Formal Specification and Verification
Model Checking with Temporal Logic

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg
Model Checking with \textit{SPIN}

- Model: name.pml
- Correctness properties

\textbf{SPIN} \rightarrow \textbf{verifier pan.c} \rightarrow \textbf{C compiler} \rightarrow \textbf{executable verifier pan}

- Either:
  - Failing run
  - "errors: 0"

Formal Specification and Verification: Model Checking
Correctness properties can be stated within, or outside, the model.
Correctness properties can be stated within, or outside, the model.

**stating properties within model**, using

- assertion statements
Stating Correctness Properties

Correctness properties can be stated within, or outside, the model.

**stating properties within model**, using

- assertion statements
- meta labels
  - end labels
  - accept labels
  - progress labels
Stating Correctness Properties

Correctness properties can be stated within, or outside, the model.

**stating properties within model**, using

- assertion statements
- meta labels
  - end labels
  - accept labels
  - progress labels

**stating properties outside model**, using

- never claims
- temporal logic formulas
Correctness properties can be stated within, or outside, the model.

**stating properties within model**, using
- assertion statements
- meta labels
  - end labels
  - accept labels
  - progress labels

**stating properties outside model**, using
- never claims
- **temporal logic formulas** (today’s main topic)
Correctness properties can be stated within, or outside, the model.

**stating properties within model**, using
- assertion statements
- meta labels
  - end labels
  - accept labels (briefly)
  - progress labels

**stating properties outside model**, using
- never claims (briefly)
- temporal logic formulas (today’s main topic)
Preliminaries

- fairness
- accept labels, accepting runs
Preliminaries I: Fairness

Does the following PROMELA model necessarily terminate?

```promela
byte n = 0;
bool flag = false;

active proctype P() {
    do :: flag -> break;
    :: else -> n = 5 - n;
    od
}
active proctype Q() {
    flag = true
}
```

Termination guaranteed only if scheduling is (weakly) fair!

Definition (Weak Fairness)
A run is called weakly fair iff the following holds:
- each continuously executable statement is executed eventually.
Preliminaries I: Fairness

Does the following PROMELA model necessarily terminate?

```promela
byte n = 0;
bool flag = false;

active proctype P() {
    do :: flag -> break;
    :: else -> n = 5 - n;
    od
}
active proctype Q() {
    flag = true
}
```

Termination guaranteed only if scheduling is (weakly) fair!
Does the following Promela model necessarily terminate?

```
byte n = 0;
bool flag = false;

active proctype P() {
    do :: flag -> break;
    :: else -> n = 5 - n;
    od
}
active proctype Q() {
    flag = true
}
```

Termination guaranteed only if scheduling is (weakly) fair!

**Definition (Weak Fairness)**

A run is called weakly fair iff the following holds:
each continuously executable statement is executed eventually.
Definition (Accept Location)

A location marked with an accept label of the form “accept xxx:” is called an accept location.
Definition (Accept Location)

A location marked with an accept label of the form “accept xxx:” is called an accept location.

Accept locations can be used to specify cyclic behavior.
Preliminaries II: Acceptance Cycles

Definition (Accept Location)
A location marked with an accept label of the form “acceptxxx:” is called an accept location.

Accept locations can be used to specify cyclic behavior.

Definition (Acceptance Cycle)
A run which infinitely often passes through an accept location is called an acceptance cycle.
Definition (Accept Location)

A location marked with an accept label of the form “acceptxxx:” is called an accept location.

Accept locations can be used to specify cyclic behavior.

Definition (Acceptance Cycle)

A run which infinitely often passes through an accept location is called an acceptance cycle.

