Summary

Concurrent datatypes are convenient tools for programmers. With concurrent datatypes, programmers can write code with multiple threads as if they are writing a single-threaded code. However, it is crucial to know the correctness of concurrent datatypes. In this thesis, we present CSP models for common concurrent primitive according to their behaviours. We then systematically build several concurrent datatypes from their Scala sources. We make assertions of these concurrent datatypes with a technique derived from linearizability testing. We find these assertions can effectively find bugs in a concurrent datatype and provide a history to give more context for the developer.

1 Introduction

Concurrent objects are convenient tools for programmers. With concurrent datatypes, programmers can write code with multiple threads as if they are writing a single-threaded code. However, it is crucial to know the correctness of concurrent objects. If the implementation of a concurrent object is wrong, then code using the concurrent object is very likely to be faulty.

The Channel object is one synchronization object commonly used in Go. The channel object can be used to share data from one process to another process. A process can send data to another process by calling send with the data to share. Likewise, a process can receive data from other processes by calling receive function. In Go and Communicating Scala Object package, the channels are unbuffered by default. If there is no process to receive the data, the sending process blocks until a process is willing to receive its data. Similarly, a receiving process blocks until a process sends some data.

```
trait Channel{
  def send(data: Int ): Int
  def receive (): Int
}
```

Figure 1: Interface of a MenWomen object

In this thesis, we shall study the correctness of synchronization objects. Each synchronization in a synchronization object involves multiple processes, whereas synchronization in concurrent datatypes like concurrent queue and concurrent only involves a single process.

There are two main properties to check for a synchronization object, the safety property and the liveness property. The safety property states that the history of the synchronization object should satisfy some conditions. For example, if one process sends 1 when no other process is sending, then a process calling receive should only receive 1. The liveness property states that the concurrent object should not refuse to synchronize when synchronization is possible between one or more processes. For example, if a process calls send and a process receive, the system should be able to synchronize and should not deadlock.

1.1 Thesis Overview

In the remaining part of Section 1, we describe the correctness condition for a synchronization object and abstractly how to test these conditions in CSP using the linearization test technique.

In section 2, we build CSP modules for common concurrent primitives such as shared variables, monitors and semaphores.

Starting from section 3, we use the linearization test technique to distinguish between correct and faulty implementation for several synchronization objects. We first implement the synchronization object in CSP according to its Scala source code. Then we write specifications for a system using the synchronization object and carry out the tests.

1.2 Synchronization linearizability test

To verify the correctness of a concurrent datatype, one can carry out the linearizability test described in the paper Testing for Linearizability [1]. The linearizability testing framework logs order of each function call and function return. Then for the observed history, the testing framework attempt to find a series of synchronization point that obeys the safety property. The concurrent datatype implementation is considered faulty if the framework can not find a valid synchronization point series.

In this remaining section, we shall look at a few examples of histories of systems using the `Channel` object. Figure 2 visualizes the history of a system with two processes. Process `T1` calls `send` with argument 1 and returns. Process `T2` calls `receive` and returns with 1. Each long horizontal line in the timeline represents a function call made by the corresponding process. The short vertical bars at the two ends of the long horizontal line indicate the function call's starting time and ending time. And the long vertical line between `T1` and `T2` represents the synchronization between the two processes.

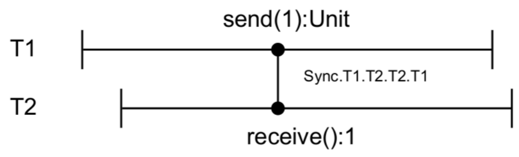

Figure 2: Visualized history of T1 calling send(1) and T2 calling receive()

Figure 3 shows a timeline similar to Figure 2, but `T2` returns 2 instead of 1. In this case, the linearizability test framework can not justify the return of processT2's `receive`, and suggests the trace is generated by a faulty channel implementation.

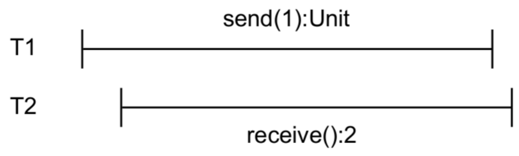

Figure 3: Visualized history of T1 sends 1 but T2 receives 2

In Figure 4, both processes calls `send`, and no synchronization is possible. Note that the liveness condition is not invalidated even if the system deadlocks in this case.

