Formal Methods for Distributed Systems
An Introduction

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

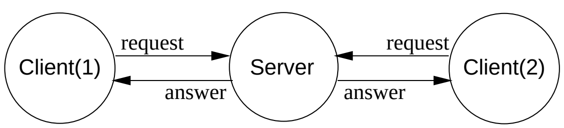
1. A Client/Server System

2. Modeling Concurrent Systems

3. Specifying System Properties

4. Verifying System Properties

A Client/Server System



- System of one server and two clients.
  - Three concurrently executing system components.
- Server manages a resource.
  - An object that only one system component may use at any time.
- Clients request resource and, having received an answer, use it.
  - Server ensures that not both clients use resource simultaneously.
  - Server eventually answers every request.

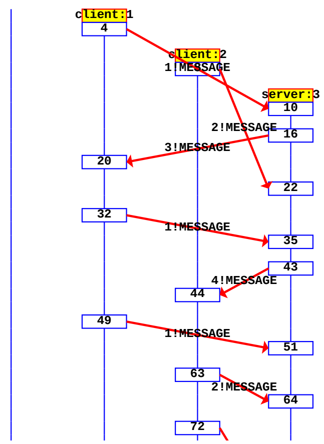
Set of system requirements.

System Implementation (Pseudo-Code)

```
Server:
  local given, waiting, sender
begin
  given := 0; waiting := 0
  loop
    sender := receiveRequest()
    if sender = given then
      if waiting = 0 then
        given := 0
      else
        given := waiting; waiting := 0
        sendAnswer(given)
      endif
    elsif given = 0 then
      given := sender
      sendAnswer(given)
    else
      waiting := sender
    endif
  endloop
end Server
```

```
Client(ident):
  param ident
begin
  loop
    ...
    sendRequest()
    receiveAnswer()
    ... // critical region
    sendRequest()
  endloop
end Client
```

Simulating the System Execution



Just one execution, infinitely many are possible!

Desired System Properties

- Property: mutual exclusion.
  - At no time, both clients are in critical region.
    - Critical region: program region after receiving resource from server and before returning resource to server.
  - The system shall only reach states, in which mutual exclusion holds.
- Property: no starvation.
  - Always when a client requests the resource, it eventually receives it.
  - Always when the system reaches a state, in which a client has requested a resource, it shall later reach a state, in which the client receives the resource.
- Problem: each system component executes its own program.
  - Multiple program states exist at each moment in time.
  - Total system state is combination of individual program states.
  - Not easy to see which system states are possible.

How can we verify that the system has the desired properties?

1. A Client/Server System

2. Modeling Concurrent Systems

3. Specifying System Properties

4. Verifying System Properties

System States

At each moment in time, a system is in a particular state.

- A state $s : Var \to Val$
  - A state s is a mapping of every system variable x to its value $s(x)$.
    - Typical notation: $s = [x = 0, y = 1, \ldots] = [0, 1, \ldots]$.
  - Var . . . the set of system variables
    - Program variables, program counters, . . .
  - Val . . . the set of variable values.
- The state space $State = \{s \mid s : Var \to Val\}$
  - The state space is the set of possible states.
    - The system variables can be viewed as the coordinates of this space.
  - The state space may (or may not) be finite.
    - If $|Var| = n$ and $|Val| = m$, then $|State| = m^n$.
    - A word of $\log_2 m^n$ bits can represent every state.

A system execution can be described by a path $s_0 \to s_1 \to s_2 \to \ldots$ in the state space.

Deterministic Systems

In a sequential system, each state typically determines its successor state.

- The system is deterministic.
  - We have a (possibly not total) transition function F on states.
  - $s_1 = F(s_0)$ means "s_1 is the successor of s_0".
- Given an initial state s_0, the execution is thus determined.
  - $s_0 \to s_1 = F(s_0) \to s_2 = F(s_1) \to \ldots$
- A deterministic system (model) is a pair $\langle I, F \rangle$.
  - A set of initial states $I \subseteq State$
    - Initial state condition $I(s) :\Leftrightarrow s \in I$
  - A transition function $F : State \overset{partial}{\to} State$.
- A run of a deterministic system $\langle I, F \rangle$ is a (finite or infinite) sequence $s_0 \to s_1 \to \ldots$ of states such that
  - $s_0 \in I$ (respectively $I(s_0)$).
  - $s_{i+1} = F(s_i)$ (for all sequence indices i)
  - If s ends in a state s_n, then F is not defined on s_n.

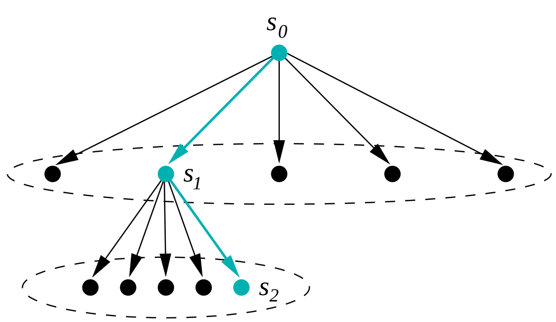
Nondeterministic Systems

In a concurrent system, each component may change its local state, thus the successor state is not uniquely determined.

- The system is nondeterministic.
  - We have a transition relation R on states.
  - $R(s_0, s_1)$ means "s_1 is a (possible) successor of s_0.
- Given an initial state s_0, the execution is not uniquely determined.
  - Both $s_0 \to s_1 \to \ldots$ and $s_0 \to s_1' \to \ldots$ are possible.
- A non-deterministic system (model) is a pair $\langle I, R \rangle$.
  - A set of initial states (initial state condition) $I \subseteq State$.
  - A transition relation $R \subseteq State \times State$.
- A run s of a nondeterministic system $\langle I, R \rangle$ is a (finite or infinite) sequence $s_0 \to s_1 \to s_2 \ldots$ of states such that
  - $s_0 \in I$ (respectively $I(s_0)$).
  - $R(s_i, s_{i+1})$ (for all sequence indices i).
  - If s ends in a state s_n, then there is no state t such that $R(s_n, t)$.

Reachability Graph

The transitions of a system can be visualized by a graph.



The nodes of the graph are the reachable states of the system.

Example: Digital Circuits

Synchronous composition of hardware components.

- A modulo 8 counter $C = \langle I_C, R_C \rangle$.

$State := \mathbb{N}_2 \times \mathbb{N}_2 \times \mathbb{N}_2$.

$I_C(v_0, v_1, v_2) :\Leftrightarrow v_0 = v_1 = v_2 = 0$.

$R_C(\langle v_0, v_1, v_2 \rangle, \langle v_0', v_1', v_2' \rangle) :\Leftrightarrow$
$R_0(v_0, v_0') \wedge$
$R_1(v_0, v_1, v_1') \wedge$
$R_2(v_0, v_1, v_2, v_2')$.

$R_0(v_0, v_0') :\Leftrightarrow v_0' = \neg v_0$.
$R_1(v_0, v_1, v_1') :\Leftrightarrow v_1' = v_0 \oplus v_1$.
$R_2(v_0, v_1, v_2, v_2') :\Leftrightarrow v_2' = (v_0 \wedge v_1) \oplus v_2$.

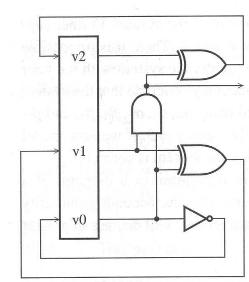


Figure 2.1
Synchronous modulo 8 counter.

Edmund Clarke et al: "Model Checking", 1999.

Example: Concurrent Software

Asynchronous composition of software components with shared variables.

P :: l_0 : **while** true **do**
  NC_0 : **wait** $turn = 0$
  CR_0 : $turn := 1$
end

||

Q :: l_1 : **while** true **do**
  NC_1 : **wait** $turn = 1$
  CR_1 : $turn := 0$
end

- A mutual exclusion program $M = \langle I_M, R_M \rangle$.

$State := PC \times PC \times \mathbb{N}_2$. // shared variable
$I_M(p, q, turn) :\Leftrightarrow p = l_0 \wedge q = l_1$.
$R_M(\langle p, q, turn \rangle, \langle p', q', turn' \rangle) :\Leftrightarrow$
$(P(\langle p, turn \rangle, \langle p', turn' \rangle) \wedge q' = q) \vee (Q(\langle q, turn \rangle, \langle q', turn' \rangle) \wedge p' = p)$.
$P(\langle p, turn \rangle, \langle p', turn' \rangle) :\Leftrightarrow$
$(p = l_0 \wedge p' = NC_0 \wedge turn' = turn) \vee$
$(p = NC_0 \wedge p' = CR_0 \wedge turn = 0 \wedge turn' = turn) \vee$
$(p = CR_0 \wedge p' = l_0 \wedge turn' = 1)$.
$Q(\langle q, turn \rangle, \langle q', turn' \rangle) :\Leftrightarrow$
$(q = l_1 \wedge q' = NC_1 \wedge turn' = turn) \vee$
$(q = NC_1 \wedge q' = CR_1 \wedge turn = 1 \wedge turn' = turn) \vee$
$(q = CR_1 \wedge q' = l_1 \wedge turn' = 0)$.

Example: Concurrent Software

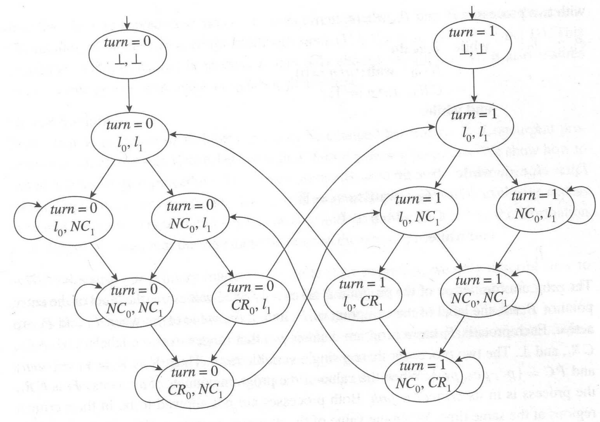


Figure 2.2
Reachable states of Kripke structure for mutual exclusion example.

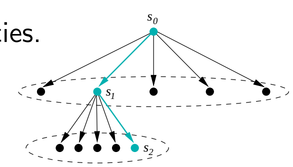
Edmund Clarke et al: "Model Checking", 1999.

Model guarantees mutual exclusion.

1. A Client/Server System
2. Modeling Concurrent Systems
3. **Specifying System Properties**
4. Verifying System Properties

Motivation

We need a language for specifying system properties.



- A system S is a pair $\langle I, R \rangle$.
  - Initial states I, transition relation R.
  - More intuitive: reachability graph.
    - Starting from an initial state s_0, the system runs evolve.
- Consider the reachability graph as an infinite computation tree.
  - Different tree nodes may denote occurrences of the same state.
    - Each occurrence of a state has a unique predecessor in the tree.
  - Every path in this tree is infinite.
    - Every finite run $s_0 \to \ldots \to s_n$ is extended to an infinite run $s_0 \to \ldots \to s_n \to s_n \to s_n \to \ldots$
- Or simply consider the graph as a set of system runs.
  - Same state may occur multiple times (in one or in different runs).

We need to talk about such trees respectively sets of system runs.

Computation Trees versus System Runs

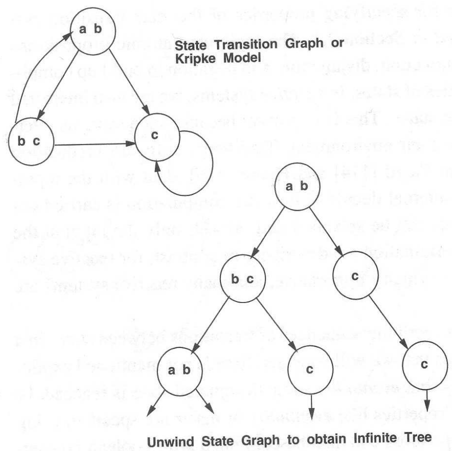


Figure 3.1
Computation trees.

Set of system runs:

$[a, b] \to c \to c \to \dots$
$[a, b] \to [b, c] \to c \to \dots$
$[a, b] \to [b, c] \to [a, b] \to \dots$
$[a, b] \to [b, c] \to [a, b] \to \dots$
...

Edmund Clarke et al: "Model Checking", 1999.

Temporal Logic

Extension of classical logic to reason about multiple states.

- Temporal logic is an instance of modal logic.
  - Logic of "multiple worlds (situations)" that are in some way related.
  - Relationship may e.g. be a temporal one.
  - Amir Pnueli, 1977: temporal logic is suited to system specifications.
  - Many variants, two fundamental classes.
- Branching Time Logic
  - Semantics defined over computation trees.
    At each moment, there are multiple possible futures.
  - Prominent variant: CTL.
    Computation tree logic; a propositional branching time logic.
- Linear Time Logic
  - Semantics defined over sets of system runs.
    At each moment, there is only one possible future.
  - Prominent variant: PLTL.
    A propositional linear time logic.