Acceptance cycles are mainly used in ‘never claims’ (see below), to define forbidden behavior of infinite kind.
many correctness properties not expressible by assertions
many correctness properties not expressible by assertions
today:

model checking of properties formulated in temporal logic
many correctness properties not expressible by assertions

today:

model checking of properties formulated in temporal logic

Remark:
in this course, “temporal logic” is synonymous to “linear temporal logic” (LTL)
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

critical ++;
assert (critical <= 1);
critical --;

Drawbacks:

▶ no separation of concerns (model vs. correctness property)
▶ changing assertions is error prone (easily out of synch)
▶ easy to forget assertions: correctness property might be violated at unexpected locations
▶ many interesting properties not expressible via assertions
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

```c
critical++; assert(critical <= 1); critical--;```

Drawbacks:

- no separation of concerns (model vs. correctness property)
- changing assertions is error prone (easily out of synch)
- easy to forget assertions: correctness property might be violated at unexpected locations
- many interesting properties not expressible via assertions
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

```c
    critical++;
    assert( critical <= 1 );
    critical--;
```

Drawbacks:

- no separation of concerns (model vs. correctness property)
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

```c
critical++; assert( critical <= 1 ); critical--;```

Drawbacks:

▶ no separation of concerns (model vs. correctness property)
▶ changing assertions is error prone (easily out of synch)
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

```c

critical++; assert( critical <= 1 ); critical--;
```

Drawbacks:

- no separation of concerns (model vs. correctness property)
- changing assertions is error prone (easily out of synch)
- easy to forget assertions: correctness property might be violated at unexpected locations
Beyond Assertions

Assertions only talk about the state ‘at their own location’ in the code.

