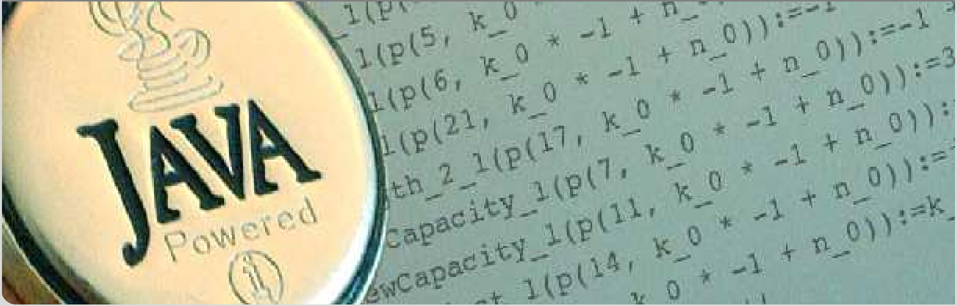
Applications of Formal Verification

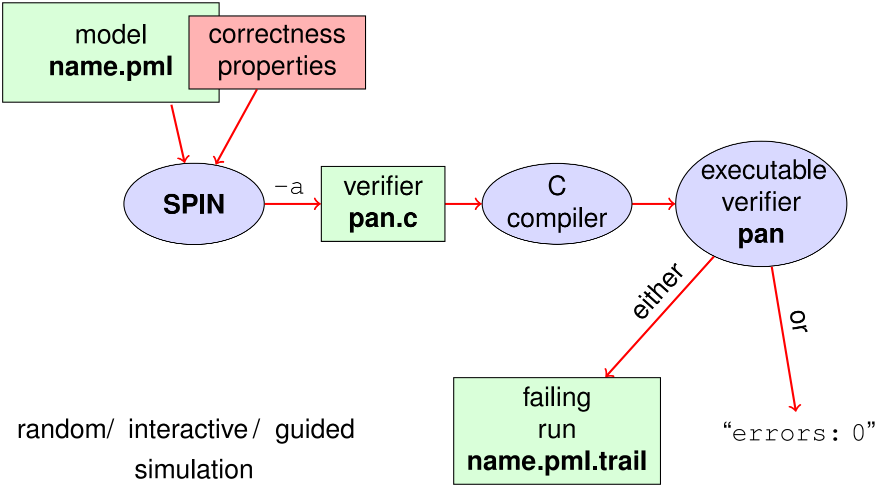
Model Checking with Temporal Logic

Bernhard Beckert · Mattias Ulbrich | SS 2019

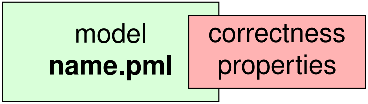
KIT – Institut für Theoretische Informatik



Model Checking with SPIN

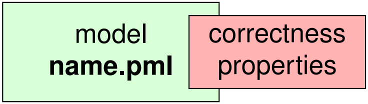


Stating Correctness Properties



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

Stating Correctness Properties



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

stating properties within model using

- assertion statements

Stating Correctness Properties

model
name.pml

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 Correctness Properties

model **name.pml**	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

Stating Correctness Properties

KIT Karlsruhe Institute of Technology

model
name.pml

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* (today's main topic)

Model Checking of Temporal Properties

many correctness properties not expressible by assertions

Model Checking of Temporal Properties

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.

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.

```
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.

```
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.

```
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.

```
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

Karlsruhe Institute of Technology

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

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

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 $\Rightarrow$ *use temporal logic*

Boolean Temporal Logic

talking about numerical variables (like in `critical <= 1` or `0 <= i <= 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

Boolean Temporal Logic over PROMELA

Set For_{BTL} of Boolean Temporal Formulas (simplified)

- all PROMELA variables and constants of type **`bool`**/**`bit`** are $\in For_{BTL}$

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 `e1` and `e2` are numerical PROMELA expressions, then all of `e1==e2`, `e1!=e2`, `e1<e2`, `e1<=e2`, `e1>e2`, `e1>=e2` are $\in For_{BTL}$

