CSCI 234

Design of Internet Protocols:
Implementation Validation

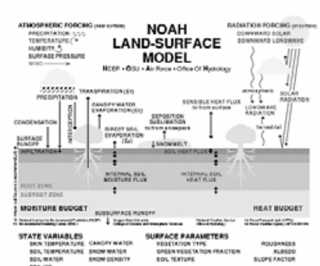
George Blankenship

Outline

- Validation verses Implementation
- Design validation
- System behavior
- Claims of correctness
- State analysis

Validation vs. Implementation

- Generality
- Validation Models



Design Validation: Correctness

- Absence of deadlock, livelock, no improper terminations
- A good design is ***provable*** free of deadlocks
- Verifying even the simplest of protocol properties, e.g., absence of deadlock, is PSPACE hard even for a finite state model

$$NL \subseteq P \subseteq NP \subseteq PSPACE$$

- Complexity can be attacked from two directions:
  - Using a relatively simple formalism for specifying correctness requirements
  - A method for reducing the complexity of models
- PROMELA is the formalism used.

Implementation Validation George Blankenship 4

Levels of Complexity

- Simple level (most frequently used requirements) e.g., absence of deadlock
  - Requirements expressed straightforwardly and checked independently
  - Can be analyzed mechanically with fast algorithms even for very large systems.
- More complicated requirements e.g., absence of livelock
  - Expressed independently
  - Independent computational expense when validated mechanically
  - Very sophisticated requirements, most expensive to check

Implementation Validation George Blankenship 5

Safety - Liveness Properties

Safety
- "nothing bad ever happens"
- example: system invariance
- ***x** is always less than **y***

- Model checker will search for any possible execution that leads to the violation of a safety property

Liveness
- "something good eventually happens"
- example: responsiveness
- *eventually a response is generated*

- Model checker will search for any possible execution in which the "good thing" can be postponed indefinitely

Implementation Validation George Blankenship 6

Reasoning about Behavior

- PROMELA models: number of possible behaviors is finite
- Two types of claims for behavior:
  - Inevitable
  - Impossible
- these two types of claims are duals
  - if something is inevitable then the opposite is impossible
  - if something is impossible then the opposite is inevitable
  - if we have a logic, we can turn one claim into another by logical negation

Implementation Validation George Blankenship 7

Behavior Types

- To state that a given behavior is inevitable, we state that all deviant behaviors are impossible
- An execution sequence is a finite ordered set of states
- A state in defined by the specification of all values, all control flow points of running processes, and the contents of message channels
- The behavior of a validation model is defined by the set of all execution sequences it can perform

Implementation Validation George Blankenship 8

Valid States

- A PROMELA model ***M*** with a finite ordered set of states is ***valid*** IFF *M* satisfies the following criteria
- First state of the sequence is the initial state of ***M*** with:
  - all variables initialized to zero
  - all message channels empty
  - only the *init* process active and set in its initial state
- If ***M*** is placed in the state with ordinal *i*, there is at least one executable statement that can bring it to the state with ordinal *i*+1

Implementation Validation George Blankenship 9

Sequence Types

- An execution sequence is ***terminating*** if:
  - no state occurs more than once in the sequence
  - model ***M*** contains no executable statements when placed in the last state of the sequence.
- An execution sequence is ***cyclic*** if:
  - all states except the last one are distinct, and the last state of the sequence is equal to one of the earlier states
- The union of all states included in the system behavior is called the *set of reachable* states of the model.

Implementation Validation George Blankenship 10

Claims of Correctness

- Model correctness claims can be built up from simple propositions
- A proposition is a Boolean condition on the state of the system

Implementation Validation George Blankenship 11

Syntax For Expressing Correctness Properties

- correctness properties
  - Reachable *states* (generic safety properties)
  - A *sequences* of states (generic liveness properties)
- (Promela)
  - **assertions**
    - local process assertions
    - system invariants
  - **end-state labels**
    - to define proper termination points of processes
  - **accept-state labels**
    - when looking for acceptance *cycles*
  - **progress-state labels**
    - when looking for *non*-progress *cycles*
  - **never claims**
  - **trace assertions**

Properties of states

Properties of sequence of states

Implementation Validation George Blankenship 12

Ordering of Propositions

- **Ordering of propositions different from ordering of statements**

In a proctype:

