Applications of Formal Verification

Model Checking: Introduction to Spin

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2012
SPIN: Previous Lecture vs. This Lecture

Previous lecture
SPIN appeared as a PROMELA simulator

This lecture
Intro to SPIN as a model checker
What Does A Model Checker Do?

A Model Checker (MC) is designed to prove the user wrong.

MC tries its best to *find a counter example* to the correctness properties.
It is tuned for that.

MC does not try to prove correctness properties.
It tries the opposite.
A Model Checker (MC) is designed to prove the user wrong.

MC tries its best to *find a counter example* to the correctness properties. It is tuned for that.

MC does not try to prove correctness properties. It tries the opposite.

But why then can a MC also *prove* correctness properties?
What Does A Model Checker Do?

A Model Checker (MC) is designed to prove the user wrong.

MC tries its best to *find a counter example* to the correctness properties. It is tuned for that.

MC does not try to prove correctness properties. It tries the opposite.

But why then can a MC also *prove* correctness properties?

MC’s *search* for counter examples is *exhaustive.*
What Does A Model Checker Do?

A Model Checker (MC) is designed to prove the user wrong.

MC tries its best to find a counter example to the correctness properties.
It is tuned for that.

MC does not try to prove correctness properties.
It tries the opposite.

But why then can a MC also prove correctness properties?

MC’s search for counter examples is exhaustive.

⇒ Finding no counter example proves stated correctness properties.
What does ‘exhaustive search’ mean here?

exhaustive search

\[ = \]

_resolving non-determinism in all possible ways_
exhaustive search

\[=\]

resolving non-determinism in all possible ways

For model checking PROMELA code, 

two kinds of non-determinism to be resolved:

- **explicit, local:**
  - if/do statements
    - :: guardX \rightarrow …. 
    - :: guardY \rightarrow …. 

- **implicit, global:**
  - scheduling of concurrent processes
  (see next lecture)
Model Checker for This Course: **SPIN**

**SPIN**: “Simple Promela Interpreter”
Model Checker for This Course: SPIN

SPIN: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!
Model Checker for This Course: **SPIN**

**SPIN**: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of **SPIN**:

- simulating a model (randomly/interactively)
- generating a *verifier*
Model Checker for This Course: **SPIN**

**SPIN**: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of **SPIN**:
- simulating a model (randomly/interactively)
- generating a *verifier*

*verifier* generated by **SPIN** is a C program performing
Model Checker for This Course: \texttt{SPIN}

\texttt{SPIN}: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of \texttt{SPIN}:
- simulating a model (randomly/interactively)
- generating a verifier

\texttt{verifier} generated by \texttt{SPIN} is a C program performing \textit{model checking}:
Model Checker for This Course: **SPIN**

**SPIN**: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of **SPIN**:
- simulating a model (randomly/interactively)
- generating a *verifier*

*verifier* generated by **SPIN** is a C program performing *model checking*:
- exhaustively *checks* PROMELA *model* against correctness properties
**Model Checker for This Course: SPIN**

SPIN: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of SPIN:
- simulating a model (randomly/interactively)
- generating a verifier

**verifier** generated by SPIN is a C program performing **model checking**:
- exhaustively checks PROMELA model against correctness properties
- in case the check is negative: generates a failing run of the model
Model Checker for This Course: **SPIN**

**SPIN**: “Simple Promela Interpreter”

If this was all, you would have seen most of it already. The name is a serious understatement!

Main functionality of **SPIN**:
- simulating a model (randomly/interactively/guided)
- generating a verifier

**verifier** generated by **SPIN** is a C program performing **model checking**:
- exhaustively checks PROMELA *model* against correctness properties
- in case the check is negative:
  generates a failing run of the model, *to be simulated by SPIN*
SPIN Workflow: Overview

- model \textit{name.pml}
- correctness properties

- SPIN
  - \texttt{-a}
  - \texttt{-i}
  - \texttt{-t}
  - random/interactive/guided simulation

- verifier \texttt{pan.c}

- C compiler

- executable verifier \texttt{pan}

- failing run \texttt{name.pml.trail}
  - “errors: 0”

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification

SS 2012 6/34
Plain Simulation with SPIN

- model name.pml
- correctness properties

- SPIN

- verifier pan.c

- C compiler

- executable verifier pan

random/ interactive/ guided simulation

- i

- failing run name.pml.trail

"errors: 0"
Rehearsal: Simulation Demo

- run example, random and interactive
  `interleave.pml, zero.pml`
Model Checking with SPIN

- model name.pml
- correctness properties

SPIN

-verifier pan.c

C compiler

-executable verifier pan

either

failing run name.pml.trail

"errors: 0"

random/ interactive/ guided simulation
Given PROMELA model $M$, and correctness properties $C_1, \ldots, C_n$.

