How to Decide the Model Checking Problem?

- How to answer \( L(Q) \cap L(\neg \varphi) = \emptyset \)?
  - Common approach:
    - Build automaton \( A \) such the \( L(A) = L(Q) \cap L(\neg \varphi) \)
    - Are accepting states of \( A \) reachable? (Infinitely often?)
  - How to build \( A \)?
    - One possible answer: Build a series of automata by recursion on structure of \( \neg \varphi \).
    - Another possible answer: Build an automaton \( B \) such \( L(B) = L(\neg \varphi) \); take \( A = B \times Q \)
  - Will do at least one approach if time after Spin

Theorem

The Model Checking Problem for finite transition systems and LTL formulae is decidable.

- Treat states \( q \in Q \) as letters in an alphabet.
- Language of \( (Q, \delta, I) \), \( L(Q, \delta, I) \) (or \( L(Q) \) for short) is set of runs in \( Q \)
- Language of \( \varphi \), \( L(\varphi) = \{ \sigma | \sigma \models \varphi \} \)
- Question: \( L(Q) \subseteq L(\varphi) \)?
- Same as: \( L(Q) \cap L(\neg \varphi) = \emptyset \)

SPIN Documentation


SPIN Overview

- Input:
  - (Abstract) model of system
  - Behavior specification
- Output:
  - Says whether model satisfies specification
  - If models fails specification, give a system run that violates requirement (counterexample)
- Focused on correctness of process communications and interactions
- Internal details generally abstracted away

LTL Model Checking

- Model Checking Problem: Given model \( M \) amd logical property \( \varphi \) of \( M \), does \( M \models \varphi \)?
- Given transition system with states \( Q \), transition relation \( \delta \) and initial state state \( I \), say \( (Q, \delta, I) \models \varphi \) for every run of \( (Q, \delta, I) \), \( \sigma \) satisfies \( \sigma \models \varphi \).

Introduction to SPIN and Promela

- SPIN Background
- Promela processes
- Promela statements
- Promela communication primitives
- Architecture of (X)Spin
- Some SPIN demo’s
  - hello world
  - mutual exclusion
  - alternating bit protocol

Slides based heavily on: Theo C. Ruys - SPIN Beginners’ Tutorial
**SPIN Introduction**

SPIN = Simple Promela Interpreter
- Tool for analyzing logical consistenct of concurrent systems
- Specifically data communication protocols
- State-of-the-art model checkers, thousands of users
- Concurrent systems described in modelling language Promela

Promela = Protocol/Process Meta Language
- Resembles C programming language
- Supports dynamic creation of concurrent processes
- Limited to describing finite-state systems
- Communication via message channels
  - Synchronous (rendezvous)
  - Asynchronous (buffered)

**Promela Models**

Promela model consist of:
- Type declarations
- Channel declarations
- Variable declarations
- Process declarations
- [init process]

A Promela model corresponds with a (usually very large, but) finite transition system, so
- No unbounded data
- No unbounded channels
- No unbounded processes
- No unbounded process creation

**Processes**

A process type (proctype) consists of
- A name
- A list of formal parameters
- Local variable declarations
- Body consisting a sequence of statements

**Sample Process Declaration**

```plaintext
proctype Sender (chan in; chan out) {
    bit sndB, rcvB; /* local variables */
    do /* body beginning */
        :: out ! MSG, sndB ->
        in ? ACK, rcvB;
        if :: sndB == rcvB => sndB = 1-sndB
        else => skip
    fi
    od /* body end */
}
```

The body consist of a sequence of statements.

**Promela Skeleton Example**

```plaintext
mtype = {MSG, ACK};
chan toS = ...
chan toP = ...
bool flag;
proctype Sender() {
    /* process body */
}
proctype Receiver() {
    /* process body */
}
init {
    /* creates processes */
}
```

Elsa L Gunter
CS477 Formal Software Development Methods
Slides mostly a reproduction of Theo C. Ruys – SPIN Beginners’ Tutorial April 6, 2018
/ 19
## Process Creation

- Processes created with `run` statement
  - Returns process id
- Process created at any point in execution (of any process)
- Processes start after execution of `run` statement
- Also created by `active` keyword before `proctype` declaration

## Sample Proctype Declaration Skeleton

```promela
proctype Foo(byte x) {
    ...
}
active[3] proctype Bar(byte y) {
    /* [3] opt; y init to 0 */
    ...
}
init {
    int pid2 = run Foo(2);
    run Bar(17);
    run Foo(27);
}
```

## Hello World

```promela
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init () {
    printf("init\n", _pid);
    run Hello();
    printf("last\n", lastpid);
}
```

## Hello World, Sample Execution

```
bash-3.2$ spin hello.pml
init process, my pid is: 1
Hello process, my pid is: 0
Hello process, my pid is: 2
last pid was: 2
3 processes created
```

```
bash-3.2$ spin hello.pml
Hello process, my pid is: 0
init process, my pid is: 1
last pid was: 2
Hello process, my pid is: 2
3 processes created
```

## Hello Processes

```
Hello()
print "Hello"
init()
print "init"
run Hello()
print "last"
print "Hello"
```

## Hello Processes Interleavings

```
Hello()
print "Hello"
Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
print "Hello"
>Hello()
Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved
  - Only one process can execute a statement at each point in time.
  - Exception: rendez-vous communication.
- All statements are atomic
  - Each statement is executed without interleaving it parts with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - Only one choice is made, non-deterministically (randomly).