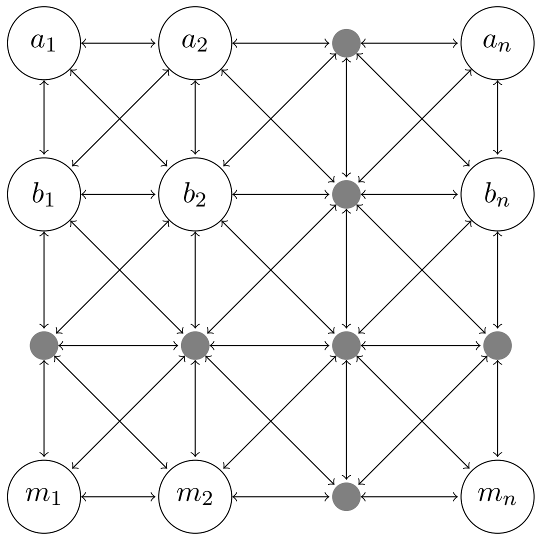


Fig. 2. Visualization of the System Model

B. Direction Mapping

The implemented state machine will move a ball in the direction it is currently moving in, until it reaches a node at the corner or edge. At that point, it will "bounce" off the edge in a supplementary angle and off a "corner" in a complementary angle. The "bounce" involves changing the lower 4 bits of the ball, which correlate to the direction of movement.

Figure 3 demonstrates how directions are mapped in ball movement.

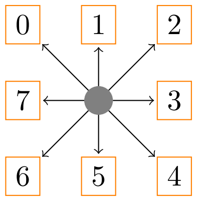


Fig. 3. Direction Map

C. Controlled Re-execution

When applying these theoretical models in the real-world, we must practically synchronize and maintain the global perspective, and inevitably faults occur in which the observation thread loses track of the ball being passed in its global snapshot. We were inspired by the Generalized Mutual Exclusion Predicate Algorithm to implement a controlled re-execution application of the predicate control problem. However, we found it impractical to scale to a large quantity of processes in the system, as we had around 800-1000 nodes.

We leveraged the Generalized Mutex Algorithm since we can present our model as a generalized mutex predicate: a thread must *read* to "ask a node if they have the ball" while a thread must *write* to "pass the ball."

As we scaled, we used the basis of this algorithm to pursue a more practical method of controlled re-execution, in which we leverage timestamps to revert to a previous global snapshot and re-execute from there. Our algorithm is generally as follows:

```
iterate through every node $i$:
    if $i$'s timestamp of last having the ball is greater than our running greatest timestamp then:
        update the running greatest timestamp to $i$'s timestamp of last having the ball
    reset $i$'s timestamp to 0
set timestamp of the last node to have the ball to 1
re-execute the ball passing from the location of the greatest timestamp to last have the ball
```

With this algorithm, the observation thread essentially searches until it finds the node that last had the ball with the highest timestamp, resets all the nodes, and re-executes.

D. Adding Complexity

The model we presented thus far only incorporates a single "ball" being passed, however in a different model, it is possible for multiple balls to be passed simultaneously. This would change our predicate to have multiple critical sections characterized by the passage of multiple balls.

Although we did not implement multiple balls, a few challenges arise with accommodating this complexity. Firstly, we must decide how we want to model intersecting balls. Our assumption would be that we need to include channels to control how balls are passed. This would also alleviate the question of intersecting balls, since balls on different channels can pass through each other. Secondly, we must decide how we handle re-execution when one ball faults. Do we revert back to the global snapshot for all balls? Or only for the ball with the fault? We propose to revert back to the previous non-faulty state for all balls, in order to prevent further complexity of ball tracking.

E. Testing Opportunity

Although the initial implementation of the distributed system algorithm demonstrated success in analyzing re-execution and predicate control, concerns arose regarding potential unnoticed undesirable behaviors. Initial testing during this implementation phase relied on manual verification and standard code debugging techniques. However, the adoption of a comprehensive software test suite allows for a systematic evaluation of the code, covering all possible edge cases and diverse ball movement directions. This approach enhances the robustness of the testing process, ensuring a more thorough examination of the distributed system's behavior.

F. Test Suite Implementation

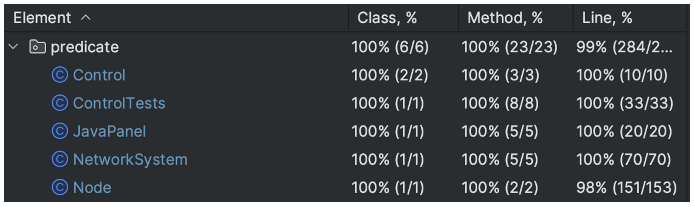
JUnit tests were designed to assess the functionality of how the message traversed the network using the IntelliJ IDE. These tests cover various scenarios, including movements in different directions (up, down, left, right, and combinations), ball finding, and the execution of the main method in the Control class. Each movement test initializes a Network System, simulates a specific movement, and verifies the success of the movement by examining the lastClock property of a particular cell in the world. The ball-finding test sets up conditions for a ball to be present, calls the findBall method, and checks if the ball is successfully detected. Overall, these tests provide a comprehensive evaluation of the NetworkSystem class's core functionalities and node coverage.

V. Analysis and Conclusion

In the original implementations of this model for the distributed system class, a system grid of $35x25$ nodes was mostly used, resulting in 875 total nodes on our system. With a system of this size, it was nearly always encountered a fault within minutes of initial execution. This reinforced a need for fault tolerance and recovery methods for real world networks and why a robust testing suite is needed.

It was identified that most of the original code was hard coded and not setup for testing with a JUnit test class. The original code had to be modified to allow values to be passed into the network and ball creation methods to test different paths and network sizes. The tests were ran with test coverage and initially we only had 25% Node coverage and we would the tests would end prematurely. After modification of test order and adding tests, we were able to achieve a 98% node coverage as seen in figure 4.

The test suite was able to identify numerous issues and pinpoint their fault locations in the code. After step by setp corrections we were able to make the program more robust. This project was a good exercises to show the need of software testing to make any code more robust and ready for the field.



Element	Class, %	Method, %	Line, %
predicate	100% (6/6)	100% (23/23)	99% (284/2...
Control	100% (2/2)	100% (3/3)	100% (10/10)
ControlTests	100% (1/1)	100% (8/8)	100% (33/33)
JavaPanel	100% (1/1)	100% (5/5)	100% (20/20)
NetworkSystem	100% (1/1)	100% (5/5)	100% (70/70)
Node	100% (1/1)	100% (2/2)	98% (151/153)

Fig. 4. Node Coverage at 98%

References

[1] A. Tarafdar and V. K. Garg, "Predicate Control: Synchronization in Distributed Computations with Look-ahead," UT internal paper.

[2] GeeksForGeeks, "Strongly Connected Components," website, 2022.