INF5140: Specification and Verification of Parallel Systems

Lecture 09 – Defining Correctness Claims

Gerardo Schneider (author) Cristian Prisacariu (lecturer)

Department of Informatics
University of Oslo

INF5140, Spring 2011
Credits

Many slides (all the figures with blue background and few others) were taken from Holzmann’s slides on “Logical Model Checking”, course given at Caltech
A Spin model consists of

- Behavior specification (*what is possible*)
  - Asynchronous process behavior
  - Variables, data types
  - Message channels

- Logical correctness properties (*what is valid*)
  - Assertions
  - End-state, progress-state, and acceptance state labels
  - Never claims
  - Trace assertions
  - Temporal logic formulae
  - Default properties checked automatically:
    - Absence of system deadlock
    - Absence of dead code (unreachable code)

We will see how to express correctness properties in Promela and how to use Spin in order to (dis)prove them
Logical correctness is concerned with possibilities not with probabilities

- We are interested in determining which design requirements could possibly be violated, not in how probable such violation might be.

- The above distinction help strengthen the proofs of correctness
  - If the verifier says there is no possible violation of a given requirement, this is stronger than simple saying that the violation has a low probability of occurrence.

- In Promela/Spin the proof of correctness properties of distributed systems is intended to be independent of any assumptions about
  - The relative speeds of execution of processes
  - The execution time of each instruction
  - The probability of occurrence of particular types of events (e.g. lost of messages)
Introduction

- The above assumptions are reasonable for proving certain correctness properties of, e.g., communication protocols and distributed operating systems.
- In other cases (e.g., in hardware verification and real-time critical systems) time—and speed—may be relevant.
  - E.g., the correctness of a chip may depend on the delays of signal propagation.
Safety and Liveness

Safety and liveness are standard types of correctness requirements

- **Safety** establishes that "something bad never happens" (i.e., the set of properties the system may not violate)
  - Example: System invariance (e.g., $x$ is always less than $y + 5$)
  - The model checker will search for any possible execution that leads to the violation of a safety property (the "bad thing" that should not happen)

- **Liveness** establishes that "something good eventually happens" (i.e., the set of properties the system must satisfy)
  - Example: Responsiveness (e.g., every request is eventually followed by an acknowledgement)
  - The model checker will search for any possible execution in which the "good thing" can be postponed indefinitely
State and Path Properties

- At a lower level, there are two types of correctness claims
  - **State properties**, i.e., claims about reachable or unreachable states
  - **Path properties**, i.e., claims about feasible or unfeasible executions

- Both state and path properties may be specified in Promela and verified by Spin

The following are typical **state** properties

- **System invariants**: properties which should hold in every reachable state of the system
- **Local process assertions**: properties which should hold in specific reachable states
- **End-state labels**: to define proper termination points of processes

Example of **path** properties are:

- **Accept-state labels**: when looking for *acceptance cycles*
- **Progress-state labels**: when looking for *non-progress cycles*
- **Never claims**: (or derived from LTL formulae) used to specify finite or infinite system behavior that should *never* occur
- **Trace assertions**: properties of message channels
Correctness Properties in Promela

In Promela, correctness properties are formalized using the following constructs:

- **Basic assertions**
  - Local process assertions
  - System invariants
- **Meta labels**
  - End-state labels
  - Progress-state labels
  - Accept-state labels
- **Fair cycles**
- **never claims**
- **Trace assertions**
Basic Assertions

- Are statements of the form `assert(expression)`

```c
byte state = 1;
aactive proctype A()
{
    (state == 1) -> state++;
    assert(state == 2)
}
active proctype B()
{
    (state == 1) -> state--;
    assert(state == 0)
}
```

```
$ spin -a simple.pml
$ gcc -o pan pan.c
$ ./pan -E # -E means ignore invalid endstate errors...
pan: assertion violated (state==2) (at depth 6)
pan: wrote simple.pml.trail
...
```

```
$ spin -t -p simple.pml
1: proc  1 (B) line   7 "simple.pml" (state 1) [((state==1))]  
2: proc  0 (A) line   3 "simple.pml" (state 1) [((state==1))]  
3: proc  1 (B) line   7 "simple.pml" (state 2) [state--]   
4: proc  1 (B) line   8 "simple.pml" (state 3) [assert((state==0))]  
5: proc  0 (A) line   3 "simple.pml" (state 2) [state++]   
spin: line   4 "simple.pml", Error: assertion violated
spin: text of failed assertion: assert((state==2))
```
Basic Assertions

