Trace-Based Run-Time Analysis of Message-Passing Go Programs

Martin Sulzmann and Kai Stadtmüller

Faculty of Computer Science and Business Information Systems
Karlsruhe University of Applied Sciences
Moltkestrasse 30, 76133 Karlsruhe, Germany
martin.sulzmann@hs-karlsruhe.de
kai.stadtmueller@live.de

Abstract. We consider the task of analyzing message-passing programs by observing their run-time behavior. We introduce a purely library-based instrumentation method to trace communication events during execution. A model of the dependencies among events can be constructed to identify potential bugs. Compared to the vector clock method, our approach is much simpler and has in general a significant lower run-time overhead. A further advantage is that we also trace events that could not commit. Thus, we can infer more alternative communications. This provides the user with additional information to identify potential bugs. We have fully implemented our approach in the Go programming language and provide a number of examples to substantiate our claims.

1 Introduction

We consider run-time analysis of programs that employ message-passing. Specifically, we consider the Go programming language [4] which integrates message-passing in the style of Communicating Sequential Processes (CSP) [6] into a C style language. We assume the program is instrumented to trace communication events that took place during program execution. Our objective is to analyze program traces to assist the user in identifying potential concurrency bugs.

Motivating Example In Listing 1.1 we find a Go program implementing a system of newsreaders. The main function creates two synchronous channels, one for each news agency. Go supports (a limited form of) type inference and therefore no type annotations are required. Next, we create one thread per news agency via the keyword go. Each news agency transmits news over its own channel. In Go, we write ch <- "REUTERS" to send value "REUTERS" via channel ch. We write <-ch to receive a value via channel ch. As we assume synchronous channels, both operations block and only unblock once a sender finds a matching receiver. We find two newsreader instances. Each newsreader creates two helper threads that wait for news to arrive and transfer any news that has arrived to a common channel. The intention is that the newsreader wishes to receive any news whether it be from Reuters or Bloomberg. However, there is a subtle bug (to be explained shortly).

© Springer International Publishing AG 2017
https://doi.org/10.1007/978-3-319-70389-3_6
func reuters(ch chan string) { ch <- "REUTERS" } // r!
func bloomberg(ch chan string) { ch <- "BLOOMBERG" } // b!

func newsReader(rCh chan string, bCh chan string) {
    ch := make(chan string)
    go func() { ch <- (<- rCh) }() // r?; ch!
    go func() { ch <- (<- bCh) }() // b?; ch!
    x := <-ch // ch?
}

func main() {
    reutersCh := make(chan string)
    bloombergCh := make(chan string)
    go reuters(reutersCh)
    go bloomberg(bloombergCh)
    go newsReader(reutersCh, bloombergCh) // N1
    newsReader(reutersCh, bloombergCh) // N2
}

Listing 1.1. Message passing in Go

Trace-Based Run-Time Verification  We only consider finite program runs and therefore each of the news agencies supplies only a finite number of news (exactly one in our case) and then terminates. During program execution, we trace communication events, e.g. send and receive, that took place. Due to concurrency, a bug may not manifest itself because a certain ‘bad’ schedule is rarely taken in practice.

Here is a possible trace resulting from a ‘good’ program run.

r!; N1.r?; N1.ch!; N1.ch?; b!; N2.b?; N2.ch!; N2.ch?

We write r! to denote that a send event via the Reuters channel took place. As there are two instances of the newsReader function, we write N1.r? to denote that a receive event via the local channel took place in case of the first newsReader call. From the trace we can conclude that the Reuters news was consumed by the first newsreader and the Bloomberg news by the second newsreader.

Here is a trace resulting from a bad program run.

r!; b!; N1.r?; N1.b?; N1.ch!; N1.ch?; DEADLOCK

The helper thread of the first newsreader receives the Reuters and the Bloomberg news. However, only one of these messages will actually be read (consumed). This is the bug! Hence, the second newsreader gets stuck and we encounter a deadlock. The issue is that such a bad program run may rarely show up. So, the question is how can we assist the user based on the trace information resulting from a good program run? How can we infer that alternative schedules and communications may exist?