- Be $R_M$ the set of all possible runs of $M$.
- For each correctness property $C_i$, $R_{M,C_i}$ is the set of all runs of $M$ satisfying $C_i$. ($R_{M,C_i} \subseteq R_M$)
- $M$ is correct wrt. $C_1, \ldots, C_n$ iff $(R_{M,C_1} \cap \ldots \cap R_{M,C_n}) = R_M$.
- If $M$ is not correct, then each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.
Meaning of Correctness wrt. Properties

Given PROMELA model $M$, and correctness properties $C_1, \ldots, C_n$.

- Be $R_M$ the set of all possible runs of $M$.
- For each correctness property $C_i$, $R_{M,C_i}$ is the set of all runs of $M$ satisfying $C_i$.
  \[ R_{M,C_i} \subseteq R_M \]
- $M$ is correct wrt. $C_1, \ldots, C_n$ iff \[(R_{M,C_1} \cap \ldots \cap R_{M,C_n}) = R_M.\]
- If $M$ is not correct, then each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

We know how to write models $M$. 
Meaning of Correctness wrt. Properties

Given PROMELA model $M$, and correctness properties $C_1, \ldots, C_n$.

- Be $R_M$ the set of all possible runs of $M$.
- For each correctness property $C_i$, $R_{M,C_i}$ is the set of all runs of $M$ satisfying $C_i$.
  \[(R_{M,C_i} \subseteq R_M)\]
- $M$ is correct wrt. $C_1, \ldots, C_n$ iff \((R_{M,C_1} \cap \ldots \cap R_{M,C_n}) = R_M\).
- If $M$ is not correct, then each $r \in (R_M \setminus (R_{M,C_1} \cap \ldots \cap R_{M,C_n}))$ is a counter example.

We know how to write models $M$.
But how to write Correctness Properties?
Correctness properties can be stated (syntactically) within or outside the model. Stating properties within the model, using assertion statements meta labels end labels accept labels progress labels stating properties outside the model, using never claims temporal logic formulas.
Stating Correctness Properties

Correctness properties can be stated (syntactically) \textit{within} or \textit{outside} the model.
Correctness properties can be stated (syntactically) **within** or **outside** the model.

- stating properties within the model, using
  - assertion statements
Stating Correctness Properties

Correctness properties can be stated (syntactically) within or outside the model.

stating properties within the model, using
- assertion statements
- meta labels
  - end labels
  - accept labels
  - progress labels
Correctness properties can be stated (syntactically) within or outside the model.

Stating properties within the model, using
- assertion statements
- meta labels
  - end labels
  - accept labels
  - progress labels

Stating properties outside the model, using
- never claims
- temporal logic formulas
Stating Correctness Properties

Correctness properties can be stated (syntactically) within or outside the model.

- Stating properties within the model, using
  - *assertion statements* (today)
  - *meta labels*
    - *end labels* (today)
    - *accept labels*
    - *progress labels*

- Stating properties outside the model, using
  - *never claims*
  - *temporal logic formulas*
Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

\[
\text{assert} \ (\text{expr})
\]

were \( \text{expr} \) is any PROMELA expression.
Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

```plaintext
assert(expr)
```

were `expr` is any PROMELA expression.

Typically, `expr` is of type `bool`.
Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form

\[ \text{assert}(expr) \]

were \( expr \) is any PROMELA expression.

Typically, \( expr \) is of type \texttt{bool}.

Assertion statements can appear anywhere where a PROMELA statement is expected.
Assertion statements in PROMELA are statements of the form

```
assert(expr)
```

were `expr` is any PROMELA expression.