- A sequential ordering of two statements implies that the second statement is to be executed *after* the first one terminates.
- No assumptions about relative speeds of concurrently executing processes. So: *after* means …. ***eventually after***

In a temporal claim:

- A sequential ordering of two propositions defines an *immediate* consequence.
- An important requirement that applies to terminating sequences is absence of deadlock

Assertions

- Correctness criteria expressed as Boolean conditions that can be satisfied when a process reaches a given state

`assert(condition)`

- Always executable, if condition is true no effect!
- Validity is violated if there is at least one execution sequence in which the condition is false when the `assert` statement becomes executable.

Assertion Claim

- We try to claim that when process `A()` completes the value of `state` must be 2 and when process `B()` completes it must be 0
- Is the claim true or false?

```
byte state = 1;
proctype A() {
  (state == 1) ->
     state = state + 1;
  assert(state == 2)
  }
proctype B(){
  (state == 1) ->
     state = state - 1;
  assert(state == 0)
  }
init {run A(); run B()}
```

System Invariants

- Boolean conditions
- If true in initial system state remain true in *all* reachable states.

(independently of the execution sequence that leads to each specific state)

proctype monitor () { assert(invariant)

- Once an instance of monitor has been started, ***it executes independently of the rest of the system***; assert statement executable precisely once for every state of the system

Implementation Validation George Blankenship 16

Deadlocks

- In a finite state system, all execution sequences **either terminate** or they **cycle back** to a previously visited state.
- Terminating sequences are **not necessarily deadlocks**.
- Distinguish between *expected*, *proper*, end-states and unexpected ones.

Implementation Validation George Blankenship 17

Unexpected States

- Unexpected end-states include cases of incomplete protocol specification.
  - i.e.: the *unspecified reception*.
- The final state in a terminating execution sequence must minimally satisfy the following:
  - **Every process instantiated has terminated**
  - **All message channels are empty**

Implementation Validation George Blankenship 18

End-State

- Not all processes necessarily terminate.
- We must be able to identify individual process states as proper end-states.
- This is done with *end-state* labels:

```
proctype dijkstra()
{
end:  do
:: sema!p -> sema?v
od
}
```

- A state labeled `end` is a proper end-state
- Every process instantiated has either terminated or has reached a state marked as a proper end-state.

Implementation Validation George Blankenship 19

Bad Cycles

- Progress States
  - Finite sequence of states leading to an end state
  - Computation moving towards completion
- Non-progress States
  - Finite sequence of states not leading to end state
  - No apparent movement towards completion

Implementation Validation George Blankenship 20

Progress-States

- Marked states are called *progress-states*
- Based on explicit marking of states:
- There are no infinite behaviors of only unmarked states.
  - Execution sequences that violate the above correctness claim are called *non-progress cycles*
- There are no infinite behaviors that include marked states.
  - Execution sequences that violate the latter claim are called *livelocks.*

Implementation Validation George Blankenship 21

Non-Progress Cycles

- A progress-state label marks a state that must be executed for the protocol to make progress.
- Semaphore example: label the successful passing of a semaphore test as "progress"

```
proctype dijkstra() {
end: do
  :: sema!p ->
progress:  sema?v
  od
}
```

- By marking the progress state we express the correctness criteria that passing the semaphore guard cannot be postponed infinitely long.

Implementation Validation George Blankenship 22

Livelock

- An *acceptance-state label* is any label starting with the letter sequence "accept"

```
proctype dijkstra()
  {
  end:  do
  :: sema!p ->
  accept:       sema?v
  od
  }
```

- We claim we cannot cycle through a series of p and v operations. This claim is false.
- We can either prove it is false manually, or we can use an automated validator to provide a counter-example.

Implementation Validation George Blankenship 23

Temporal Claim

- Supposed we want to express the temporal claim:
  "*every state in which property P is true is followed by a state in which property Q is true.*"
- Two different interpretations of "*follows*" are possible:
  - **immediately follows**
  - **eventually follows**
- No assumption can be made in PROMELA about relative timing of process execution.
- Temporal claims define temporal ordering of *properties* of states.

