AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Test of Distributed Systems
Lecture 08

Verifying system properties

The Round-Robin token example

Example: Neilsen-Mizuno

Today's lecture

- **Verifying system properties w. Spin**
- The Round-Robin token example
- Example: Neilsen-Mizuno
- Next time

Verifying (absence of) deadlock w. Spin

- Deadlock
  - Run a Spin verification in “Safety” mode
  - If the model ends up in a state where all processes are blocked and one or more processes are blocked at a “non-end” statement, Spin will report a deadlock
  - So no reason to do anything special, perhaps apart from marking valid end-statements
  - Let’s see small example ... (safety-deadlock.pml)

Verifying safety properties w. Spin (1)

- Suppose we want to verify a safety property, e.g mutual exclusion, and we have defined the predicate

```
#define mutex (critical <= 1)
```

- We make this safety specification `[]mutex` and want Spin to verify it for us
- You now enter a competition with Spin – you provide the model and the safety claim, and Spin tries to prove you wrong

Verifying safety properties w. Spin (2)

- To prove your safety claim wrong, Spin needs to provide just a single counter-example (find a single state where your claim fails)
- The claim []mutex is converted to a ***never claim***, i.e. a claim that can never be true if your model is correct
- The never claim represents the negation of your safety claim: from “mutex always true” to “mutex false at least once”

Verifying safety properties w. Spin (3)

- Your claim []mutex is transformed into this:

```
never { /* !([]mutex) */
T0_init:
  if
  :: (! ((mutex))) -> goto accept_all
  :: (1) -> goto T0_init
  fi;
accept_all:
  skip
}
```

If we ever get here, i.e. the never claim terminates before our program, we have lost

Verifying safety properties w. Spin (4)

- Spin now plays the “let me prove you wrong game”
- After initialization the first round goes to Spin (because your claim may fail immediately) and the never claim gets to execute one atomic statement
- Then it’s your model’s turn – Spin forwards your model’s state by one atomic statement
- And so it goes, one atomic statement at a time: never-claim, your model, never-claim, your model ... etc.

Verifying safety properties w. Spin (5)

- The “program” that ends first wins – your model or the never claim
- If Spin ends its expansion/search of your models’s state space before the never-claim ends up in an acceptance state, you win – otherwise you lose
- To perform the verification, run Spin in “Safety” mode against your (LTL) claim

Verifying liveness properties w. Spin (1)

- Suppose we want to verify a liveness property, e.g **<>csp**
- To prove your liveness claim wrong, Spin needs to provide a counter-example in the form af an infinite sequence of states (a cycle) in which your claim is false. Again your claim <>csp is converted to a ***never claim***, i.e. a claim that can never be true if your model is correct
- The never claim represents the negation of your liveness claim: from “csp eventually becomes true” to “csp is never true”

Verifying liveness properties w. Spin (2)

- The never claim for <>csp is:

```
never { /* !(<>csp) */
accept_init:
T0_init:
  if
  :: (! ((csp))) -> goto T0_init
  fi;
}
```

A label starting with ‘accept’ marks the start of a search for a cycle – if a state is seen twice during such a search we loose

If csp becomes true, the never claim blocks (i.e. it can’t terminate), and we win

Verifying liveness properties w. Spin (3)

- To prove you wrong, Spin searches for a sequence of states in which !csp is true
- If csp ever becomes true, the never claim is blocked and you win
- But if Spin finds a state it has seen earlier, and !csp has been false so far (the never claim hasn't blocked yet), you loose because Spin has found an ***acceptance cycle***
- Spin marks the sequence from the start of the cycle to the point where the cycle was found

Verifying liveness properties w. Spin (4)

- To test a liveness property, run Spin in "Acceptance" mode against your (LTL) claim

Today's lecture

- Verifying system properties
- **The Round-Robin token example**
- Example: Neilsen-Mizuno
- Next time

Looking at an example

- Let's see an example, a Spin model of the Round-Robin token passing algorithm from last time
  - Absence of deadlock
  - Mutual exclusion
  - Absence of starvation

Today's lecture

- Verifying system properties
- The Round-Robin token example
- **Example: Neilsen-Mizuno**
- Next time

