The PlusCal Language

• We will now focus on the PlusCal language.

• This is translated into TLA+.

• It’s more important to understand how this translates into state machines.

• The notion of labels and atomicity is crucial for us.

variables x = 5, y = 0;

begin
  Line1: x := x + 1;
  Line2: y := 2 * x
variables $x = 5, y = 0$;

begin
  start: $x := x + 1$;
  mid: $y := 2 \times x$

Labels $\rightarrow$ States

What happens if we remove the second label?
variables \( x = 5, y = 0; \)

begin
    start: \( x := x + 1; \)
    mid: \( y := 2 \times x \)

The two assignments are executed atomically!
(This is critical for modeling concurrent processes)
Conditionals

What if we now remove the label “mid2”?

variables $x = 5, y = 10$

begin
start: if $x > y$ then
    mid1: $y := y + 1$
else
    mid2: $x := x + 1$
end if;
variables \( x = 5, y = 10; \)

begin
start: if \( x > y \) then
    mid1: \( y := y + 1; \)
else
    mid2: \( x := x + 1; \)
end if;

**Conditionals**

Yes, this is an atomic test and set!
Example 1

(Practical TLA: Chapter 5)
The Reader Writer Pattern

---

**MODULE** server

**EXTENDS** TLC, Integers, Sequences

**CONSTANTS** MaxQueueSize

(*--algorithm message_queue

**variable** queue = <<<>>;

**define**
  BoundedQueue == Len(queue) <= MaxQueueSize
**end define**;

**process** writer = "writer"
begin Write:
  while TRUE do
    queue := Append(queue, "msg");
  end while;
end process;

**process** reader = "reader"
variables current_message = "none"
begin Read:
  while TRUE do
    current_message := Head(queue);
    queue := Tail(queue);
  end while;
end process;

**end algorithm;*)

---

To be explained!
MODULE server

EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>

variable <variables>;

define
   <predicates>
end define;
end define;

<body>

end algorithm;*)

=============================================================================
--- MODULE server ---

EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>

variable <variables>;

define
  <predicates>
end define;
end algorithm;*)

=============================================================================
EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>

variable <variables>;

define
   <predicates>
end define;
end define;

<body>

end algorithm;*)

=============================================================================
Constants allow us to configure the model, but remain constant throughout simulation.

Variables (like program variables) do change throughout the simulation of the model.

**Structure**

EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>
variable <variables>;
define
  <predicates>
end define;
end algorithm;*)
--- MODULE server ---

EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>

variable <variables>;

define
  <predicates>
end define;

<body>

end algorithm;*)

Here we can define predicates (conditions) that we want to check.
--- MODULE server ---

EXTENDS <imports>
CONSTANTS <constants>

(*--algorithm <name>

variable <variables>;

define
  <predicates>
end define;

<body>

end algorithm;*)

And this is where we define the processes.

Structure
What will we model?

- Two asynchronous processes that communicate.
- They share a bounded queue.
- We want to make sure our concurrent design does not lead to a situation where the queue is overfilled!
- This is a simple example, but we’ll make some deliberate mistakes to understand the basic concepts and what kind of bugs we can detect with this tool.
The Invariant

We define a predicate called BoundedQueue. This condition should hold at every step: invariant!
The Queue Operations

• We use the Sequence library for this.

• A sequence: <<1, 2, 3>>

• We can append:
  Append(4, <<1, 2, 3>>) == <<1, 2, 3, 4>>

• And decompose to remove an element:
  Head(<<1, 2, 3>>) == 1
  Tail(<<1, 2, 3>>) == <<2, 3>>
process writer = "writer"
begin Write:
    while TRUE do
        queue := Append(queue, "msg");
    end while;
end process;
The writer process is given the identifier “writer”. Yes, this may look weird, but we could have more than one writer process: 

```
writer \in \{“John”, “Mary”\}
```

```plaintext
process writer = "writer"
begin
  Write:
    while TRUE do
      queue := Append(queue, "msg");
    end while;
end process;
```
process writer = "writer"
begin Write:
    while TRUE do
        queue := Append(queue, "msg");
    end while;
end process;

