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, 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 and not on 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 variables of that node. We refer to set of distributed state variables of all nodes as distributed state of the system.

II. Motivating Example

For motivation example, consider pseudo-two-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 "Commit" then coordinator sends "Commit", otherwise it sends "Abort" to all other nodes. At the end of this protocol all nodes should either commit or abort . In order to verify that this algorithm is working correctly, developers can examine inferred invariants of the protocol execution and inspect if these inferred invariants match expected behaviour of the system (In this case commit value of all nodes should have the same value and this property should be inferred by executing the protocol). We use daikon [1] to infer invariants between variables so we can infer linear ($y = ax+b$), ordering ($y < x$) and containment ($x \in y$) relationships that may hold between the distributed state variables of all nodes in the system.

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. Pseudo two phase commit code for coordinator and cohorts

III. Background

Invariants are properties that should hold in certain points in program. Although previous works [1] showed that invariant inference is feasible in sequential programs, inferring invariants in a distributed system poses unique challenges and 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 [2] he also proposed an algorithm to find consistent cuts. The notion of distributed state first proposed by Ousterhout in [3] as "information retained in one place that describes something, or is determined by something, somewhere else in the system", we assume that variables whose value is affected by other nodes consists a significant portion of distributed state of that node. We use dynamic data flow analysis to detect the distributed state of the system. We also use vector clocks to partially order snapshots of the nodes in our system and form consistent cuts.

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 to solve inference problem 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 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 observation 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 at which points in program invariants should hold and can be checked 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 each 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 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)`.

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[4]. But given the number of program points and huge number of variables in a relatively large program this approach is not scalable. So

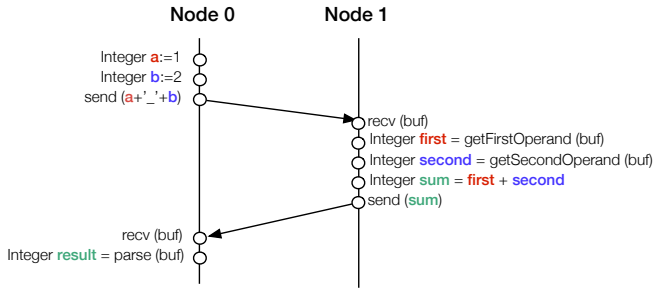

Fig. 2. Example communication between two nodes in a distributed system

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)
```

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 inserted 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. For example using a forward flow analysis on Listing 4 we can say that `sum` variable at line 4 and line 5 is affected by the `buf` variable at line 1. But currently there are no data flow analysis frameworks specifically for Go programs.

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 )
```

So we are planning to do a dynamic dataflow analysis which consists of following steps:

- Instrumenting variable declarations and assignments at the run time
- Collecting execution traces of each of these statements by runnning test suites
- Doing a forward flow analysis using the collected traces

This dynamic analysis approach was chosen because it gives information about the actual execution and therefore is more accurate. And also compared to using a static analysis-based approach, this approach makes it easier to handle Go language-specific constructs like channels which are used share data between goroutines. Only drawback in this approach is, it only examines the paths invoked during execution, unlike static analysis which can be used to reason about all possible execution paths and variable values. To compensate for that we are planning to use a comprehnsive test suite with good test coverage.

Overall execution flow of the analysis is illustrated in Figure 3 and works as follows:

- Developer annotates interesting program points in the Go program which will be used as states for mining invariants.
- Program is instrumented to
  - dump variable values at the annotated program points at run time
  - wrap all RPC communication to include vector clocks
  - log all declarations and assignments for doing the forward flow analysis

  For this we are hoping to use static analysis libraries available under Go-lang Parse package and tools[5] which provide support for Identifier resolution, Type information, Call graph navigation and Channel peers (send $\leftrightarrow$ receive) identification.
- Then as the first step in the analysis phase, this instrumented program is executed using the test suites and program traces are collected.
- Forward flow analysis is done on the collected traces
- Using the results of the previous step, variables contributing to distributed state is identified.
- From the dumped variable values at annotated program points (after multiple executions), only the values of variables chosen in the previous step are selected.
- Selected variable values belonging to 2 distinct annotated program points in the 2 nodes are fed to Daikon for inferring invariants.

V. Timeline

Following is the planned milestones and estimated timeline for the project completion.

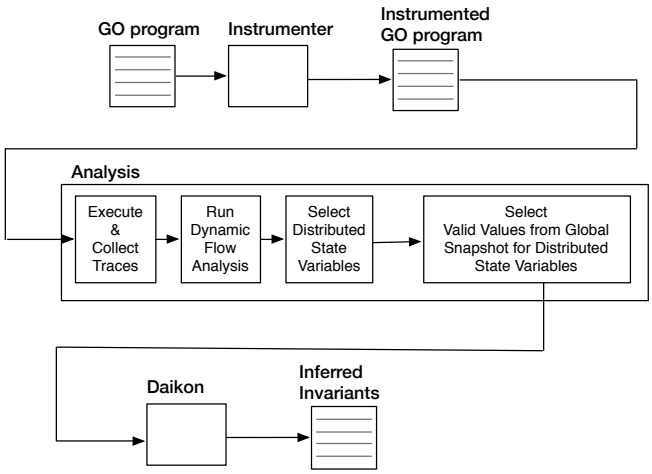

Fig. 3. Flow of the invariant inference of Go programs

2/10: Proposal Draft Submission
3/3: Proposal Submission
3/7: Finalizing the design, Familirization with required tools (Daikon, Go source analysis tools)
3/14: Implementation - Go program instrumentation to handle annotations and dumping state
3/21: Implementation - Dynamic Forward Analysis for Go programs
3/28: Implementation - Vector clocks integration and end-to-end working analysis
4/4: Evaluation - Inferring invariants in test programs
4/7: Project Presentation
4/16: Project Code and Report Submission

VI. Evaluation Methodology

We will implement our approach in a tool in Go programming language. We plan to evaluate our tool by verifying correctness of two-phase commit and leader election protocols. Our tool should be able to infer safety property of both algorithms. In two-phase commit protocol all nodes should perform the same action (commit or abort) so our tool should infer this property. In leader election algorithm there should be only one leader (uniqueness property) and at the end of protocol execution all nodes should know ID of the leader (agreement property). It is worth mentioning that our approach is not suitable for checking the liveness properties of these algorithms.

VII. Related Work

Yabandeh et al.[6] 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.[1] 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[4]. 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.[7] 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. [8] 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 the state of the program can be used to infer invariants. Although, we require developers to specify at which points in the program invariants can be checked, we infer the portion of the state that can be checked for invariants automatically with data flow analysis.

VIII. Acknowledgment

We would like to thank Ivan Beschastnikh for his valuable insight and feedback on initial draft of this proposal.

References

[1] 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.

[2] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[3] J. K. Ousterhout, "The role of distributed state," pp. 199–217, 1991.

[4] 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.

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

[6] 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.

[7] 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.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, "Mining temporal invariants from partially ordered logs," in *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, ser. SLAML '11. New York, NY, USA: ACM, 2011, pp. 3:1–3:10.