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 `e1` and `e2` are numerical PROMELA expressions, then all of `e1==e2`, `e1!=e2`, `e1<e2`, `e1<=e2`, `e1>e2`, `e1>=e2` are $\in For_{BTL}$
- if `P` is a process and `l` is a label in `P`, then `P@l` is $\in For_{BTL}$ ("`P` is at `l`", also available as `P[`*pid*`]@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 `e1` and `e2` are numerical PROMELA expressions, then all of `e1==e2`, `e1!=e2`, `e1<e2`, `e1<=e2`, `e1>e2`, `e1>=e2` are $\in For_{BTL}$
- if `P` is a process and `l` is a label in `P`, then `P@l` is $\in For_{BTL}$ ("`P` is at `l`", also available as `P[`*pid*`]@l`)
- if ϕ and ψ are formulas $\in For_{BTL}$, then all of

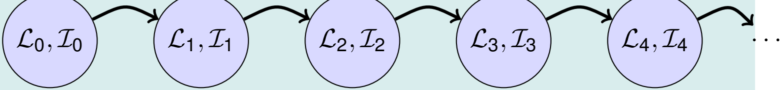
$$\neg\phi, \quad \phi \wedge\wedge \psi, \quad \phi \vee\vee \psi, \quad \phi \rightarrow \psi, \quad \phi \longleftrightarrow \psi$$

$$[\,]\phi, \quad <>\phi, \quad \phi\, \mathtt{U}\, \psi$$

are $\in For_{BTL}$

Semantics of Boolean Temporal Logic

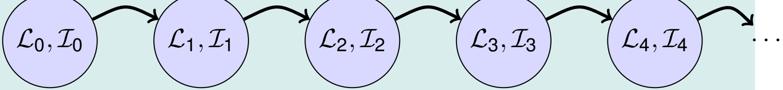
A run σ through a PROMELA model M is a chain of states



$\mathcal{L}_j$ 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.

Semantics of Boolean Temporal Logic

A run σ through a PROMELA model *M* is a chain of states

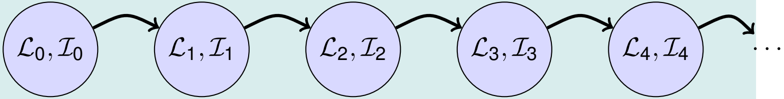


$\mathcal{L}_j$ maps each running process to its current location counter.
From $\mathcal{L}_j$ to $\mathcal{L}_{j+1}$, only one of the location counters has advanced
(exception: channel rendezvous).
$\mathcal{I}_j$ maps each variable in *M* to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g., $\mathcal{L}_j, \mathcal{I}_j \models \mathtt{x<y}$ iff $\mathcal{I}_j(\mathtt{x}) < \mathcal{I}_j(\mathtt{y})$

Semantics of Boolean Temporal Logic

A run σ through a PROMELA model M is a chain of states



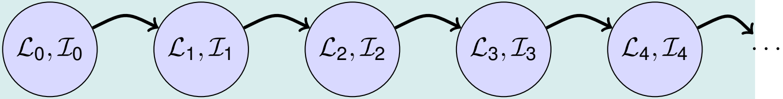
$\mathcal{L}_j$ maps each running process to its current location counter.
From $\mathcal{L}_j$ to $\mathcal{L}_{j+1}$, only one of the location counters has advanced (exception: channel rendezvous).
$\mathcal{I}_j$ maps each variable in M to its current value.

Arithmetic and relational expressions are interpreted in states as expected; e.g., $\mathcal{L}_j, \mathcal{I}_j \models \texttt{x<y}$ iff $\mathcal{I}_j(\texttt{x}) < \mathcal{I}_j(\texttt{y})$

$\mathcal{L}_j, \mathcal{I}_j \models \texttt{P@l}$ iff $\mathcal{L}_j(\texttt{P})$ is the location labeled with `l`.