Preventing the Race

How to prevent the race condition in the previous example?

```
byte state = 1;
active proctype A()
{    atomic { (state == 1) -> state++ };    assert(state == 2)
}
active proctype B()
{    atomic { (state == 1) -> state-- };    assert(state == 0)
}
```

Q: are there invalid endstates?

we added two atomic sequences to create indivisible test&sets

nothing is unreachable
Basic Assertions
System Invariants

- It is possible to define **system invariants** using assertions

```plaintext
mtype = { p, v };  
chan sem = [0] of { mtype };  
byte count;  
active proctype semaphore()  
{    do      :: sem!p ->     sem?v     od} 
active [5] proctype user()  
{    do  :: sem?p ->      count++;     /* critical section */      count--;      sem!v     od} 
active proctype invariant()  
{    assert(count <= 1) } 
```

Q: how expensive is it to check the invariant in this way?

Adding active proctype invariant multiplies the search space 3x...
(from 16 reachable states to 48)
Basic Assertions

System Invariants

- We can improve performance and save some space...

```
mtype = { p, v };  
chan sem = [0] of { mtype };  
byte count;  
active proctype semaphore()
{
  do  
    :: sem!p ->  
      sem?v  
    od  
}

active [5] proctype user()
{
  do  
    :: sem?p;  
      count++;  
      /* critical section */  
      count--;  
      sem!v  
    od  
}
```

---

```
active proctype invariant()
{
  do :: assert(count <= 1) od
}
```

```
assert(count <= 1)
```

**Instantiation**

- No increase in number of reachable states (more transitions, but not more states)

- Can also put the assertion inside proctype user to check it only when the value of the expression could change.
End States

- When checking for reachable *deadlock* states, the verifier must be able to distinguish valid system *end* states from invalid ones.
- By default, the only valid end states are those in which every instantiated Promela process has reached the end of its code.
- Not all the processes, however, are meant to reach the end of its code (e.g., waiting loop or state).
- Special labels are needed to tell the verifier that those states are valid end states: *end-state labels*.
- Every label starting with the 3-letter prefix *end* defines an end-state label.
  - **Examples**: `endone`, `end_two`, `end_whatever_you_want`.
- Spin checks invalid end states by default.
  - It is possible to disable it by calling Spin with `–E` option.
End States

Example

mtype = { p, v };
chan sem = [0] of { mtype };
byte count;
active proctype semaphore()
{ end: do
:: sem!p ->
    sem?v
od }
active [5] proctype user()
{ end: do
:: sem?p;
    count++;
/* critical section */
    count--;
    sem!v
od }

neither process is intended
to terminate
the proper endstate in
both proctypes is s_0

the model check can now search
for reachable invalid end-states
End States

Example

$ spin -a semaphore.pml
$ cc -o pan pan.c
$ ./pan

(Spin Version 4.2.6 -- 27 October 2005)
  + Partial Order Reduction

Full statespace search for:
  never claim          - (none specified)
  assertion violations +
  acceptance cycles    - (not selected)
  invalid end states   +

State-vector 40 byte, depth reached 5, errors: 0
  16 states, stored
  5 states, matched
  21 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)

2.622 memory usage (Mbyte)

There are no errors: no invalid end state
At the end “unreached ... line 13” and “unreached ...
  line 24” show that non of the processes terminates (they don’t reach the ending “}”

unreached in proctype semaphore
  line 13, state 6, "-end-"
  (1 of 6 states)
unreached in proctype user
  line 24, state 8, "-end-"
  (1 of 8 states)
Progress-State Labels

- Statements labeled with progress-state labels are used to check that the process is really making progress, not just idling or waiting for other processes to make progress.
- The verifier can check that every potentially infinite execution cycle permitted by the model passes through at least one of its progress labels.
- If the verifier finds cycles without the above property, it declares that there are non-progress loop –corresponding to possible starvation.
- So, what Spin does is to check for the absence of non-progress cycles.
  - The verifier needs to be compiled with the special option -DNP.
- Notice that enabling the search for non-progress properties (a liveness property) automatically disable the search for invalid end states (a safety property).
- In simulation runs, such labels have no meaning.
Example

```plaintext
mtype = { p, v };
chan sem = [0] of { mtype };
byte count;
active proctype semaphore()
{
do
:: sem!p ->
progress: sem?v
od
}
active [5] proctype user()
{
do
:: sem?p ->
count++;
/* critical section */
count--;
sem!v
od
}

we make effective progress each time a user gains access to the critical section: each time state s₁ is reached in proctype semaphore

the model checker can now search for reachable non-progress cycles
```
Example
Verification