Typically, `expr` is of type `bool`.

Assertion statements can appear anywhere where a PROMELA statement is expected.

```plaintext
... stmt1;
assert(max == a);
stmt2;
...
```
Assertion Statements

Definition (Assertion Statements)

Assertion statements in PROMELA are statements of the form
\[ \text{assert} \ (\text{expr}) \]
were \( \text{expr} \) is any PROMELA expression.

Typically, \( \text{expr} \) is of type \texttt{bool}.

Assertion statements can appear anywhere where a PROMELA statement is expected.

```pseudocode
... stmt1;
assert (max == a);
stmt2;
...
... if
:: b1 -> stmt3;
  assert (x < y)
:: b2 -> stmt4
...```

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification

SS 2012 12/34
Meaning of **Boolean Assertion Statements**

```plaintext
assert (expr)
- has no effect if `expr` evaluates to true
- triggers an error message if `expr` evaluates to false
```

This holds in both, simulation and model checking mode.
assert \( (expr) \)

- has no effect if \( expr \) evaluates to non-zero value
- triggers an error message if \( expr \) evaluates to 0

This holds in both, simulation and model checking mode.
Meaning of General Assertion Statements

\texttt{assert} (\textit{expr})

- has no effect if \textit{expr} evaluates to non-zero value
- triggers an error message if \textit{expr} evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

\texttt{bool true false} is syntactic sugar for
**Meaning of General Assertion Statements**

`assert(expr)`

- has no effect if `expr` evaluates to non-zero value
- triggers an error message if `expr` evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

```
bool true false  is syntactic sugar for
bit  1  0
```
Meaning of General Assertion Statements

assert\((expr)\)

- has no effect if \(expr\) evaluates to non-zero value
- triggers an error message if \(expr\) evaluates to 0

This holds in both, simulation and model checking mode.

Recall:

```
bool true false  is syntactic sugar for
bit 1 0
```

\(\Rightarrow\) general case covers Boolean case
Instead of using ‘printf’s for Debugging ...

/* after choosing a,b from {1,2,3} */
if
    :: a >= b -> max = a;
    :: a <= b -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", a, b, max);
Instead of using ‘printf’ s for Debugging ...

/* after choosing a, b from {1, 2, 3} */
if
  :: a >= b -> max = a;
  :: a <= b -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", 
a, b, max);

Command Line Execution
(simulate, inject faults, add assertion, simulate again)

> spin max.pml
... we can employ **Assertions**

quoting from file **max.pml**:

```pml
/* after choosing a,b from {1,2,3} */
if
    :: a >= b -> max = a;
    :: a <= b -> max = b;
fi;
assert( a > b -> max == a : max == b )
```

Now, we have a first example with a formulated correctness property. We can do model checking, for the first time!
... we can employ Assertions

quoting from file `max.pml`:

```pml
/* after choosing a, b from {1, 2, 3} */
if
  :: a >= b -> max = a;
  :: a <= b -> max = b;
fi;
assert ( a > b -> max == a : max == b )
```

Now, we have a first example with a formulated correctness property.
... we can employ **Assertions**

quoting from file `max.pml`:

```plaintext
/* after choosing a,b from {1,2,3} */
if
    :: a >= b -> max = a;
    :: a <= b -> max = b;
fi;
assert( a > b -> max == a : max == b )
```

Now, we have a first example with a formulated **correctness property**.

We can do **model checking**, for the first time!
Generate Verifier in C

```
> spin -a max.pml
```

SPIN generates Verifier in C, called `pan.c`

(plus helper files)
Compile To Executable Verifier

Command Line Execution

```
compile to executable verifier

> gcc -o pan pan.c
```
Compile To Executable Verifier

Command Line Execution

`gcc -o pan pan.c`

C compiler generates **executable verifier** `pan`
Compile To Executable Verifier

Command Line Execution

cOMPLETE TO EXECUTABLE VERIFIER

> gcc -o pan pan.c

C compiler generates executable verifier pan

pan: historically “protocol analyzer”, now “process analyzer”
Run Verifier (= Model Check)

run verifier pan

"errors: 0"

either

or

failing run
max.pml.trail

Command Line Execution

run verifier pan

> ./pan
Run Verifier (= Model Check)

Command Line Execution

