How SPIN handles Linear Temporal Logic

- For every LTL predicate, there exists an NDFA that accepts any event sequence that satisfies that predicate.
- Because SPIN is looking for failures, it needs an NDFA that accepts any event sequence that ***does not*** satisfy that predicate.
- So we generate the NDFA for the ***negation*** of the predicate, which is then represented as a process that can be written in Promela.
  - This process is called a "never claim"
- This NDFA is then run in parallel with the rest of the Promela model.
  - If it ever enters an "accepting" state, then it has found a violation of the predicate
  - It also signals a failure if an **`assert()`** statement fails.

Writing LTL in Promela

- According to the books on the reading list:
  - The syntax of LTL in Promela only has variables, and the logical operators.
    - It does not have expressions (boolean or otherwise), such as x < 1 or y = z + 42.
  - Instead we need to use #define to link a variable to such expressions.
  - So `[]((x==42) -> <>(y==99))`

    has to be written as something like this:

    ```
    #define answer (x==42)
    #define balloons (y==99)
    [](answer -> <>balloons)
    ```
- However since version 6 of Spin:
  - We can simply write `[]((x==42) -> <>(y==99))`

From LTL to never claims

- Early versions of Promela/SPIN required the modeller to work out never claims by hand
- Then a version came along that could translate an LTL predicate into a never claim.

```
Promela> spin -f 'p'
never  {    /* p */
accept_init:
T0_init:
        do
        :: atomic { ((p)) -> assert(!((p))) }
        od;
accept_all:
        skip
}
```

```
Promela> spin -f '<>p'
never  {    /* <>p */
T0_init:
        do
        :: atomic { ((p)) -> assert(!((p))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

The output of spin -f '….' can be piped into a file which is then included in the Promela model file.

```
Promela> spin -f '[]p'
never  {    /* []p */
accept_init:
T0_init:
        do
        :: ((p)) -> goto T0_init
        od;
}
```

- Current versions allow LTL predicates to be written directly in the Promela file

Example: a bad mutex solution

- Example of an attempt to solve mutual exclusion
  - It ensures only `P()` or `Q()` in the critical section at one time
  - Unfortunately, it can deadlock!
    - Easily demonstrated by `spin -run`.
  - How can we use LTL to verify the exclusion property?

```
/* Copyright 2007 by Moti Ben-Ari
  under the GNU GPL; see readme.txt */

bool wantP = false, wantQ = false;

active proctype P() {
  do ::  wantP = true;
          !wantQ;
          // critical section
          wantP = false
  od
}

active proctype Q() {
  do ::  wantQ = true;
         !wantP;
         // critical section
         wantQ = false
  od
}
```

Principles of the Spin Model Checker, M. Ben-Ari, Sec 4.3, p52

mutual exclusion using LTL

- We add a counter **`critical`**
- Each process:
  - increments it on entry to the critical region
  - decrements it on exit from the critical region
- We define a LTL predicate **`msafe`** that asserts that **`critical`** is always less than 2 in any state.
  - If we use **`spin -a`** we can see the never claim generated, in **`_spin_nvr.tmp`**

