

Inferring Invariants of Distributed Programs

Keheliya Gallaba*, Arash Vahabzadeh†

University of British Columbia

Vancouver, BC, Canada

Email: *kgallaba@ece.ubc.ca, †arashvhb@ece.ubc.ca

I. INTRODUCTION

Building a reliable distributed system is a difficult task, since distributed system should continue operating even when components or network fail, this means that there are many corner cases that developers should consider when building the system. Building such a reliable and dependable system is not possible without thorough testing of the system. However, unfortunately testing a distributed system is a much more difficult task because it is much harder to test components of a distributed system in isolation. There are many failure scenarios that distributed system should handle and these scenarios should be tested as well. In some cases reproducing the failures is a difficult task e.g. degrading network performance for test purposes. Many bugs turn out to be heisenbugs in distributed system which are hard to reproduce and detect. For testing and debugging distributed systems, developers and testers have to spend a considerable amount of time inspecting logs of the system manually and looking for incidents that system does not behave as expected. One alternative to this laborious approach is to define invariants for system's expected behaviour and check the software traces against these invariants. This approach is considerably easier than the former but as the system gets larger and more complex, providing invariants manually becomes more difficult.

Testing distributed systems is hard because a fault in a single node does not confine to that node, instead it may cause an error in another node. In fact errors in distributed system should be defined on global state of the system not a single node. In this work we assume that state of each node is comprised of its variables. Not all variables of one node is affected by other nodes, we define the set of variables of one node whose values are affected by other nodes as distributed state of that node. We refer to set of distributed state of all nodes as distributed state of the system.

II. MOTIVATING EXAMPLE

For motivation example, consider sudo-2-phase-commit protocol of figure 1. In this protocol coordinator first queries other nodes for their vote and if all nodes including the coordinator vote "True" then coordinator sends "Commit", otherwise it sends "Abort" to all other nodes. The correctness invariant of this protocol is that at the end of protocol value of commit variable of all nodes should be the same.

III. BACKGROUND

Invariants are properties that should hold in certain points in program. Although previous works [?] showed that invariant inference is feasible in sequential programs, inferring invariants in a distributed system poses unique challenges and

Listing 1. Coordinator Code

```
1 for i := 1 to n do
2   send('Q', i)
3 commit := decide()
4 for i := 1 to n do
5   recv(buf, i)
6   if (buf == 'A')
7     commit := 'A'
8 for i := 1 to n
9   send(commit, i)
```

Listing 2. Coordinator Code

```
1 recv(buf)
2 if (buf == 'Q')
3   vote := decide()
4   send(vote)
5 if (vote == 'A')
6   commit = 'A'
7 else
8   recv(buf)
9   commit = buf
```

Fig. 1. Sudo 2 phase commit code for coordinator and cohorts

still is a open problem. To infer invariants in a distributed system we should have global snapshots (log of variables) of the system. However, not every snapshot of the system is a valid one. Mattern introduced the notion of cut and consistent cut as snapshot and valid snapshots of a system. In [?] he also proposed an algorithm to find consistent cuts. The notion of distributed state first proposed by Ousterhout in [?] as "information retained in one place that describes something, or is determined by something, somewhere else in the system". We use dynamic data flow analysis to detect distributed state of the system. In fact we perform data flow analysis across the nodes in the distributed system.

Our approach is to automatically infer invariants that may exist in distributed state of our system. To this end, we execute distributed programs by running the test cases and dynamically select a set of variables in each node that likely represent the distributed state across our distributed nodes. We do so with dynamic data flow analysis of variables and selecting those that are affected by a send or receive operation. Finally we use Daikon to infer invariants among our distributed state variables.

We can view our approach as an optimization. The naive approach is to log all of the variables at each instruction execution and give all consistent cuts of the system to daikon to infer invariants between the variables. However, this naive approach is not efficient at all, if a total number of n instruction is executed during a run by all nodes in the system

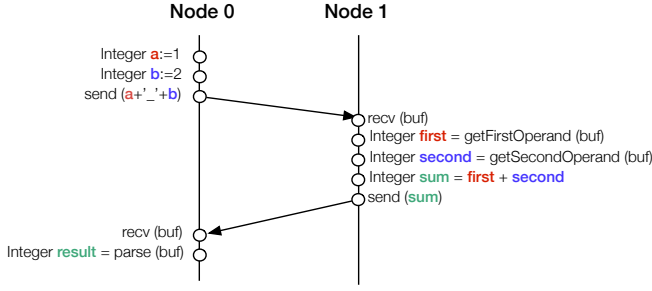


Fig. 2. Example communication between two nodes in a distributed system

and on average we have m variables for each node, we should log $n * m$ variables but many of these variables are local to one node and are not affected by other nodes, so we may even end up finding many false positives (invariants that hold by accident and are not meaningful). Our approach only logs variables that an invariant might exist between them by using data flow analysis and finding group of variables (across the nodes) which affect each other. Another optimization that our approach does is that instead of logging the variables at each instruction, we only log them at consistent cuts annotated by developers. Based on our observations some must-hold invariants can be violated for small amount of time (e.g. execution time of several instructions) until systems goes to a stable state, so we decided to give developers responsibility to decide when invariants should hold and can be checked in the system by annotating the code. By this approach we improve usefulness of our tool for developers. Another approach can be inferring the invariants from logs of the system, our approach removes the burden from developers to decide which portion of node's state needs to be logged. We automatically infer variables that are likely to have invariants between them with data flow analysis.

IV. APPROACH

The ultimate goal of this project is to infer invariants in a distributed system. For the scope of this project we will consider a system with 2 nodes. Before inferring invariants we need to identify likely distributed state variables. We define distributed state variables, as variables that are stored in one node whose values are affected or determined by the other remote node in the distributed system. It should be noted that these variables are only a part of the state in a distributed system at a given time. For our case we assume that these capture a significant portion of the state. Consider the code samples Listing 3 and Listing 4 showing the communication between two nodes in a distributed system. This conversation is depicted visually in Figure 2. Here it will be useful to infer the equality between elements in each pair $(a, first)$, $(b, second)$, $(sum, result)$ and also the properties: $(result = a + b)$ and $(sum = first + second)$.

The naive approach for inferring these invariants will be to dump all state variable values at all program points and feed them to an invariant detector like daikon[?]. But given the number of program points and huge number of variables in a relatively large program this approach is not scalable. So

Listing 3. Sample code for Communication between 2 nodes - Node 0

```

1 Integer a := 1
2 Integer b := 2
3 send ( a + '_' + b )
4 recv (buf)
5 Integer result := parse (buf)

```

Listing 4. Sample code for Communication between 2 nodes - Node 1

```

1 recv (buf)
2 Integer first := getFirstOperand( buf )
3 Integer second := getSecondOperand( buf )
4 Integer sum := first + second
5 send ( sum )

```

we propose two main optimizations to reduce the number of program point, variable value combinations.

A. Detecting interesting program points

Rather than dumping state at all program points we expect the developer to mark interesting program points using annotations. The program state will be dumped at only these annotated points. This will also help to narrow down the focus to points that are identified by the developer as interesting points without wasting resources analyzing useless program points. The annotations will be identified by the parser at the instrumentation phase and re-written to include statements which will write the values of variable to a file when the program reaches that point in the runtime.

Since there can be multiple points annotated in a program by a developer, vector clocks will be used to partially order annotation pairs in the two programs. This will also help to invalidate annotations put at irrelevant locations by the developer.

B. Dataflow analysis for Go programs

Even at interesting program points, variable space that need to be captured is very large. So only the variables that were affected by the communication with other nodes need to be identified. To detect these variables, some form of dataflow analysis is required. This includes the dataflow of the distributed system as a whole and also in a single program. But currently there are no data flow analysis frameworks available for Go programs.

This is a challenging problem in itself because Go language has special constructs like channels to share data between goroutines. We are planning to do this analysis dynamically by instrumenting variable declarations and assignments, and then collecting execution traces of each of these statements. We have chosen this approach because it gives information about the actual execution and therefore is more accurate. For this we are hoping to use static analysis libraries available under Go-lang tools[?] which provide support for Identifier resolution, Type information, Call graph navigation and Channel peers (send \leftrightarrow receive) identification.

Our dynamic dataflow analysis is focused on mainly providing the following features. Given a variable, start line and end line it will return a set of variables affected by that variable between start line and the end line.

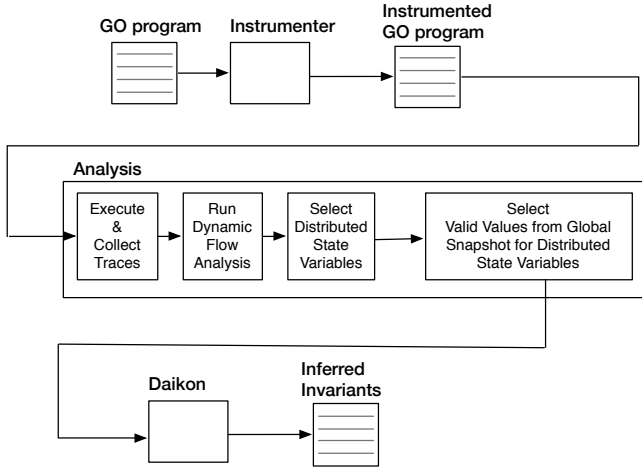


Fig. 3. Flow of the invariant inference of Go programs

As the next phase of our project, using the above analysis, we hope to find out variables affected by receiving messages in interprocess communication channels like RPC and TCP/UDP sockets. We will get the values of these variables just before another send/receive or end of function whichever occurs first. We assume these set of variables at the send statement in both (sending and receiving) nodes taken together represent global state of the distributed system.

Given two nodes which are communicating we have two such sets of variables representing a particular state. So we will select the corresponding variable pairs from send and receive of each node mentioned in the previous step and provide as inputs for the Daikon invariant detector. This will give us a set of properties that were true over the observed executions across 2 nodes. Overall execution flow of the analysis is shown in Figure 3.

V. EVALUATION METHODOLOGY

VI. RELATED WORK

Yabandeh et al.[?] proposed an approach to infer almost-invariants in distributed systems. They infer the invariants that are true in most cases and assume these invariants only get violated due to manifestation of bugs. There are a number of ways our approach differ from theirs: Their approach requires user to provide a list of variables and functions that they want to consider for inferring invariants while our approach infers distributed state variables automatically. Moreover, they assume that an external module generates a trace of globally consistent state system for their algorithm while our approach extract these states through dynamic execution of the distributed system.

Ernst et al.[?] infer invariants in a sequential program by executing the program on a collection of inputs. This invariant detector has been released as a tool called Daikon[?]. Although, we use their tool to infer invariants, detecting invariants in a distributed system poses unique challenges to identify distributed state and valid global states among a larger set of variables. Ne Win et al.[?] use daikon to assist theorem provers for verifying distributed algorithms. To

demonstrate their approach, they proved the correctness of the Paxos algorithm.

Beschastnikh et al [?] infer temporal invariants from partially ordered logs of the distributed system. By using logs of the system, they put the responsibility on developers to decide which portion of state of the program can be used to infer invariants. We require developers at which point in program invariants can be checked but we infer the portion of state that can be checked for invariants automatically with data flow analysis.

VII. ACKNOWLEDGMENT

We thank Ivan Beschastnikh for his valuable feedback on initial draft of this proposal.