```
run verifier pan
> ./pan
```

- prints "errors: 0"
Run Verifier (= Model Check)

**Command Line Execution**

```
run verifier pan
> ./pan
```

- prints "errors: 0"  ⇒ Correctness Property verified!
Run Verifier (= Model Check)

executable
verifier
pan

either
“errors: 0”
or
failing run
max.pml.trail

Command Line Execution

run verifier pan
> ./pan

- prints “errors: 0”, or
- prints “errors: n” (n > 0)
Run Verifier (= Model Check)

Command Line Execution

```
run verifier pan
> ./pan
```

- prints "errors: 0", or
- prints "errors: n" \((n > 0)\) \(\Rightarrow\) counter example found!
Run Verifier (= Model Check)

**Executable Verifier**

```
run verifier pan
```

Either

- "errors: 0"

or

```
> ./pan
```

prints "errors: 0", or

prints "errors: n" (n > 0)  \(\Rightarrow\) counter example found!

records failing run in `max.pml.trail`

Command Line Execution
Guided Simulation

To examine failing run: employ simulation mode, “guided” by trail file.

SPIN

Command Line Execution

inject a fault, re-run verification, and then:

> spin -t -p -l max.pml
Output of Guided Simulation

can look like:

Starting P with pid 0
1: proc 0 (P) line 8 "max.pml" (state 1) [a = 1]
   P(0):a = 1
2: proc 0 (P) line 14 "max.pml" (state 7) [b = 2]
   P(0):b = 2
3: proc 0 (P) line 23 "max.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max.pml" (state 14) [max = a]
   P(0):max = 1
spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
   assert(((a>b)) -> ((max==a)) : ((max==b)))
Output of Guided Simulation

can look like:

Starting P with pid 0
1: proc 0 (P) line 8 "max.pml" (state 1) [a = 1]
   P(0): a = 1
2: proc 0 (P) line 14 "max.pml" (state 7) [b = 2]
   P(0): b = 2
3: proc 0 (P) line 23 "max.pml" (state 13) [((a<=b))]
3: proc 0 (P) line 23 "max.pml" (state 14) [max = a]
   P(0): max = 1
spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
   assert(( ((a>b)) -> ((max==a)) : ((max==b)) ))

assignments in the run
can look like:

Starting P with pid 0
1: proc 0 (P) line 8 "max.pml" (state 1) \[a = 1\]
   \[P(0):a = 1\]
2: proc 0 (P) line 14 "max.pml" (state 7) \[b = 2\]
   \[P(0):b = 2\]
3: proc 0 (P) line 23 "max.pml" (state 13) \[((a<=b))\]
3: proc 0 (P) line 23 "max.pml" (state 14) \[max = a\]
   \[P(0):max = 1\]

spin: line 25 "max.pml", Error: assertion violated
spin: text of failed assertion:
   \texttt{assert}(( ((a>b)) -> ((max==a)) : ((max==b)) ))

assignments in the run
values of variables whenever updated
What did we do so far?
following whole cycle (most primitive example, assertions only)

model name.pml

correctness properties

verifier pan.c

C compiler

executable verifier pan

random/ interactive / guided simulation

verifier

failing run name.pml.trail

"errors: 0"
What did we do so far?
following whole cycle (most primitive example, assertions only)

- model `name.pml`
- correctness properties

```
SPIN
```

- verifier `pan.c`
- C compiler
- `pan`

```
errors: 0
```

```
random/interactive/guided simulation
-p -l -g ...
```

```
verifier pan.c
```

```
C compiler
```

```
executable verifier pan
```

```
running name.pml.trail
```

```
“errors: 0”
```

either

or
Further Examples: Integer Division

```c
int dividend = 15;
int divisor = 4;
int quotient, remainder;

quotient = 0;
remainder = dividend;
do
  :: remainder > divisor ->
    quotient++;
    remainder = remainder - divisor
  :: else ->
    break
od;
printf("%d divided by %d = %d, remainder = %d\n",
    dividend, divisor, quotient, remainder);
```
Further Examples: Integer Division

