Applications of Formal Verification
Model Checking with Temporal Logic

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2010
Model Checking with SPIN

- model name.pml
- correctness properties

SPIN

- a

verifier pan.c

C compiler

executable verifier pan

random/ interactive/ guided simulation

failing run name.pml.trail

“errors: 0”
Correctness properties can be stated syntactically within or outside the model.
Correctness properties can be stated syntactically within or outside the model.

- stating properties within model using
  - assertion statements
Correctness properties can be stated syntactically 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 syntactically 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 syntactically \textit{within} or \textit{outside} the model.

\begin{itemize}
  \item stating properties within model using
    \begin{itemize}
      \item assertion statements
      \item meta labels
        \begin{itemize}
          \item end labels
          \item accept labels
          \item progress labels
        \end{itemize}
    \end{itemize}
  \item stating properties outside model using
    \begin{itemize}
      \item never claims
      \item \textit{temporal logic formulas} (today's main topic)
    \end{itemize}
\end{itemize}
Model Checking of Temporal Properties

many correctness properties not expressible by assertions
many correctness properties not expressible by assertions today:

model checking of properties formulated in temporal logic
Model Checking of Temporal Properties

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.

```c
critical++; assert (critical <= 1); critical--;
```
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--;
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
\textbf{critical++; assert ( critical} \textbf{<= 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
  \[ \text{critical} \leq 1 \text{ holds throughout the run} \]

- Array Index within Bounds (given array \(a\) of length \(len\))
  \[ 0 \leq i \leq len - 1 \text{ holds throughout the run} \]

- 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”
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"
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 <= 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.”

all these are temporal 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 \( \text{len} \))

“0 <= i <= \( \text{len}-1 \) 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 \( \Rightarrow \) use temporal logic
Boolean Temporal Logic

talking about numerical variables (like in \( \text{critical} \leq 1 \) 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 \texttt{bool/bit} are $\in For_{BTL}$
Boolean Temporal Logic over PROMELA

Set $\text{For}_{\text{BTL}}$ of Boolean Temporal Formulas (simplified)

- all PROMELA variables and constants of type bool/bit are $\in \text{For}_{\text{BTL}}$
- 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 $\in \text{For}_{\text{BTL}}$
Boolean Temporal Logic over PROMELA

Set $\text{For}_{BTL}$ of Boolean Temporal Formulas (simplified)

- All PROMELA variables and constants of type \texttt{bool/bits} are $\in \text{For}_{BTL}$
- If $e_1$ and $e_2$ are numerical PROMELA expressions, then all of $e_1==e_2$, $e_1!=e_2$, $e_1\lt e_2$, $e_1\leq e_2$, $e_1>e_2$, $e_1\geq e_2$ 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 $l$”)
Boolean Temporal Logic over PROMELA

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 $e_1==e_2$, $e_1!=e_2$, $e_1>e_2$, $e_1>e_2$, $e_1>=e_2$ 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 $l$”)
- if $\phi$ and $\psi$ are formulas $\in For_{BTL}$, then all of $!\phi$, $\phi && \psi$, $\phi || \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$, $[]\phi$, $<>\phi$, $\phi U \psi$

are $\in For_{BTL}$
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 $L_0, I_0$, $L_1, I_1$, $L_2, I_2$, $L_3, I_3$, $L_4, I_4$, $\ldots$

$L_j$ 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

\[ \mathcal{L}_0, \mathcal{I}_0 \rightarrow \mathcal{L}_1, \mathcal{I}_1 \rightarrow \mathcal{L}_2, \mathcal{I}_2 \rightarrow \mathcal{L}_3, \mathcal{I}_3 \rightarrow \mathcal{L}_4, \mathcal{I}_4 \rightarrow \ldots \]

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

$\mathcal{I}_j$ maps each variable in $M$ to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g.

\[ \mathcal{L}_j, \mathcal{I}_j \models x < y \quad \text{iff} \quad \mathcal{I}_j(x) < \mathcal{I}_j(y) \]

\[ \mathcal{L}_j, \mathcal{I}_j \models P @ l \quad \text{iff} \quad \mathcal{L}_j(P) \text{ is the location labeled with } l \]
Semantics of Boolean Temporal Logic

A run $\sigma$ through a PROMELA model $M$ is a chain of states $L_0, I_0, L_1, I_1, L_2, I_2, L_3, I_3, L_4, I_4, \ldots$

$L_j$ 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 $I_j(x) < I_j(y)$

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

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

but
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
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 $\Box \phi$ are called safety properties
Safety Properties

Formulas of the form $\Box \phi$ are called safety properties. They state that something good, $\phi$, is guaranteed throughout each run.

A special case is $\Box \neg \psi$, which states that something bad, $\psi$, never happens. An example is $\Box (\text{critical} \leq 1)$, which guarantees that at most one process visits its critical section. This is equivalent to saying that more than one process visiting its critical section will never happen.
Safety Properties

Formulas of the form $[]\phi$ are called safety properties. They 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: ‘$\Box (\text{critical} \leq 1)$’
Safety Properties

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: ‘$[]($critical $\leq 1)$’

“it is guaranteed throughout each run that at most one process visits its critical section”
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: ‘$\Box (\text{critical} \leq 1)$’
“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”
Applying Temporal Logic to Critical Section Problem

We want to verify ‘\([\text{critical} \leq 1]\)’ as correctness property of:

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

/* 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
Never Claims

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
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: */
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.
```
formulas of the form $\langle \rangle \phi$ are called liveness properties
formulas of the form $\langle\rangle \phi$ are called liveness properties
state that something good, $\phi$, eventually happens in each run

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 $P$)
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”
Applying Temporal Logic to Starvation Problem

We want to verify ‘<>csp’ as correctness property of:

```c
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 JS\textsc{pin}

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 (\textsc{spin} will search for accepting cycles through the never claim)
5. for the moment uncheck 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!
Fairness

Does the following PROMELA model necessarily terminate?

```pascal
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.
Does the following PROMELA model necessarily terminate?

```plaintext
byte n = 0;
bear 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?

```pascal
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.
Model Checking Liveness with Weak Fairness!

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 Liveness with **SPIN** directly

Command Line Execution

```bash
> spin -a -f "!csp" liveness1.pml
> gcc -o pan pan.c
> ./pan -a -f
```
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 !inCriticalQ is not continuously executable!
Verification Fails

Verification fails again.

Why?

Weak fairness is still too weak.

Note that \( \text{inCriticalQ} \) is not continuously executable!

Designing a fair mutual exclusion algorithm is complicated.
Literature for this Lecture

Ben-Ari  Chapter 5