Example: mutual exclusion expressed by adding assertion into each critical section.

```c
critical++; assert( critical <= 1 ); critical--;```

Drawbacks:

- no separation of concerns (model vs. correctness property)
- changing assertions is error prone (easily out of synch)
- easy to forget assertions:
  correctness property might be violated at unexpected locations
- many interesting properties not expressible via assertions
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

Mutual Exclusion

\[ \text{critical} \leq 1 \]

holds throughout the run.

Array Index within Bounds

\[ 0 \leq i \leq \text{len} - 1 \]

holds throughout the run.

Absence of Deadlock

If some processes try to enter their critical section, eventually one of them does so.

Absence of Starvation

If one process tries to enter its critical section, eventually that process does so.

all these are temporal properties ⇒ use temporal logic
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical <= 1 holds throughout the run”
properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical <= 1 holds throughout the run”

**Array Index within Bounds** (given array \( a \) of length \( len \))

“0 <= i <= len-1 holds throughout the run”
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical <= 1 holds throughout the run”

**Array Index within Bounds** (given array a of length len)

“0 <= i <= len-1 holds throughout the run”

properties impossible to express via assertions:
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical <= 1 holds throughout the run”

**Array Index within Bounds** (given array a of length len)

“0 <= i <= len-1 holds throughout the run”

properties impossible to express via assertions:

**Absence of Deadlock**

“If some processes try to enter their critical section, eventually one of them does so.”
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical <= 1 holds throughout the run”

**Array Index within Bounds** (given array `a` of length `len`)

“0 <= i <= len-1 holds throughout the run”

properties impossible to express via assertions:

**Absence of Deadlock**

“If some processes try to enter their critical section, eventually one of them does so.”

**Absence of Starvation**

“If one process tries to enter its critical section, eventually that process does so.”
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical \( \leq 1 \) holds throughout the run”

**Array Index within Bounds** (given array \( a \) of length \( \text{len} \))

“\( 0 \leq i \leq \text{len}-1 \) holds throughout the run”

properties impossible to express via assertions:

**Absence of Deadlock**

“If some processes try to enter their critical section, eventually one of them does so.”

**Absence of Starvation**

“If one process tries to enter its critical section, eventually that process does so.”

all these are temporal properties
Temporal Correctness Properties

properties more conveniently expressed as global properties, rather than assertions:

**Mutual Exclusion**

“critical \(\leq 1\) holds throughout the run”

**Array Index within Bounds** (given array \(a\) of length \(len\))

“\(0 \leq i \leq len-1\) holds throughout the run”

properties impossible to express via assertions:

**Absence of Deadlock**

“If some processes try to enter their critical section, eventually one of them does so.”

**Absence of Starvation**

“If one process tries to enter its critical section, eventually that process does so.”

all these are temporal properties \(\Rightarrow\) use temporal logic
talking about numerical variables (like in critical $\leq 1$ or $0 \leq i \leq \text{len}-1$) requires variation of *propositional temporal logic* which we call **Boolean temporal logic**:

- **Boolean expressions** (over PROMELA variables), rather than *propositions*, form basic building blocks of the logic
Set $For_{BTL}$ of Boolean Temporal Formulas (simplified)

- all PROMELA variables and constants of type bool/bit are $\in For_{BTL}$
Set $For_{BTL}$ of Boolean Temporal Formulas (simplified)

- all Promela variables and constants of type bool/bit are $\in For_{BTL}$
- if $e_1$ and $e_2$ are numerical Promela expressions, then all of $e_1==e_2, e_1!=e_2, e_1<e_2, e_1<=e_2, e_1>e_2, e_1>=e_2$ are $\in For_{BTL}$
Set $For_{BTL}$ of **Boolean Temporal Formulas** (simplified)

- all **Promela** variables and constants of type **bool/bit** are $\in For_{BTL}$
- if $e_1$ and $e_2$ are numerical **Promela** expressions, then all of $e_1==e_2$, $e_1!=e_2$, $e_1<e_2$, $e_1<=e_2$, $e_1>e_2$, $e_1>=e_2$ are $\in For_{BTL}$
- if $P$ is a process and $l$ is a label in $P$, then $P@l$ is $\in For_{BTL}$
  ($P@l$ reads “$P$ is at $l$”)
### Set $\text{For}_{BTL}$ of Boolean Temporal Formulas (simplified)

- All **Promela** variables and constants of type `bool`/`bit` are in $\text{For}_{BTL}$