```c
int dividend = 15;
int divisor = 4;
int quotient, remainder;

quotient = 0;
remainder = dividend;
do
    :: remainder > divisor ->
        quotient++;
        remainder = remainder - divisor
    :: else ->
        break
od;
printf("%d divided by %d = %d, remainder = %d\n", dividend, divisor, quotient, remainder);
```
simulate, put assertions, verify, change values, ...
Further Examples: Greatest Common Divisor

```c
int x = 15, y = 20;
int a, b;
a = x; b = y;
do
      :: a > b -> a = a - b
      :: b > a -> b = b - a
      :: a == b -> break
od;
printf("The GCD of %d and %d = %d\n", x, y, a)
```

Further Examples: Greatest Common Divisor

```c
int x = 15, y = 20;
int a, b;
a = x; b = y;
do
    :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
od;
printf("The GCD of %d and %d = %d\n", x, y, a)
```

full functional verification not possible here (why?)
Further Examples: Greatest Common Divisor

\begin{verbatim}
int x = 15, y = 20;
int a, b;
a = x; b = y;
do
    :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
od;
printf("The GCD of %d and %d = %d\n", x, y, a)
\end{verbatim}

full functional verification not possible here (why?)

still, assertions can perform \textit{sanity check}
Further Examples: Greatest Common Divisor

```c
int x = 15, y = 20;
int a, b;
a = x; b = y;
do :: a > b -> a = a - b
    :: b > a -> b = b - a
    :: a == b -> break
od;
printf("The GCD of %d and %d = %d\n", x, y, a)
```

full functional verification not possible here (why?)
still, assertions can perform sanity check
⇒ typical for model checking
Typical Command Lines

typical command line sequences:
random simulation

\texttt{spin \textit{name.pml}}

Typical Command Lines

typical command line sequences:
random simulation
   spin name.pml
interactive simulation
   spin -i name.pml
typical command line sequences:

random simulation  
  spin name.pml

interactive simulation  
  spin -i name.pml

model checking  
  spin -a name.pml  
  gcc -o pan pan.c  
  ./pan
typical command line sequences:
random simulation
   spin name.pml
interactive simulation
   spin -i name.pml
model checking
   spin -a name.pml
gcc -o pan pan.c
   ./pan
and in case of error
   spin -t -p -l -g name.pml
Ben-Ari produced Spin Reference Card, summarizing

- typical command line sequences
- options for
  - **SPIN**
  - gcc
  - pan
- **PROMELA**
  - datatypes
  - operators
  - statements
  - guarded commands
  - processes
  - channels
- temporal logic syntax
Why SPIN?

- SPIN targets software, instead of hardware verification.
- Based on standard theory of $\omega$-automata and linear temporal logic.
- 2001 ACM Software Systems Award (other winning software systems include: Unix, TCP/IP, WWW, Tcl/Tk, Java).
- Used for safety critical applications.
- Distributed freely as research tool, well-documented, actively maintained, large user-base in academia and in industry.
- Annual SPIN user workshops series held since 1995.
Why SPIN? (Cont’d)

- PROMELA and SPIN are rather simple to use
- good to understand a few system really well, rather than many systems poorly
- availability of good course book (Ben-Ari)
- availability of front end JSPIN (also Ben-Ari)
What is JSpin?

- graphical user interface for SPIN
- developed for pedagogical purposes
- written in Java
- simple user interface
- SPIN options automatically supplied
- fully configurable
- supports graphics output of transition system
What is JSPIN?

- graphical user interface for SPIN
- developed for pedagogical purposes
- written in Java
- simple user interface
- SPIN options automatically supplied
- fully configurable
- supports graphics output of transition system
- makes back-end calls transparent
**Calling JSPIN**

```bash
> java -jar /usr/local/jSpin/jSpin.jar
```

*(with path adjusted to your setting)*

**or use shell script:**

```bash
> jspin
```
command line execution

**Calling JSpin**

> java -jar /usr/local/jSpin/jSpin.jar

*with path adjusted to your setting*

or use shell script:

> jspin

play around with similar examples ...
Catching A Different Type of Error

quoting from file max2.pml:

/* after choosing a,b from {1,2,3} */
if
    :: a >= b -> max = a;
    :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", a, b, max);
Catching A Different Type of Error

quoting from file max2.pml:

```plaintext
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a;
  :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", a, b, max);
```

simulate a few times

⇒ crazy "timeout" message sometimes
⇒ generates "errors: 1"
Catching A Different Type of Error

quoting from file `max2.pml`:

```c
/* after choosing a,b from {1,2,3} */
if
   :: a >= b -> max = a;
   :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n",
       a, b, max);
```

simulate a few times
⇒ crazy “timeout” message sometimes
Catching A Different Type of Error

quoting from file max2.pml:

```pml
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a;
  :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n",
       a, b, max);
```

simulate a few times
⇒ crazy “timeout” message sometimes

generate and execute pan
Catching A Different Type of Error

quoting from file \texttt{max2.pml}:

\begin{verbatim}
/* after choosing \(a, b\) from \{1, 2, 3\} */
if
  :: a \geq b \rightarrow \text{max} = a;
  :: b \leq a \rightarrow \text{max} = b;
fi;
\text{printf}("the maximum of \%d and \%d is \%d\n",
          a, b, max);
\end{verbatim}

simulate a few times
\(\Rightarrow\) crazy \texttt{"timeout"} message sometimes

generate and execute \texttt{pan}
\(\Rightarrow\) reports \texttt{"errors: 1"}
Catching A Different Type of Error

quoting from file max2.pml:

```c
/* after choosing a, b from {1,2,3} */
if
  :: a >= b -> max = a;
  :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", a, b, max);
```

simulate a few times
⇒ crazy “timeout” message sometimes

generate and execute pan
⇒ reports “errors: 1”

????
Catching A Different Type of Error

quoting from file max2.pml:

```pml
/* after choosing a,b from {1,2,3} */
if
  :: a >= b -> max = a;
  :: b <= a -> max = b;
fi;
printf("the maximum of %d and %d is %d\n", a, b, max);
```

simulate a few times
⇒ crazy "timeout" message sometimes

generate and execute pan
⇒ reports "errors: 1"

Note: no assert in max2.pml.
Catching A Different Type of Error

Further inspection of `pan` output:

```plaintext
...
pan: invalid end state (at depth 1)
pan: wrote max2.pml.trail
...
```
A process may legally block, **as long as some other process can proceed.**
Legal and Illegal Blocking

A process may legally block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical, for concurrent and distributed models (i.p. protocols).
Legal and Illegal Blocking

A process may legally block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical, for concurrent and distributed models (i.p. protocols).

But

it's an error if a process blocks while no other process can proceed.
Legal and Illegal Blocking

A process may legally block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical, for concurrent and distributed models (i.e., protocols).

But

it’s an error if a process blocks while no other process can proceed

⇒ “Deadlock”
A process may legally block, as long as some other process can proceed.

Blocking for letting others proceed is useful, and typical, for concurrent and distributed models (i.p. protocols).

But

it’s an error if a process blocks while no other process can proceed

⇒ “Deadlock”

in max1.pml, no process can take over.
Valid End States

Definition (Valid End State)

An end state of a run is valid iff the location counter of each processes is at an end location.
Valid End States

Definition (Valid End State)
An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)
End locations of a process $P$ are:
- $P$’s textual end
**Valid End States**

<table>
<thead>
<tr>
<th>Definition (Valid End State)</th>
</tr>
</thead>
<tbody>
<tr>
<td>An <strong>end state</strong> of a <strong>run</strong> is valid iff the location counter of each processes is at an <strong>end location</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (End Location)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>End locations</strong> of a process $P$ are:</td>
</tr>
<tr>
<td>- $P$’s textual end</td>
</tr>
<tr>
<td>- each location marked with an <strong>end label</strong>: “endxxx:”</td>
</tr>
</tbody>
</table>
Valid End States

Definition (Valid End State)
An end state of a run is valid iff the location counter of each processes is at an end location.

Definition (End Location)
End locations of a process \( P \) are:
- \( P \)'s textual end
- each location marked with an end label: “endxxx:”

End labels are not useful in max1.pml, but elsewhere, they are. Example: end.pml
Literature for this Lecture

Ben-Ari Chapter 2, Sections 4.7.1, 4.7.2