6.826 Principles of Computer Systems	Fall Semester, 1992

Lecture 9: October 14, 1992

Lecturer: Bill Weihl *Scribe: Susan Yeh*

1 Lecture Overview

Today's lecture continues discussion on concurrent modules. Topics covered include:

- Proving Concurrent Modules Correct
- Examples
- Safely versus Liveness

2 State Transitions in Concurrent Modules

The difficulty in mapping transitions in the implementation to transitions in the specification is that there is no longer a one-to-one relationship. The abstraction function for state transitions is as follows:

```
(AF: Tr(Implementation) -> SEQ[Tr(Spec)]
```

In general, however, each transition in the implementation maps to 0 or 1 transition in the specification. There are two types of transitions possible in the implementation: internal and external. External transitions are invocations and responses of external routines of the module. They map to exactly one external transition in the specification. Internal transitions, however, can not map to any external transition but can map to arbitrary sequences of internal transitions in the specification.

3 Example: FIFO Buffer

The specification and implementation of this module is in pp.9 of Handout 19. The proof can be found in pp. 2-4 of Handout 23.

3.1 Abstraction Function

To proof the module correct, we need to look at the entire state of the module, including program counters and local variables of the threads. To do the mapping, we enumerate all the program counters to find all possible thread states. The possible program counters are listed in Handout 23. Note that the statement

```
items := items++t;
```

actually contains two atomic actions.

Then, the strategy is to pick a transition in the implementation to map to the transition in the specification. In Produce, we choose the of update of `items` as that decisive transition; all other internal transitions in the implementation map to null transitions. Actually, some other transition in the implementation could be chosen as well, but in order to complete the proof, the abstraction function on states would need to be chosen differently too — probably something more contrived. (Recall that the proof requires the chosen transition to change the abstract state, and other transitions to leave the abstract state unchanged.) In general, it is preferable to choose a transition that admits a simple abstraction function on states.

3.2 Proof

To prove the implementation correct, we need some invariants. We use the standard inductive technique to prove these invariants true: show that each initial state of the implementation satisfies the invariants, and show that each atomic action in the implementation preserves the invariant. Then, to prove the abstraction function correct, we go through a similar inductive proof.

4 Example–Mutex

The specification and implementation of this module is in pp.6-7 of Handout 19. The proof is in pp.5-8 of Handout 23.

4.1 Abstraction Function

Here, there is a problem with mapping the states–the implementation does not keep track of which thread holds the mutex while the specification does. To handle this, we introduce a history variable `ms` and modify the code and the abstraction of the states slightly. Again, enumerate the possible program counters and define the abstraction function on the transitions.

4.2 Abstraction Function

To prove the abstraction function of the transitions correct, we do a case analysis of the transitions. We need to look at invocation, return, the body of Release, and the body of Acquire. To complete the proof, we need to introduce an invariant which is discussed and proven in Handout 23.

5 Safety versus Liveness

Most of the behavioral properties of a system can be divided into two classes: *safety* and *liveness*. A safety property specifies what the system is allowed to do, or equivalently, what it *must not do*.

Safety properties are often used to assert that the responses to invocations of routines be correct and that the abstract state of the system be one of the allowed ones.

Thus, partial correctness is an example of a safety property. Even absence of deadlock is a safety property, because deadlock is a function of the state of the system[1], and the property disallows the system to enter such a state. Another example of safety property is the worst-case behavior of the system; *e.g.*, that an invocation must respond within some specified time can be expressed as: the clock is not allowed to reach a certain value without the system having responded[2].

Liveness properties, on the other hand, specify what the system *must do*. They are often used to assert that something good must eventually happen. Thus, termination is a liveness requirement. So is absence of livelock. A livelock occurs when a number of contentious processes are active but none makes any real progress; *e.g.*, this can happen when two nodes try to send a packet on Ethernet and keep colliding. In general, the verification of liveness is difficult because one must consider infinite-sequences of system transitions. If the system admits one single invalid sequence, the property is violated — even if the probability of the occurrence of that sequence is infinitesimal.

It is possible, though difficult, to specify liveness requirements in Spec. For the rest of this course, we will largely ignore liveness and concentrate on safety issues.

[1]given a state, it is possible to deduce if there is a deadlock

[2]from the paper by L. Lamport on "A simple approach to Specifying Concurrent Systems," Handout 27