Implementation Validation George Blankenship 24

Temporal Claim as Impossible

- *For every state in which property P is true is followed by a state in which property Q is true*
- We could write: P->Q
- Since all our correctness criteria are based on properties that are claimed to be *impossible*, the temporal claims we use must also express *ordering* of properties that are impossible.
- The temporal claims are defined on complete execution sequences. Even if a prefix of the sequence is irrelevant, it must still be represented as a trivially-true sequence of propositions.

Never is there a P without Q

- Thus `P->Q` can be expressed as:
  - `never { do :: skip od P->!Q }`

This is the format of a temporal claim in PROMELA

… independent of the initial sequence of events, it is impossible for a state in which property *P* is true to be followed by a state in which property *Q* is true.

Never Construct

- Suppose we want to express the temporal property that `condition1` can **never remain true** infinitely long:
- We must find where `condition1` may be:
  - a) false initially
  - b) becomes true eventually
  - c) remains true

```
never {
   do
   :: skip
   :: condition1 -> break
   od;
accept:      do
   :: condition1
   od
   }
```

condition1

- `skip` is always true.
- when `:: condition1 -> break` executes, it means `condition1` turns **true**.
- `condition1` in the second `do` stays true

```
never {
   do
   :: skip
   :: condition1 -> break
   od;
accept:
   condition1
}
```

Implementation Validation George Blankenship 28

P->Q FSM

- The claim contains just one more state transition after `condition1` becomes true.
- The claim is matched if there is at least one execution sequence in which `condition1` holds in two subsequent states.
- A claim can be seen as a finite state machine.
- The finite state machine above contains three states:
  - the initial state
  - the state labeled `accept`
  - normal end state

```
never {
   do
   :: skip
   :: condition1 -> break
   od;
accept:
   condition1
}
```

Implementation Validation George Blankenship 29

Livelock

- The correct version of the claim states that it would be an error (a livelock) if the machine can stay in the second state infinitely long.
- The second version is that it would be an error if the third state is reachable.

Implementation Validation George Blankenship 30

Never claims

- Never claims in combination with acceptance-state labels can express also the absence of non-progress cycles.
- The complexity of finding non-progress cycles directly with progress-state labels is smaller than the expense of the validation of a claim that specifies the same property.

Implementation Validation George Blankenship 31

Message Loss

- Modeled explicitly with clauses that can steal an incoming message before it is processed.
- Claim: "*it is always true that when the sender transmits a message, the receiver will eventually accept it.*"
- The claim is a four-state machine:
  - **the initial state**
  - the **two states that were labeled**
  - the **normal end state**

Implementation Validation George Blankenship 32

System State Recognition

```
never {
do
    :: len(receiver) == 0
    :: receiver?[msg0] ->
      goto accept0
    :: receiver?[msg1] ->
      goto accept1
od;
accept0:
do
    :: !Receiver[2]:P0
od;
accept1:
do
    :: !Receiver[2]:P1
od;
}
```

- At least one of the three conditions must be true in the initial system state.
- The claim remains in this state as long as `receiver` is empty.
- If `receiver` contains `msg0` or `msg1`, it will change state to either `accept0` or `accept1`.
- Once a transition is made, the claim can only remain in that state if the receiver process will never accept a message with the same sequence number.

Implementation Validation George Blankenship 33

Process State Recognition

```
never {
    do
        :: len(receiver) == 0
        :: receiver?[msg0] -> goto
            accept0
        :: receiver?[msg1] -> goto
            accept1
    od;

    accept0:
    do
        :: !Receiver[2]:P0
    od;

    accept1:
    do
        :: !Receiver[2]:P1
    od;
    }
```

The receiver never passes the state labeled `P0.`

- The `pid` of init is 0, Sender is 1, and Receiver is 2.
- `Receiver[2]` refers to the receiver process.
- `Receiver[2]:P0` refers to that Receiver is currently in state labels `P0.`

Implementation Validation George Blankenship 34

Exercise

a) How many reachable states do you predict will the following naive *Promela* model generate?
b) Will the simulation terminate?
c) Estimate the total number of reachable states that should be inspected in an exhaustive verification. Is it a finite number? Will a verification run terminate?
d) What would happen if you had declared the variable to be a short instead of a byte ?

```
init {
byte i = 0;
do :: i = i+1
od
}
```

Implementation Validation George Blankenship 35