$ spin -a sem-prog.pml
$ cc -DNP -o pan pan.c # enable non-progress checking
$ ./pan -l # search for non-progress cycles

(Spin Version 4.2.6 -- 27 October 2005)
+ Partial Order Reduction

Full statespace search for:
  never claim +
  assertion violations + (if within scope of claim)
  non-progress cycles + (fairness disabled)
  invalid end states - (disabled by never claim)

State-vector 44 byte, depth reached 9, errors: 0
  21 states, stored
  5 states, matched
  26 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)

2.622 memory usage (Mbyte)

unreached in proctype semaphore
  line 13, state 6, "-end-
  (1 of 6 states)

unreached in proctype user
  line 24, state 8, "-end-
  (1 of 8 states)

There are no errors: no assertion violations nor non-progress cycles were found
This means the model does not permit infinite executions that do not contain infinitely many semaphore v operations
If the \textit{progress} label is removed from the previous example, what would be the result?

A non-progress cycle would be found!

Why?

Because if \textit{progress} labels are not used, every cycle (in this case there is only one) is guaranteed to be non-progress cycle.

After applying the verification you can obtain a counter-example by executing \texttt{spin -t -p sem-prog2.pml}.
Another Example

byte x = 2;
active proctype A() {
  do
  :: x = 3 - x
  od
}
active proctype B() {
  do
  :: x = 3 - x
  od
}
x alternates between values 2 and 1 ad infinitum
each process has just 1 state
no progress labels used just yet: every cycle is
a non-progress cycle

$ spin -a fair.pml
$ gcc -DNP -o pan pan.c  # non-progress cycle detection
$ ./pan -l  # invoke np-cycle algorithm
  pan: non-progress cycle (at depth 2)
  pan: wrote fair.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
+ Partial Order Reduction
Full statespace search for:
  never claim  +
  assertion violations  + (if within scope of claim)
  non-progress cycles  + (fairness disabled)
  invalid end states  - (disabled by never claim)
State-vector 24 byte, depth reached 7, errors: 1
  3 states, stored (5 visited)
  4 states, matched
  9 transitions (= visited+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

Q1: what happens if we mark one of the do-od
  loops with a progress
  label?
Q2: what happens if we mark both do-od loops?
Another Example

Obtaining a Counter-example

$ spin -t -p fair.pml
spin: couldn't find claim (ignored)
  2: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
  4: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
<<<<<<START OF CYCLE>>>>>
  6: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
  8: proc 1 (B) line 12 "fair.pml" (state 1) [x = (3-x)]
spin: trail ends after 8 steps
#processes: 2
  x = 2
  8: proc 1 (B) line 11 "fair.pml" (state 2)
  8: proc 0 (A) line 5 "fair.pml" (state 2)
2 processes created

we cannot make any assumptions about the relative speeds of processes
it is possible (though not probable) that process B makes infinitely many more steps than process A
the non-progress cycle reported by Spin is not necessarily a fair cycle

the claim was precompiled with -DNP
Accept States

- **Accept-state** labels are usually used in *never* claims, but not necessarily.
- By marking a state with a label which start with the prefix *accept* the verifier can be asked to find all cycles that *do* pass through at least one of those labels.
- The implicit correctness claim expressed by the presence of an accept-state label is that there should *not* exist any execution that can pass through an accept-state label infinitely often.
- In simulation runs, such labels have no meaning.
Example

```

mtype = { p, v };
chan sem = [0] of { mtype };
byte count;
active proctype semaphore()
{
  do
    :: sem!p ->
      sem?v
    od
}
active [5] proctype user()
{
  do
    :: sem?p ->
      accept: count++; /* critical section */
      count--; sem!v
  od
}

we may want to find infinite executions that do pass through a specially marked state
the state can be marked with an accept label
the model checker can now search for reachable acceptance cycles
```
Why are they called acceptance cycles?
It has to do with the automata theoretic foundation we have seen
never claims (discussed later) formally define $\omega$-automata that accept only those sequences that violate a correctness claim

acceptance cycle:
a state marked with an accept label
that is reachable from the initial system state and is also reachable from itself
i.e.,
a strongly connected component in the reachability graph, containing at least one accept state
Since Spin does not make any assumption about relative speed of executing processes, it can give counter-examples where a process *pauses indefinitely*

We might be interested in the existence of property violations under more *fair* assumptions

One of such assumptions is the *finite progress* assumption: If a process *can* execute a statement, it will eventually proceed with that execution

There are two variations of this assumption

- **Weak fairness**: If a statement is executable (enabled) infinitely *long*, it will eventually be executed
- **Strong fairness**: If a statement is executable infinitely *often*, it will eventually be executed

Several interpretations are still possible – Fairness can be applied to

- Non-deterministic statement selection *within* a process
- Non-deterministic statement selection *between* processes
Spin contains a predefined option for enforcing one specific variant of weak-fairness (run-time option pan -l -f or pan -a -f): if a process contains at least one statement that remains executable infinitely long, that process will eventually execute. Applies only to infinite executions (cycles).
The notion of fairness used in Spin applies **only** to process scheduling

- It does not apply to the resolution of non-deterministic choices inside processes

However, any type of fairness can be expressed in LTL formulae

- Adding fairness assumptions increases the cost of verification
- Enforcing strong fairness constraints is more costly than enforcing weak fairness constraints
  - Weak: penalty is *linear* in the number of active processes
  - Strong: penalty is *quadratic* in the number of active processes
A never claim defines an observer process executing **synchronously** with the system.

```
never {
  do
  :: true
  :: (p) -> break
  od;
accept:
  do
  :: (q)
  od
}
(p)
(q)
true
```

**Property:**
the truth of \( p \) is always followed within a finite number of steps by the truth of \( \lnot q \)

**Never Claim (negation of property):**
the truth of \( p \) is not followed within a finite number of steps by the truth of \( \lnot q \)
Reasoning about Executions

- There are at least three different ways to formalize an execution in a concurrent system:
  - Sequence of states
  - Sequence of events (state transitions)
  - Sequence of propositions on states (state properties)
    - This is what Spin does!!
Reasoning about Executions

Example

- **Checking for every state that** $p$ **implies** $!q$ **is simple**
  - It is a system invariant that we can check with a monitor process:
    
    ```
    active proctype invariant() {
        do
            :: assert(!p || !q) /* p implies !q */
        od
    }
    ```

- **Consider checking:**
  - “Every state where property $p$ holds is followed by a state where property $!q$ holds” (i.e., a temporal instead of a causal property)
  - This does **not** work:
    
    ```
    active proctype invariant() /* first try */
    { (p) -> /* after p holds */
        accept:
            do
                :: (q) /* then forever q is bad */
            od
    }
    ```
Limitations of Previous Mechanisms

- In fact, with the mechanisms performed so far it is not possible to define properties about every single execution step.

- The previous property cannot be expressed so far.
  - “Every state where property $p$ holds is followed by a state where property $\neg q$ holds”

- never claims gives us the capability of defining this kind of properties.
  - It checks system properties just before and after each statement execution, no matter which process performs it.
Reasoning about Executions

Example

- The checker must execute **synchronously** with the system!

```
never {
  do
  :: true
  :: (p) -> break
  od;
accept:
  do
  :: (q) /* first p and then forever q is bad */
  od
}
```

- A never claim executes an expression statement at every step in an execution.
- Never claims are intended to observe system behavior; they should not contribute to system behavior.
- The automaton can be non-deterministic.
- The never claim tracks behavior and can identify the bad executions (in this case with an accept label).
- Be prepared to wait for p to become true at any point in the execution.
Never Claims

- A **never** claim can be either deterministic or non-deterministic.
- It can contain **all** control flow constructs including `if`, `do`, `unless`, `atomic`, `d_step`, `goto`.
- It should **only** contain side-effect free expression statements (corresponding to boolean propositions on system states).
  - `q?[ack] or nfull(q)` is okay, but not `q?ack` or `q!ack`.
- Used to define **invalid** execution sequences.
- It cannot block.
  - A block would mean that the pattern expressed cannot be matched.
  - The **never** claim process gives up trying to match the current execution sequence, backs up and tries to match another.
  - Pausing in the never claim must be represented explicitly with a self-loop on true.
- There is a complete match (corresponding to error found) when
  - The closing curly brace of never claim is reached.
  - An acceptance cycle is closed.
Never Claims

- Convention: use accept-state labels only in never claims and progress and end-state labels only in the behavior model.
- Special precautions are needed if non-progress conditions are checked in combination with never claims:
  - non-progress is normally encoded in Spin as a predefined never claim.
A never claim in a Spin model is defined **globally**

Within a claim we can therefore refer to:

- Global variables
- Message channels (using poll statements)
- Process control-flow states (remote reference operations)
- Predefined global variables such as `timeout`, `_nr_pr`, `np_` but **not** process local variables

In general, we cannot refer to events, only to properties of states

- The effect of an event has to be made visible in the state of the system to become visible to a claim
- Only trace assertions can refer to send/recv events...
Never Claims: Counterexamples

- Promela Behavior Specification
- Fairness Constraints
- Never Claim Specification (negation of properties: capturing violations)

Counterexamples to correctness claims
“Question \( q \) is always eventually followed by answer \( a \) (assume \( q \) and \( a \) are properties of states) BEFORE the next question is asked”

This requirement is violated by any execution where a \( q \) is not followed by an \( a \) at all, AND by any execution where a \( q \) follows a \( q \) without an \( a \) in between

```
never { 
    do 
    :: true 
    :: q -> break 
    od; 
accept0: do 
    :: !a 
    :: q -> break 
    od; 
accept1: do 
    :: true 
    od 
}
```

reaching the end of a never claim is an automatic error we can (but need not) make this explicit; as is done here
Never Claims

Convention

reaching the closing curly brace of a never claim means that the entire behavior pattern that was expressed was matched, and is always interpreted as an error (it should never happen)

never claims are designed to 'accept' bad behavior - property violations
“There is no execution where first $p$ becomes true, then $q$, and then $r$”

incorrect monitors only the first 3 steps in any execution....

correct version applies to an execution of any length
never claims can be obtained from LTL formula

The never claim automaton of the (negated) formula
![[](p \rightarrow <>!q)] can be obtained by executing the following Spin command:

```
spin -f '![[](p \rightarrow <>!q)]'
```

Alternatively,

- You can use the timeline editor (see Holzmann’s Chap. 13), or
- You can use the LTL 2 BA fast algorithm from LTL to Büchi Automata `ltl2b -f '![[](p \rightarrow <>!q)]'` (not distributed with Spin, see [http://www.liafa.jussieu.fr/~oddoux/ltl2ba/](http://www.liafa.jussieu.fr/~oddoux/ltl2ba/))

never claims are equally expressive as ω-word automata (and Büchi automata), so they are more expressive than LTL

- You can write properties with never claims which cannot be written with LTL
Referencing Process States

- From within a `never` claims we can refer to the control-flow states of any active process.
- The syntax of a remote reference is:
  
  \[
  \text{proctypename}[\text{pidnr}]@\text{labelname}
  \]

  This expression is true if and only if the process with process instantiation number `pidnr` is currently at the control-flow point marked with `labelname` in `proctypename`.

- Example:
  
  \[
  \text{user}[1]@\text{label}
  \]

- If there is only one process of type user, we can also omit the `[pid]` part and use a simpler form:
  
  \[
  \text{user}@\text{label}
  \]
Referencing Process States

- an example

```plaintext
mtype = { p, v };
chan sem = [0] of { mtype };
active proctype semaphore()
{
do :: sem!p ; sem!v od
}
active [2] proctype user()
{
  assert(_pid == 1 || _pid == 2);
  do :: sem?p ->
  crit: /* critical section */
     sem?v
  od
/* reaching the end of a never claim is always an error */
}
```

Q1: why not?
Q2: what if we added one anyway?

remote referencing expressions can only be used in never claims...
they are meant to monitor behavior not to define behavior

we do not need an accept label in the never claim in this case

a way to make sure we are using the right pid numbers in the claim

using a label, instead of a counter to check mutual exclusion

label names
process instantiation numbers
proctype names

Cristian Prisacariu (Ifi, UiO)
INF5140 - Lecture 09: Correctness Claims 24.03.2011 51 / 64
Checking Process Termination

```
active proctype runner()
{
    do
    :: ... ...
    :: else -> break
    od
}
```

The expression:
```
(runner@L)
```

will be true if and only if the process reaches label L. Once the process reaches this label, it can never proceed beyond it.

Another method:
We can also try to use the predefined global variable `_nr_pr` to count how many processes are running...
Trace Assertions

- Trace assertions can be used to reason about valid or invalid sequences of `send` and `receive` statements

```plaintext
mtype = { a, b };
chan p = [2] of { mtype };
chan q = [1] of { mtype };

trace {
  do
  :: p!a; q?b
  od
}
```

- This assertion only claims something about how send operations on channel `p` relate to receive operations on channel `q`.

- It claims that every send of a message `a` to `p` is followed by a receive of a message `b` from `q`.

- A deviation from this pattern triggers an error.

- If at least one send (receive) operation on a channel `q` appears in the trace assertion, all send (receive) operations on that channel `q` must be covered by the assertion.

- Cannot use variables in trace assertions.

- Cannot use any statement other than `send` or `receive` statements in trace assertions.

- Can use `q?_` to specify an unconditional receive.
Notrace Assertions

A `notrace` assertion states that a particular access pattern is impossible (it reverses the claim) invalid sequences of `send` and `receive` statements

```plaintext
mtype = { a, b };
chan p = [2] of { mtype };
chan q = [1] of { mtype };
notrace {
  if
    :: p!a; q?b
    :: q?b; p!a
  fi
}
```

this notrace assertion claims that there is no execution where the send of a message a to channel p is followed by the receive of a message b from q, or vice versa: it claims that there must be intervening sends or receives to break these two patterns of access

the notrace assertion is fully matched when the closing curly brace is reached
Correctness Claims

All correctness properties that can be verified with Spin can be interpreted as formal claims that certain types of behavior are, or are not, possible

- An **assertion** formalizes the claim
  - It is impossible for the given expression to evaluate to false when the assertion is reached

- An **end-state** label formalizes the claim
  - It is impossible for the system to terminate without all active processes having either terminated, or having stopped at a state that was marked with an end-state label

- A **progress-state** label formalizes the claim
  - It is impossible for the system to execute forever without passing through at least one of the states that was marked with a progress-state label infinitely often
Correctness Claims (cont.)

- An **accept-state** label formalizes the claim
  - It is impossible for the system to execute forever while passing through at least one of the states that was marked with an accept-state label infinitely often

- A **never claim** formalizes the claim
  - It is impossible for the system to exhibit the behavior (finite or infinite) that completely matches the behavior that is specified in the claim

- A **trace assertion** formalizes the claim
  - It is impossible for the system to exhibit behavior that does not completely match the pattern defined in the trace assertion
In all the cases, Spin will try to find a counterexample to at least one of the formal claims stated.

So, Spin never tries to prove the **correctness** of a specification, but the opposite!

Finding counterexamples instead of getting a direct proof allows the verifier to employ a more efficient search procedure.
Properties of Models

- Let \( E \) be the set of all executions of a model.
- Let \( f \) be a correctness property.
- A model **satisfies** a property if **all** its executions do:

  \[(E \models f) \iff (\forall \delta, (\delta \in E) \implies (\delta \models f))\]

- A model **violates** a property if **at least one** of its executions does:

  \[\neg(E \models f) \iff (\exists \delta, (\delta \in E) \land \neg(\delta \models f))\]

Ex: \( p \implies \Box(\Box q) \)
Negating a Property

- Showing that property \( f \) can be violated, i.e., showing that
  \[ \neg(E \models f) \]
  is **not** the same as showing that \( \neg f \) is satisfied, i.e., showing that
  \[ (E \models \neg f) \]

- If \( E \) violates \( f \), we have that:
  \[ \neg(E \models f) \iff (\exists \delta, (\delta \in E) \land \neg(\delta \models f)) \]
  But
  \[ (E \models \neg f) \iff (\forall \delta, (\delta \in E) \Rightarrow \neg(\delta \models f)) \]

- In fact we have that
  \[ (E \models \neg f) \implies \neg(E \models f) \text{ but } \neg(E \models f) \not\implies (E \models \neg f) \]
Example

The following is an example of a system where

\[-(E \models \neg f) \quad \text{and} \quad -(E \models f)\]

byte \(x = 0;\)
init {
  do
  :: x = 0
  :: x = 2
  od
}
never {
  f
  do
  :: assert(x == 0)
  od
}
violated by \(x = 2\)

never {
  \neg f
  do
  :: assert(x != 0)
  od
}
violated by \(x = 0\)

\(x\) is neither always zero
nor is it always non-zero
Further Reading

- This lecture was based on Chapter 4 of Holzmann’s book “The Spin Model Checker”
- Next lecture we’ll see a bit more of theory (Spin search algorithms)