Eric Crosson, Nhan Do, William Mauldin, and Daniel Officewala

Professor Vijay Garg

EE 360P

April 28, 2016

TODO: Add abstract

1 Introduction

Spin is a multi-threaded software verification tool that supports the high-level language Promela to specify models of systems [http://spinroot.com/spin/Man/Manual.html]. Given a model, Spin either performs random simulations of the system's execution or generates a C program that verifies the system's correctness properties, including deadlocks, unspecified receptions, and unexecutable blocks, as well as system invariants and other properties defined by the user. Promela allows the user to define processes, message channels, and variables.

2 Project Description

The intent of this project is to verify four models of distributed algorithms written in Promela. We chose to model the Dining Philosophers Algorithm [INSERT REFERENCE], the Token Ring Algorithm [INSERT REFERENCE], Szymanski's Mutual Exclusion Algorithm [INSERT REFERENCE], and Chandy and Lamport's Global Snapshot Algorithm [INSERT REFERENCE (Stormy # 1)]. We based our implementation heavily upon existing implementations of these algorithms, modified to suit our specific uses.

2.1 Dining Philosophers Algorithm

The Dining Philosophers Algorithm is a widely used algorithm for regulating resource sharing between multiple processes. It uses ”philosophers” and ”forks” as analogies for the processes and

resources. Lets say there are N philosophers, who only eat and think, sitting around a table. At the center of the table is a plate of spaghetti, and distributed between each philosopher are N forks. If a philosopher wants to eat, he/she must get two forks (the ones on either side of that particular philosopher). If one of the forks is currently in use by another philosopher, then he/she must wait until both forks are available before eating. Once a philosopher is done eating, he/she releases the forks so that other philosophers may acquire them to start their meal.

The algorithm to regulate this scenario includes an initial array of N boolean variables (`bool forks[N]`), representing the respective statuses of resources (`true` if the resource is available, `false` if not). If a process with with identifier `pid` wants to access a resource, it first checks if `forks[pid] == true`. If `true`, the status changes to `false` to indicate that the process acquired the resource. However, if `forks[pid] == false`, the process must wait until that resource becomes free. Then the process must obtain its second resource, namely `forks[(pid + 1) % N]`, repeating the same check as above.

In every group, there is a philosopher "kinder" than the rest. This philosopher behaves almost identical to his fellow philosophers except in one aspect: when he/she attempts to get this second fork, if the attempt fails, he/she releases the first fork instead of waiting for the second fork to become available. Then the philosopher goes back to thinking and tries again later.

2.2 Token Ring Algorithm

In the Token Ring Algorithm, a ring of processes is constructed, and each process is assigned a specific position in the ring. Each process knows which process is next in line. Upon initialization, a token is given to a specific process on the ring. This token is then passed indefinitely around the ring, from process k to process $k + 1$. When a process receives a token, it checks to see if it wants to enter the critical section. If so, the process enters the CS, does the work it needs to, and then exits the section. If not, the process simply passes the token along to the next process. The process is not allowed to re-enter the critical section using the same token.

Proving that the Token Ring satisfies mutual exclusion is simple: since a process can only

enter the CS if it has a token, and there is only one token, mutual exclusion must be upheld. This algorithm also satisfies Liveness because every process along the ring that wants to enter CS will have a chance to enter CS. It is important to note that the Token Ring is not a self-stabilizing system. In addition, the basic version of the algorithm is not fault tolerant. However, it is possible to construct a fault-tolerant version of the Token Ring algorithm by having processes send an acknowledgement upon receiving a token from a previous process.

2.3 Szymanski's Mutual Exclusion Algorithm

insert text here

2.4 Chandy and Lamport's Global Snapshot Algorithm

Chandy and Lamport's Global Snapshot Algorithm defines a method to take a global snapshot of a distributed system, which has proven to be a challenge due to the absence of shared memory and a shared clock [**TODO**: http://www.cfdvs.iitb.ac.in/projects/CourseProjs/Y2K2/naren.ps]. In this algorithm, the global snapshot is determined by collecting all process states (messages sent and received by a given process) and channel states (messages in transit). Processes record their own states and use markers (special types of messages) to deduce the channel states. All channels are assumed to be unidirectional and FIFO.

For any given process, for each of its outbound channels, the process sends a marker after recording its state and before sending any subsequent messages. When a marker is received by another process, if such a process has not recorded its own state, it does so with its current status, and designates the channel as empty (since this must have been the case for the message have been received prior to the local snapshot). Otherwise, the receiving process records its channel state as the sequence of messages it received after recording its state and prior to the reception of the marker. This recorded sequence of messages reflects the channel state.

The purpose of model-checking the Chandy-Lamport algorithm is to verify that each state

recorded is consistent, i.e.

$$\forall i, j : G[i]||G[j],$$

where G is a set of local states with exactly one local state from each process, $G[i]$ is the local state from process P_i, and $G[j]$ is the local state for process P_j [**TODO**: citation from Garg book].

2.5 Division of Labor

Each of the four team members was responsible for researching one algorithm, finding a suitable model, and re-implementing it for use within our directory structure and build system. Eric created the majority of the build system and set up Docker [INSERT REFERENCE] to allow for our research to be reproducible. William assisted with this process by helping to write some of the CMake files necessary to build the project. Eric focused his research on Szymanski's Algorithm, Daniel on the Token Ring Algorithm, William on Chandy and Lamport's Algorithm, and Nhan on the Dining Philosophers Algorithm.

3 Performance Results

3.1 Dining Philosophers Algorithm

A common problem that pops up in any discussion of Dining Philosopher problem/algorithm is the possibility of deadlock, when all the philosophers pick up a fork and forever wait on a someone (nonexistent) to finish with their other fork. In this case, the special philosopher circumvents the issue by giving up both fork if he/she cannot get both forks, which allows another philosopher to finish eating and surrender its fork for another philosopher to start eating. Spins demonstrates that this technique does in fact solve deadlock. Unfortunately, this algorithm does not prevent starvation. The philosopher that share the second fork with this special philosopher could in theory manage to get the fork before the special philosopher, and thus forever make that special philosopher surrender his/her forks.

3.2 Token Ring Algorithm

3.3 Szymanski's Mutual Exclusion Algorithm

3.4 Chandy and Lamport's Global Snapshot Algorithm

To verify Chandy and Lamport's Global Snapshot Algorithm, we used the Promela model created originally by Mordecai Ben-Ari in Principles of the Spin Model Checker [INSERT REFERENCE (Stormy #2)]. Since a global state is considered consistent if any two local states of two separate processes are effectively concurrent during that global state, the model simply consists of one sender and one receiver. All that is needed to verify a consistent state is to show that a the channel and process states fit the definition of consistent, i.e. that all messages sent prior to the marker are received, and that any messages sent after the marker are recorded as a part of the channel's state. We attempted to verify this using two test files: the first test's intent was to ensure that the This is checked by the two assert statements `assert (lastSent == messageAtMarker)` and `assert (messageAtRecord <= messageAtMarker)`. Although Ben-Ari suggested that having a channel size of 4 and a total message number of 6 would be sufficient in verifying the model, our build system allowed for us to input custom parameters if desired. The model as seen after being evaluated in iSpin is shown below. [Spin Book, pages 198-200]

4 Conclusions