This is the initial label, it’s also the head of the loop. This does not make the entire loop atomic…

The Writer Process
process writer = "writer"
begin
  Write:
    queue := Append(queue, "msg");
  goto Write
end process;

The Writer Process

We could have written the while-loop like this.
process writer = "writer"
begin Write:
    while TRUE do
        queue := Append(queue, "msg");
    end while;
end process;

Anyway, so this basically means the writer just keeps adding to the queue whenever it has something to say.
MODULE server ---------------------------------------------

EXTENDS TLC, Integers, Sequences
CONSTANTS MaxQueueSize

(*--algorithm message_queue

variable queue = <<>>;

define
    BoundedQueue == Len(queue) <= MaxQueueSize
end define;

process writer = "writer"
begins Write:
    while TRUE do
        queue := Append(queue, "msg");
    end while;
end process;

process reader = "reader"
variables current_message = "none"
begins Read:
    while TRUE do
        current_message := Head(queue);
        queue := Tail(queue);
    end while;
end process;

end algorithm;*)
Demo

And of course it fails…
Fix the reader

Await (or when) adds an enabling condition!
This was \([x > y]\) in the Statechart syntax.

```plaintext
process reader = "reader"
variables current_message = "none";
begin Read:
    while TRUE do
        await queue /= <<>>;
        current_message := Head(queue);
        queue := Tail(queue);
    end while;
end process;
```
Transcript of Lecture

• No we ran it again…

• This time model checking does not terminate!

• Obviously, the writer can still just keep adding stuff, but we should be checking that the queue does not exceed the limit?

• Oops… We didn’t actually add the invariant to the model, so currently the state space is infinite and we really aren’t checking anything.
Demo

Now we add Invariant “BoundedQueue”
Demo

And now we see that the unfettered writer can overstep the queue bound. Astonishing! :)

TLA+ Toolbox display showing a model named Model_1 with an invariant stating that BoundedQueue is violated.
We fix the writer
And now model checking succeeds!

```plaintext
process writer = "writer"
begin
  Write:
    await Len(queue) < MaxQueueSize;
    queue := Append(queue, "msg");
  goto Write
end process;
```
So what’s the gain?

• Sure, these were not hard bugs to find… but we’ll get to slightly subtler bugs.

• But even here it’s nice that we can ask questions about the design and test things out very easily.

• What if the readers want to write back? Can they use the same channel?
process reader = "reader"
variables current_message = "none";
begin
  Read:
    while TRUE do
      current_message := Head(queue);
      queue := Tail(queue);
      either
        skip;
      or
        NotifyFailure:
          await Len(queue) < MaxQueueSize;
          queue := Append(queue, "fails");
      end either;
    end while;
end process;

Reader answers
Either is just nondeterministic “if”
process writer = "writer"
begin
    Write:
        await Len(queue) < MaxQueueSize;
        queue := Append(queue, "msg");
goto Write
end process;

process reader = "reader"
variables current_message = "none"
begin Read:
    while TRUE do
        await queue /= <<>>;
        current_message := Head(queue);
        queue := Tail(queue);
        either
            skip;
        or
            NotifyFailure:
                await Len(queue) < MaxQueueSize;
                queue := Append(queue, "fails");
        end either;
    end while;
end process;

What can go wrong?
We check for the bounds everywhere, so this should be safe, right?
Deadlock!

So if the writer fills the queue after a read, but before the reader wants to respond, they will both be waiting...
Desperation Breeds

• A deadlock is a situation where no process has any enabled transitions!

• This happened because both are waiting for someone else to empty the queue...

• One desperate attempt to fix this would be to maybe add another reader who could step up to rescue us.

• Will this fix it?
Multiple readers!