```
/* Copyright 2007 by Moti Ben-Ari
  under the GNU GPL; see readme.txt */
bool wantP = false, wantQ = false;
byte critical;
active proctype P() {
  do ::  wantP = true;
         !wantQ;
         critical++;
         // in critical section
         critical--;
         wantP = false
  od
}
active proctype Q() {
  do ::  wantQ = true;
         !wantP;
         critical++;
         // in critical section
         critical--;
         wantQ = false
  od
}
ltl msafe { [](critical <=1) }
```

Principles of the Spin Model Checker,
M. Ben-Ari, Sec 5.3.2, pp75-78

Other ways … ?

- We have seen shared-variable concurrency
  - Threads have ***non-atomic*** read-write access to shared global variables in other threads that are part of the ***same*** Process.
  - The pthread library provides a discipline for avoiding unwanted interference
  - The Promela modelling we have done has dealt with this approach
- What if we didn't share variables?
  - How can we do this?
  - Would this make all our problems go away?

Other Ways.

- The main alternative is some form of ***message passing***
- Processes/Threads communicate by sending and receiving messages
  - All variables are kept local
- Message passing is the main mechanism used by operating systems to allow communication between Processes
  - Remember, different Processes do not (usually) share any memory.
- Message Passing is the only way to go when doing any networking code
  - Different computers, far apart, cannot share main memory!

To Synchronise or not to Synchronise?

- There are two forms of message passing: synchronous, and asynchronous
- Synchronous:
  - both sender and receiver wait until the communication has completed
  - If one thread does a send/receive, but another never does a receive/send, then that thread will deadlock!
  - Also known as "Rendezvous Communication"
- Asynchronous:
  - the sender returns immediately, while the message is buffered somewhere
  - the receiver will wait for a message if necessary
  - again the only way to efficiently perform networking
- Hybrid: it is possible to mix the two

Message Passing in Promela

- Distributed (network) systems consist of nodes connected by communication channels
  - Internet: computers connected by wires, optical fibres, wireless, satellite, and networking hardware running network and communication protocols
- Promela can model this
  - Nodes/Computers are modelled by Promela Processes
  - Network Communication is modelled using Promela Channels

Channels in Promela

- Declaring a channel: **`chan`** `ch` **`=`** **`[`**`capacity`**`]`** **`of`** **`{`** `typename, …, typename` **`}`**
  - capacity is size of (hidden) buffer - can be zero, or positive
  - `typename` cannot be an array type, but can be a struct that contains an array
  - `typename` can be **`chan`** - so we can send/receive communication channels !
- Writing/Sending to a channel: `ch` **`!`** `expr, …, expr`
  - the number and type of `expr` must match the types in the declaration for `ch`
- Reading/Receiving from a channel: `ch` **`?`** `var, …, var`
  - the number and type of `var` must match the types in the declaration for `ch`

Simple example: Server + 2*Client

PSMC, Listing 7.1, p107

- Channel `request` has no buffer
  - Synchronous communication
  - Only time in Promela that two threads perform a step at the exact same time (one does a send-step, the other does a receive step).
- Server loops forever
  - waiting for input on channel request
- Clients make one request and stop.
  - both write to the same channel

```
/* Copyright 2007 by Moti Ben-Ari
   under the GNU GPL; see readme.txt */

chan request = [0] of { byte };

active proctype Server() {
  byte client;
end:
  do
    ::  request ? client
          -> printf("Client %d\n", client);
  od
}

active proctype Client0() {
  request ! 0;
}

active proctype Client1() {
  request ! 1;
}
```

Buffered Channels

- If channel capacity is non-zero, then we can have asynchronous communication
  - Send puts stuff in buffer, if not full
  - Receive takes stuff, if not empty
- There are builtin predicates to check if a channel buffer is full, empty:
  - **`full`**, **`empty`**, **`nfull`**, **`empty`**
  - e.g. **`empty`**`(mychan)` is executable/true if the buffer for mychan has no messages,

```
/* Copyright 2007 by Moti Ben-Ari under the GNU GPL; see readme.txt */

chan request = [2] of { byte, chan};
chan reply[3] = [2] of { byte };

bool waiting = false;
/* Verify []!waiting to show clients waiting */

active [2] proctype Server() {
  byte client;
  chan replyChannel;
  do
  :: empty(request) ->
       printf("No requests for server %d\n", _pid)
  :: request ? client, replyChannel ->
       printf("Client %d processed by server %d\n", client, _pid);
       replyChannel ! _pid
  od
}

active [3] proctype Client() {
  byte server;
  do
  :: full(request) ->
       waiting = true;
       printf("Client %d waiting for non-full channel\n", _pid)
  :: request ! _pid, reply[_pid-2] ->
       reply[_pid-2] ? server;
     printf("Reply received from server %d by client %d\n", server, _pid)
  od
}
```

Shared Variable vs. Message Passing

- See https://wiki.c2.com/?MessagePassingConcurrency
- See https://wiki.c2.com/?SharedStateConcurrency