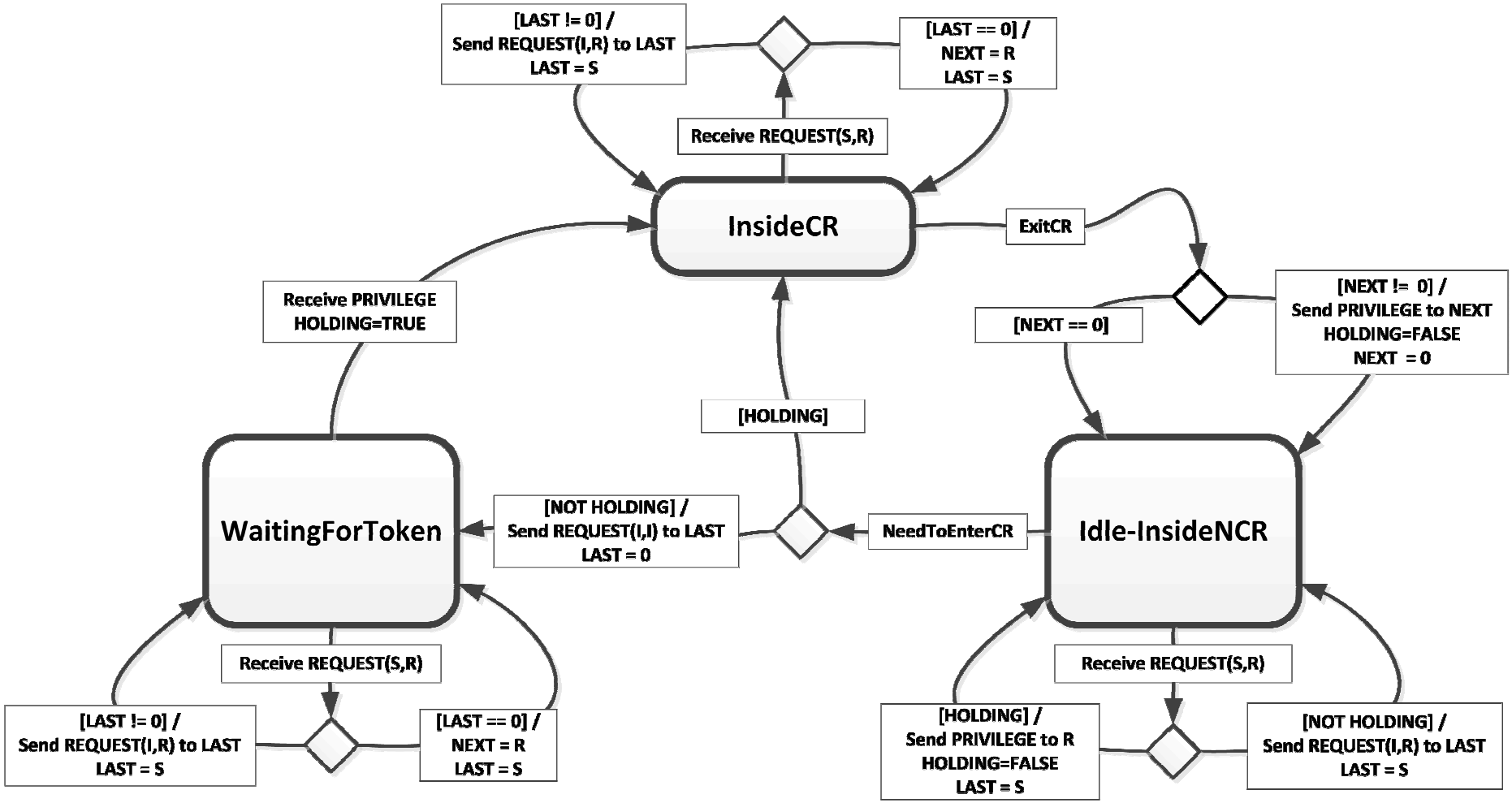
Neilsen-Mizuno (1)

- Just an example of an algorithm for mutual exclusion in DS's
- Provides an impression of the difficulties of describing and proving the properties of an DS algorithm

Neilsen-Mizuno (2)

- The key points of the algorithm
  - Nodes fully connected, but placed in a fixed DAG
  - Each node remembers where it last sent the token and forwards requests in that direction
  - A process that wants to enter its CR sends a request in the direction where the token (or forwarded requests) were sent
  - A node that receives a request while waiting for the token, remembers the id of the requesting node (builds a “waiting list”)

Neilsen-Mizuno (3)



Neilsen-Mizuno (4)

- Neilsen-Mizuno provides (sketches of) proof for
  - Mutual exclusion
  - Freedom of deadlock and starvation
- Hard! But we are not concerned with the proofs ...
- The performance analysis in interesting as an example, though ...

Neilsen-Mizuno (5)

- Neilsen-Mizuno provides an analysis of performance in terms of
  - The number of messages that must be exchanged (upper bound (N), is on average $(3-(5/N)+2/N^2)$
  - Storage overhead (3 ints per node, 2 ints per msg)
  - Synchronization delay (1 to (N+1) msgs)

Today's lecture

- Verifying system properties
- The Round-Robin token example
- Example: Neilsen-Mizuno
- **Next time**

Next time

- Exercise on Monday (perhaps)
  - Implement (BACI) and model (Spin) “I want the token broadcast” token passing for mutual exclusion
  - Add (design, implement, verify) features that remove problems with starvation
- The Chandy-Lamport snapshot algorithm
- Reading material
  - Section13 in the paper "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms"