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. But 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 systems 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.

Our approach is to automatically infer invariants that may exist in distributed state of our system. To this end, we execute distributed programs 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.

II. 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 1 and Listing 2 showing the communication between two nodes in a distributed system. The communication between the nodes can be depicted as shown in Figure 1. We need to infer the equality between elements in each pair `(a,first), (b,second),(sum,result)` and also the properites: `(result = a + b)` and `(sum = first + second)`.

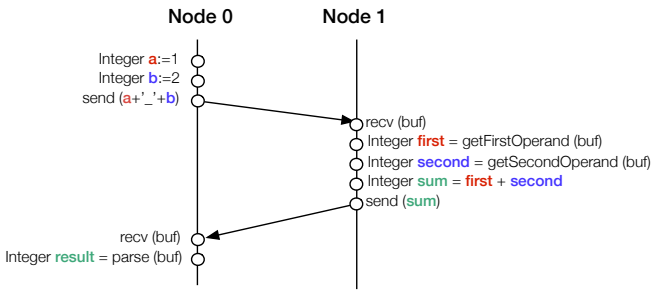

Fig. 1. Example communication between two nodes in a distributed system

Listing 1. 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)
```

The naive approach for inferring these invariants will be to dump all state varaible values at all program points and feed them to an invariant detector like daikon[1]. But given the number of program points and huge number of variables in a relatively large program this approach is not scalable. So we propose 2 main optimizations to reduce the number of program point, variable value combinations.

Listing 2. 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 )
```

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.

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 Golang tools[2] 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.

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 2.

C. Timeline

III. Related Work

Yabandeh et al.[3] proposed an approach to infer almost-invariants in distributed systems. They infer the invariants that

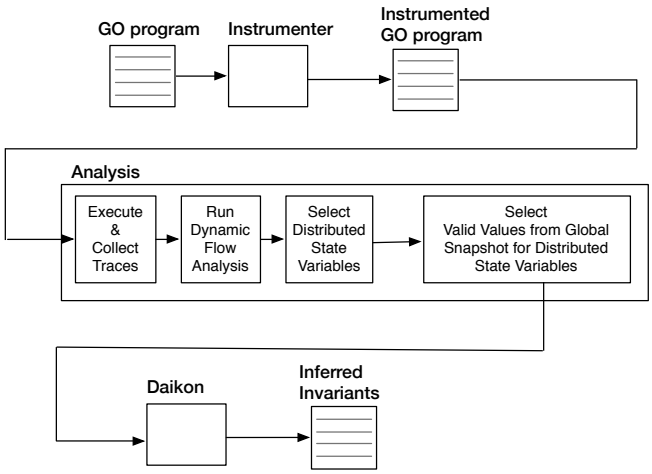

Fig. 2. Flow of the invariant inference of Go programs

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.[4] 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[1]. 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.[5] use daikon to assist theorem provers for verifying distributed algorithms. To demonstrate their approach, they proved the correctness of the Paxos algorithm.

References

[1] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[2] "Static analysis features of godoc - the go programming language," http://golang.org/lib/godoc/analysis/help.html.

[3] M. Yabandeh, A. Anand, M. Canini, and D. Kostic, "Finding almost-invariants in distributed systems," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011, pp. 177–182.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.

[5] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch, "Using simulated execution in verifying distributed algorithms," *Software Tools for Technology Transfer*, vol. 6, no. 1, pp. 67–76, Jul. 2004.