- If $e_1$ and $e_2$ are numerical **Promela** expressions, then all of $e_1==e_2$, $e_1!=e_2$, $e_1<e_2$, $e_1<=e_2$, $e_1>e_2$, $e_1>e_2$ are in $\text{For}_{BTL}$
- If $P$ is a process and $l$ is a label in $P$, then $P@l$ is in $\text{For}_{BTL}$ ($P@l$ reads “$P$ is at $l$”)
- If $\phi$ and $\psi$ are formulas in $\text{For}_{BTL}$, then all of $\neg \phi$, $\phi \land \psi$, $\phi || \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$, $[]\phi$, $<>\phi$, $\phi U \psi$ are in $\text{For}_{BTL}$
A run $\sigma$ through a Promela model $M$ is a chain of states

$L_0, I_0 \rightarrow L_1, I_1 \rightarrow L_2, I_2 \rightarrow L_3, I_3 \rightarrow L_4, I_4 \rightarrow \ldots$

$L_j$ maps each running process to its current location counter. From $L_j$ to $L_{j+1}$, only one of the location counters has advanced (exception: channel rendezvous).

$I_j$ maps each variable in $M$ to its current value.
A run $\sigma$ through a Promela model $M$ is a chain of states $L_0, I_0 \rightarrow L_1, I_1 \rightarrow L_2, I_2 \rightarrow L_3, I_3 \rightarrow L_4, I_4 \rightarrow \ldots$

$L_j$ maps each running process to its current location counter. From $L_j$ to $L_{j+1}$, only one of the location counters has advanced (exception: channel rendezvous).

$I_j$ maps each variable in $M$ to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g. $L_j, I_j \models x < y$ iff $I_j(x) < I_j(y)$
A run $\sigma$ through a Promela model $M$ is a chain of states $L_0, I_0 \rightarrow L_1, I_1 \rightarrow L_2, I_2 \rightarrow L_3, I_3 \rightarrow L_4, I_4 \rightarrow \ldots$

$L_j$ maps each running process to its current location counter. From $L_j$ to $L_{j+1}$, only one of the location counters has advanced (exception: channel rendezvous).

$I_j$ maps each variable in $M$ to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g. $L_j, I_j \models x < y$ iff $I_j(x) < I_j(y)$

$L_j, I_j \models P@1$ iff $L_j(P)$ is the location labeled with 1
A run $\sigma$ through a Promela model $M$ is a chain of states $L_0, I_0 \rightarrow L_1, I_1 \rightarrow L_2, I_2 \rightarrow L_3, I_3 \rightarrow L_4, I_4 \rightarrow \ldots$

$L_j$ maps each running process to its current location counter. From $L_j$ to $L_{j+1}$, only one of the location counters has advanced (exception: channel rendezvous).

$I_j$ maps each variable in $M$ to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g. $L_j, I_j \models x < y$ iff $I_j(x) < I_j(y)$

$L_j, I_j \models P@1$ iff $L_j(P)$ is the location labeled with 1

Evaluating other formulas $\in For_{BTL}$ in runs $\sigma$: see lecture 2.
Spin supports Boolean temporal logic
Spin supports Boolean temporal logic

but
**Spin** supports Boolean temporal logic

**but**

arithmetic operators (+, -, *, /, ...),
relational operators (==, !=, <, <=, ...),
label operators (@)
cannot appear directly in TL formulas given to **Spin**
**Boolean Temporal Logic Support in SPIN**

SPIN supports Boolean temporal logic

but

arithmetic operators (+, -, *, /, ...),
relational operators (==, !=, <, <=, ...),
label operators (@)

cannot appear directly in TL formulas given to SPIN

instead

Boolean expressions must be abbreviated using `#define`
formulas of the form \(\square \phi\) are called safety properties
formulas of the form $[\phi]$ are called safety properties

state that something good, $\phi$, is guaranteed throughout each run

example: $[](\text{critical} \leq 1)$

"it is guaranteed throughout each run that at most one process visits its critical section"

or equivalently: "more than one process visiting its critical section will never happen"
formulas of the form $[]\phi$ are called safety properties

state that something good, $\phi$, is guaranteed throughout each run

special case:

$[]\neg\psi$ states that something bad, $\psi$, never happens
Safety Properties

formulas of the form $\Box \phi$ are called safety properties

state that something good, $\phi$, is guaranteed throughout each run

special case:

$\Box \neg \psi$ states that something bad, $\psi$, never happens

example: ‘$\Box (\text{critical} \leq 1)$’
formulas of the form \([\square] \phi\) are called **safety properties**

state that something good, \(\phi\), is **guaranteed throughout** each run

special case:
\([\square] \neg \psi\) states that something bad, \(\psi\), **never happens**

example: ‘\([\square](\text{critical} \leq 1)\)’

“it is **guaranteed throughout** each run that at most one process visits its critical section”
formulas of the form $[\phi]$ are called safety properties

state that something good, $\phi$, is guaranteed throughout each run

special case:

$[\neg \psi]$ states that something bad, $\psi$, never happens

example: ‘$[\text{critical} \leq 1]$’

“it is guaranteed throughout each run that at most one process visits its critical section”

or equivalently:

“more than one process visiting its critical section will never happen”
We want to verify ‘[](critical<=1)’ as correctness property of:

```c
active proc type P() {
    do :: /* non-critical activity */
    atomic {
        !inCriticalQ;
        inCriticalP = true
    }
    critical++;
    /* critical activity */
    critical--;
    inCriticalP = false
}

/* similarly for process Q */
```
1. add ‘#define mutex (critical <= 1)’ to Promela file
2. open Promela file
3. enter []mutex in LTL text field
4. select Translate to create a ‘never claim’, corresponding to the negation of the formula
5. ensure Safety is selected
6. select Verify
7. (if necessary) select Stop to terminate too long verification
you may ignore them, but if you are interested:

- a never claim tries to show the user wrong
- it defines, in terms of Promela, all violations of a wanted correctness property
- it is semantically equivalent to the negation of the wanted correctness property
- jSpin adds the negation for you
- using Spin directly, you have to add the negation yourself
Model Checking a Safety Property with Spin directly

Command Line Execution

make sure ‘#define mutex (critical <= 1)’ is in safety1.pml

> spin -a -f "!([] mutex)" safety1.pml
> gcc -DSAFETY -o pan pan.c
> ./pan
formulas of the form $<>\phi$ are called liveness properties
formulas of the form $<>\phi$ are called \textit{liveness properties} state that something good, $\phi$, \textit{eventually happens} in each run
Liveness Properties

Formulas of the form $<>\phi$ are called liveness properties. They state that something good, $\phi$, eventually happens in each run.

Example: $<>csp$

(with $csp$ a variable only true in the critical section of $P$)
Liveness Properties

formulas of the form $<>\phi$ are called **liveness properties**
state that something good, $\phi$, **eventually happens** in each run

example: ‘$<>csp$’
(with $csp$ a variable only true in the critical section of $P$)

“in each run, process $P$ visits its critical section **eventually**”
We want to verify ‘<>csp’ as correctness property of:

```
active proctype P() {
   do :: /* non-critical activity */
   atomic {
      !inCriticalQ;
      inCriticalP = true
   }
   csp = true;
   /* critical activity */
   csp = false;
   inCriticalP = false
   od
}
```

/* similarly for process Q */
/* here using csq */
Model Checking a Liveness Property with \textsc{JSpin}

1. open \textsc{Promela} file
2. enter $<>\text{csp}$ in LTL text field
3. select Translate to create a ‘never claim’, corresponding to the negation of the formula
4. ensure that \texttt{Acceptance} is selected
   (\textsc{Spin} will search for \textit{accepting} cycles through the never claim)
5. \textit{for the moment} uncheck \texttt{Weak Fairness} (see discussion below)
6. select Verify
Verification fails.

Why?
Verification fails.

Why?

The liveness property on one process ‘had no chance’. Not even weak fairness was switched on!
Always switch **Weak Fairness** on when checking for liveness!

1. open *PROMELA* file
2. enter `<>csp` in LTL text field
3. select Translate to create a ‘never claim’, corresponding to the negation of the formula
4. ensure that **Acceptance** is selected
   *(SPIN will search for *accepting* cycles through the never claim)*
5. ensure **Weak Fairness** is checked
6. select Verify
Model Checking Lifeness with \texttt{SPIN} directly

\textbf{Command Line Execution}

\begin{verbatim}
> spin -a -f "!csp" liveness1.pml
> gcc -o pan pan.c
> ./pan -a -f
\end{verbatim}
Verification Fails

Verification fails again.

Why?
Verification fails again.

Why?

Weak fairness is still too weak.
Verification fails again.

Why?

Weak fairness is still too weak.

Note that \texttt{inCriticalQ} is \textit{not} continuously executable!
Temporal MC Without Ghost Variables

We want to verify mutual exclusion without using ghost variables.

```c
#define mutex !(P@cs && Q@cs)

bool inCriticalP = false, inCriticalQ = false;

active proctype P() {
    do :: atomic {
        !inCriticalQ;
        inCriticalP = true
    }
    cs: /* critical activity */
    inCriticalP = false
    od
}
/* similarly for process Q */
/* with same label cs: */
```

Formal Specification and Verification: Model Checking

B. Beckert
Temporal MC Without Ghost Variables

We want to **verify mutual exclusion** without using ghost variables

```c
#define mutex !(P@cs && Q@cs)

bool inCriticalP = false, inCriticalQ = false;

active proctype P() {
    do :: atomic {
        !inCriticalQ;
        inCriticalP = true
    }
    cs: /* critical activity */
    inCriticalP = false
    od
}

/* similarly for process Q */
/* with same label cs: */

Verify ‘[]mutex’ with JSPIN.
```
Ben-Ari  Chapter 5