Report: The Dining Philosopher SMV

Introduction

The Dining Philosopher is a classic problem illustration used in algorithm design to focus on resource allocation, synchronization, and concurrency. It presents a scenario where 5 philosophers seated around a table must share a limited number of utensils—traditionally forks but referred to as chopsticks here for clarity—and need both their left and right chopstick to eat their meal. Their left and right chopsticks however are shared with their adjacent philosophers. The challenge arises that each philosopher can be either hungry or thinking at any time and if they are hungry, they must eventually eat. In this report we outline two approaches in solving the problem: a sequential and concurrent model, both using NuSMV, a Symbolic Model Verifier (SMV).

phil.smv

The SMV source file called *phil.smv* is originally obtained as a prewritten faulty solution to be modified to assert and satisfy safety and liveness properties and to include fairness constraints. This solution will be sequential.

Adding SPECs and Constraints

The following SPECs were added to check the safety and liveness properties, respectively.

```
SPEC AG ((state = eating) -> gotleft & gotright)

SPEC AG ((state = hungry) -> AF (state = eating))
```

The safety SPEC asserts that a philosopher must have both chopsticks to eat. The liveness SPEC asserts that if a philosopher is hungry, they must eventually eat.

As well, the following fairness constraints are also added,

```
FAIRNESS running;

FAIRNESS state=hungry;
```

The first ensuring every philosopher instance will execute a transition and the second ensuring that a philosopher will eventually be hungry. These guarantee that philosophers will make some decision on their state and also at some point always transition to being hungry.

At this time, running *phil.smv* produces an output stating that the safety SPEC has passes but the liveness SPEC fails as some philosophers never get chopsticks while hungry and therefore never eat.

Token Solution

To satisfy liveness while maintaining safety, we implemented a sequential token system. This system allowes a single token to be passed around the table. Now, a philosopher can only pick up chopsticks if and only if they are hungry and have the token. The token is passed to the next seat if the philosopher holding the token is in the thinking or done state (i.e., finished eating or not yet hungry). The token location is represented by a number that corresponds to the respective philosopher holding it and initially is held by philosopher 0. This prevents the deadlock that happens when all philosophers pick up their left chopstick and therefore none can pick up their right.

phil.extended.smv

The SMV source file called *phil.extended.smv* contains the concurrent solution to the dining philosophers problem. As modified from the sequential solution (*phil.smv)* found above, this solution requires no strict sequencing along with the same need for satisfaction and assertion of safety and liveness and fairness constraints.

Adding SPECs and Constraints

In addition to the SPECs from *phil.smv*, more SPECs were added to assert concurrent eating and no strict sequencing among philosophers. The following SPECs assert no strict sequencing for philosopher 0, ensuring philosopher 0 (*p0*) can eat before philosophers 1, 2, 3, and 4.

```
SPEC EF (p0.state = eating & E [ p0.state = eating U (!(p0.state
= eating) & E [ !(p1.state = eating) U p0.state = eating ])])

SPEC EF (p0.state = eating & E [ p0.state = eating U (!(p0.state
= eating) & E [ !(p2.state = eating) U p0.state = eating ])])

SPEC EF (p0.state = eating & E [ p0.state = eating U (!(p0.state
= eating) & E [ !(p3.state = eating) U p0.state = eating ])])

SPEC EF (p0.state = eating & E [ p0.state = eating U (!(p0.state
= eating) & E [ !(p4.state = eating) U p0.state = eating ])])
```

The same SPECs were added for each philosopher.

As well, SPECs were added to assert concurrent eating among philosophers. The following asserts that philosophers who are not sitting beside each other must at some point eat at the same time.

```
SPEC EF ((p0.state=eating)&(p2.state=eating))

SPEC EF ((p0.state=eating)&(p3.state=eating))

SPEC EF ((p1.state=eating)&(p3.state=eating))

SPEC EF ((p1.state=eating)&(p4.state=eating))

SPEC EF ((p2.state=eating)&(p4.state=eating))
```

Modified Token Solution

The modified token solution involves the introduction of a second token. A philosopher now can pick up chopsticks if and only if they are holding either token. The philosopher will pass a token to the next seat when they are thinking or done and if the next seat isn't directly next to the other token. This modified solution allows concurrency with the addition of the second token and maintains the concurrency by not allowing the tokens to interfere with each other.