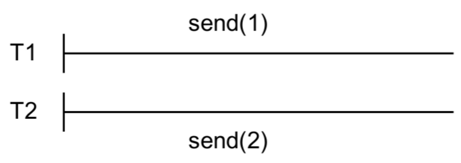

Figure 4: Visualized history of both T1 and T2 calling send

Scheduling is one of the reasons validating a history can be complicated. In Figure 5, process T3 calls send(3) first but gets descheduled. Then T1 calls send(1) and synchronizes with T2 which later calls receive. The linearization framework usually needs to search a large state to find a valid series of synchronization points.

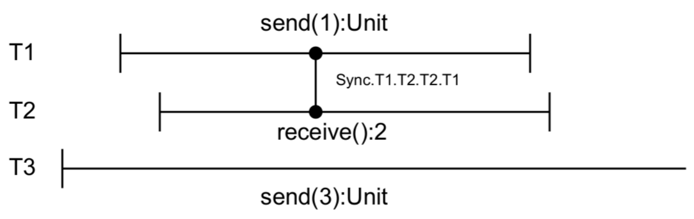

Figure 5: Visualized history of T3 get descheduled

1.3 Checking safety property using CSP

The history observed by the linearizability framework can be captured as a trace in CSP. A Call event in CSP represents the start of a function call in the observed history. A Return event represents the returning of a function call. For the safety property, we check that set of all possible histories of a testing system is a subset of all correct histories. In CSP, this corresponds to an assertion that a testing system trace refines a specification of systems using the synchronization object.

A generic and scalable system is used as the testing system. Each process in the testing system can call any function from the concurrent object with any arguments allowed. Each process must be allowed to terminate. Otherwise, the testing system only models a system that runs forever, given that there is no deadlock. We shall see how this affects bug finding in a concurrent datatype in later objects.

The specification process is constructed using the linearization technique. On the high level, the specification process for the system internally uses Sync events to represent synchronization between processes. Inside the specification process, some sub-processes generate corresponding Call and Return event for every synchronization point. When all sub-processes are placed in parallel, the Sync event agrees. So the resulting specification system generates all possible histories.

We shall see a concrete implementation of a testing system and a specification process in the MenWomen section.

1.4 Checking liveness property using CSP

For liveness property, we check the same generic and scalable testing system refines the same specification process, but in the failure model. One could use a datatype-specific specification process that does not explicitly use any synchronization points. However, reusing the linearizer process is easier.

1.5 Related work

Testing for Linearizability [1] presents a framework to test concurrent datatypes. However, because the testing framework uses observations of histories, it is unlikely to exhaust all possible histories of a system.

In Chapter 19 of Understanding Concurrent Systems [2], the author describes a CSP model for shared variables and provides a tool to analyze shared variable programs. But the tool lacks support for objects frequently used in concurrent programming, such as monitors and semaphores.

There are also runtime programming tools to detect race conditions and deadlocks in concurrent code. Thread Sanitizer [3] detects race conditions and deadlocks in C++ and Go.

2 Common Objects

2.1 Shared Variable

The usage of shared variables is common in concurrent datatypes. For example, some concurrent datatypes may temporarily store the identity of a waiting process. However, CSP is more like a functional programming language and does not support mutable variables.

A recursive process in CSP can capture the behaviour of a shared variable. The recursive process holds the value of the variable in its parameter. At any time, the variable process is willing to answer a query for the variable value in channel `getValue`. Alternatively, the process can receive an update on the variable value in channel `getValue`, after which the function recurses with the new variable value.

Because it is natural for a concurrent datatype to use multiple shared variables, the global variable is implemented as a CSP module in Figure 6 to allow better code reuse. The module requires two parameters. `TypeValue` is the set of possible values for the variable, and `initialValue` is the value before any process modifies the variable. An uninitialized variable module is also available in the same Figure 6, with the only difference that the variable non-deterministically chooses an initial value from `TypeValue` at start time. `runWith` is a convenient helper function to run a given process `P` with the `Var` process. If the parameter