Semantics of Boolean Temporal Logic

A run σ through a PROMELA model M is a chain of states



$\mathcal{L}_j$ 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., $\mathcal{L}_j, \mathcal{I}_j \models \mathtt{x<y}$ iff $\mathcal{I}_j(\mathtt{x}) < \mathcal{I}_j(\mathtt{y})$

$\mathcal{L}_j, \mathcal{I}_j \models \mathtt{P@l}$ iff $\mathcal{L}_j(\mathtt{P})$ is the location labeled with $\mathtt{l}$.

Evaluating other formulas $\in$ For_{BTL} in a run σ: as usual (see the book / “Formale Systeme”).

Boolean Temporal Logic Support in SPIN

SPIN supports Boolean temporal logic

Boolean Temporal Logic Support in SPIN

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

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`**

Temporal Logic Quiz

What does the following LTL formula mean?

[]((Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)

Temporal Logic Quiz

What does the following LTL formula mean?

`[](( Q & !R & <>R) -> (P -> (!R U (S & !R))) U R)`

P triggers S between Q (e.g., end of system initialization) and R (start of system shutdown).

Safety Properties

Safety properties are formulas for which a finite prefix of a run suffices as counterexample.

Safety Properties

Safety properties are formulas for which a finite prefix of a run suffices as counterexample.

Often have the form $[\,]\phi$:
something good, ϕ, is guaranteed throughout each run resp.
something bad, $\neg\phi$, never happens

Safety Properties

Safety properties are formulas for which a finite prefix of a run suffices as counterexample.

Often have the form $[]\phi$:
something good, ϕ, is guaranteed throughout each run resp.
something bad, $\neg\phi$, never happens

example: ‘`[](critical <= 1)`’

Safety Properties

Safety properties are formulas for which a finite prefix of a run suffices as counterexample.

Often have the form $[]\phi$:
something good, ϕ, is guaranteed throughout each run resp. something bad, $\neg\phi$, never happens

example: ‘`[](critical <= 1)`’

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

Safety Properties

Safety properties are formulas for which a finite prefix of a run suffices as counterexample.

Often have the form $[]\phi$:
something good, ϕ, is guaranteed throughout each run resp.
something bad, $\neg\phi$, never happens

example: ‘`[](critical <= 1)`’

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

or equivalently:
“more than one process being in its critical section will never happen”

Applying Temporal Logic to Critical Section Problem

We want to verify `'[](critical<=1)'` as correctness property of:

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

/* similarly for process Q */
```

Model Checking a Safety Property with JSPIN

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

```
#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

```
#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.

Liveness Properties

Liveness properties are formulas where potential counterexamples are necessarily infinite runs.

Liveness Properties

Liveness properties are formulas where potential counterexamples are necessarily infinite runs.

Often of the form $<>\phi$:
something good, ϕ, eventually happens in each run

Liveness Properties

Liveness properties are formulas where potential counterexamples are necessarily infinite runs.

Often of the form $<>\phi$:
something good, ϕ, eventually happens in each run

example: ‘`<>csp`’

(with `csp` a variable only true in the critical section of `P`)

Liveness Properties

Liveness properties are formulas where potential counterexamples are necessarily infinite runs.

Often of the form $<>\phi$:
something good, ϕ, 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:

```
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 JSPIN

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. *for the moment* uncheck `Weak Fairness` (see discussion below)
6. select `Verify`

Verification Fails

Verification fails.

Why?

Verification Fails

Verification fails.

Why?

The liveness property on one process ‘had no chance’.
The scheduler can unfairly select the other process all the time.

Fairness

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
}
```

Fairness

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!

Fairness

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.

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

```
> spin -a -f "!csp" liveness1.pml
> gcc -o pan pan.c
> ./pan -a -f
```

Verification Fails

Verification fails again.

Why?

Verification Fails

Verification fails again.

Why?

Weak fairness is still too weak.

Verification Fails

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 `!inCriticalQ` is not continuously executable!

Designing a fair mutual exclusion algorithm is complicated.

Literature for this Lecture

Ben-Ari Chapter 5

Progress Labels

- name starts with `progress`.
- must be traversed (infinitely often) in any infinite execution,
- otherwise: “non-progress cycle”.

Accept Labels

- name starts with `accept`.
- state cannot persist forever, and
- cannot be revisited infinitely often