No, the model-checker still finds a deadlock. It’s still nice that we can test these things by just changing a few lines.
Example 2

(Practical TLA: Chapter 5)
MODULE cache

EXTENDS Integers, Sequences, TLC

CONSTANT ResourceCap, MaxConsumerReq

ASSUME ResourceCap > 0
ASSUME MaxConsumerReq \in 1..ResourceCap

(*--algorithm cache
variables resources_left = ResourceCap;

define ResourceInvariant == resources_left >= 0 end define;

process actor = "actor"
variables resources_needed \in 1..MaxConsumerReq;
begin UseResources:
  while TRUE do
    resources_left := resources_left - resources_needed;
  end while;
end process;

end algorithm; *)
So what’s new?

Assumptions are restrictions on our constants.
MODULE cache

EXTENDS Integers, Sequences, TLC
CONSTANT ResourceCap, MaxConsumerReq

ASSUME ResourceCap > 0
ASSUME MaxConsumerReq \in 1..ResourceCap

(*--algorithm cache
variables resources_left = ResourceCap;

define ResourceInvariant == resources_left >= 0 end define;

process actor = "actor"
variables resources_needed \in 1..MaxConsumerReq;
begin UseResources:
  ...
end process;

end algorithm; *)
What are we modeling?

• Basically, consumption of a shared critical resource…

• Consumers request resources and consume them. They may consume up to \( \text{MaxConsumerReq} \) each time.

• The resource is occasionally replenished to the \( \text{ResourceCap} \).

• Again, we need to add “await” to pass the invariant.

    \[
    \text{await resources_left} \geq \text{resources_needed};
    \]
The replenishing code

Without this, we deadlock when out of resources.
process actor \in \{"actor1", "actor2"\}
variables resources_needed \in 1..MaxConsumerReq
begin
  WaitForResources:
    while TRUE do
      await
        resources_left >= resources_needed;
        resources_left := resources_left - resources_needed;
    end while;
end process;

But Now...
Let’s consider two actors: is it safe?
process actor \in \{"actor1", "actor2"\} 
variables resources_needed \in 1..MaxConsumerReq 
begin WaitForResources:
  while TRUE do
    await resources_left \geq resources_needed;
    UseResources:
      while resources_needed > 0 do
        resources_left := resources_left - 1;
        resources_needed := resources_needed - 1;
      end while;
      with x \in 1..MaxConsumerReq do
        resources_needed := x;
      end with;
  end while;
end process;

Of course it was...
But what if they use resources concurrently?
What could possibly go wrong?
## Race Condition

<table>
<thead>
<tr>
<th>PC</th>
<th>Actor 1 need</th>
<th>Actor 2 need</th>
<th>Resources left</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A2 Check</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A2 Use</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A1 Check</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A1 Use</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A2 Use</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A1 Use</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
### Race Condition

<table>
<thead>
<tr>
<th>PC</th>
<th>Actor 1 need</th>
<th>Actor 2 need</th>
<th>Resources left</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>A2 Check</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>A2 Use</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A1 Check</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A1 Use</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A2 Use</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A1 Use</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

This check passes, although A2 still plans to eat one.
The fix

- Let’s try to fix it by reserving the amount needed.
- And then just reset this at each time tick.
- This should finally work, right?
Of course not!

Who's stupid idea was it to reset at each replenish?
process actor \in \{"actor1", "actor2"\}
variables resources_needed \in 1..MaxConsumerReq
begin
  WaitForResources:
    while TRUE do
      await reserved + resources_needed <= resources_left;
      reserved := reserved + resources_needed;
  UseResources:
    while resources_needed > 0 do
      resources_left := resources_left - 1;
      resources_needed := resources_needed - 1;
      reserved := reserved - 1;
    end while;
  with x \in 1..MaxConsumerReq do
    resources_needed := x;
  end with;
end while;
end process;

No resetting!
Instead decrease reserved at each consumption.
Summary

• We encountered the main problems that TLA can discover in our designs.

• Primarily we check all possible states of the model for invariant violations.

• Invariant violations due to race conditions are particularly hard to spot without help from TLA.

• Deadlocks are another hard-to-find concurrency problem when none of our processes can make progress.