State Formula

Temporal logic is based on classical logic.

- A state formula F is evaluated on a state s.
  - Any predicate logic formula is a state formula:
    $p(x), \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1, \forall x : F, \exists x : F$.
  - In propositional temporal logic only propositional logic formulas are state formulas (no quantification):
    $p, \neg F, F_0 \wedge F_1, F_0 \vee F_1, F_0 \Rightarrow F_1, F_0 \Leftrightarrow F_1$.
- Semantics: $s \models F$ ("F holds in state s").
  - Example: semantics of conjunction.
    - $(s \models F_0 \wedge F_1) \;:\Leftrightarrow\; (s \models F_0) \wedge (s \models F_1)$.
    - "$F_0 \wedge F_1$ holds in s if and only if F_0 holds in s and F_1 holds in s".

Classical logic reasons on individual states.

Linear Time Logic (LTL)

We use temporal logic to specify a system property P.

- Core question: $S \models P$ ("P holds in system S").
  - System $S = \langle I, R \rangle$, temporal logic formula P.
- Linear time logic:
  - $S \models P \;:\Leftrightarrow\; r \models P$, for every run r of S.
  - Property P must be evaluated on every run r of S.
  - Given a computation tree with root s_0, P is evaluated on every path of that tree originating in s_0.
    - If P holds for every path, P holds on S.

LTL formulas are evaluated on system runs.

Formulas

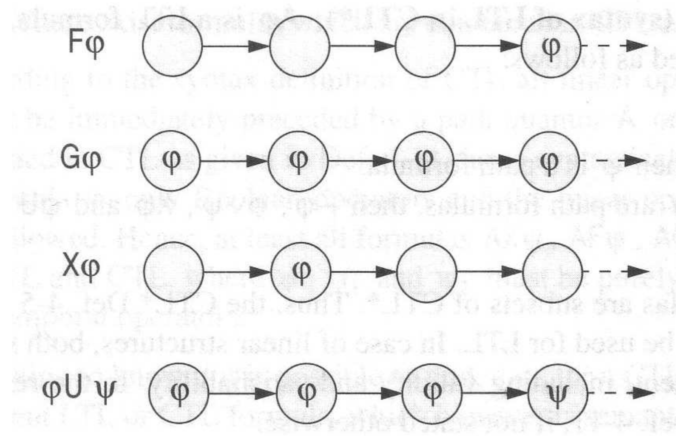
No path quantifiers; all formulas are path formulas.

- Every formula is evaluated on a path p.
  - Also every state formula f of classical logic (see below).
  - Let F and G denote formulas.
  - Then also the following are formulas:

    $\mathbf{X}\ F$ ("next time F"), often written $\bigcirc F$,
    $\mathbf{G}\ F$ ("always F"), often written $\Box F$,
    $\mathbf{F}\ F$ ("eventually F"), often written $\Diamond F$,
    $F\ \mathbf{U}\ G$ ("F until G").
- Semantics: $p \models P$ ("P holds in path p").
  - $p^i := \langle p_i, p_{i+1}, \ldots \rangle$.

    $p \models f :\Leftrightarrow p_0 \models f.$
    $p \models \mathbf{X}\ F\ :\Leftrightarrow p^1 \models F.$
    $p \models \mathbf{G}\ F\ :\Leftrightarrow \forall i \in \mathbb{N} : p^i \models F.$
    $p \models \mathbf{F}\ F\ :\Leftrightarrow \exists i \in \mathbb{N} : p^i \models F.$
    $p \models F\ \mathbf{U}\ G\ :\Leftrightarrow \exists i \in \mathbb{N} : p^i \models G \wedge \forall j \in \mathbb{N}_i : p^j \models F.$

Formulas



Thomas Kropf: "Introduction to Formal Hardware Verification", 1999.

Frequently Used LTL Patterns

In practice, most temporal formulas are instances of particular patterns.

Pattern	Pronounced	Name
$\Box F$	always F	invariance
$\Diamond F$	eventually F	guarantee
$\Box\Diamond F$	F holds infinitely often	recurrence
$\Diamond\Box F$	eventually F holds permanently	stability
$\Box(F \Rightarrow \Diamond G)$	always, if F holds, then eventually G holds	response
$\Box(F \Rightarrow (G\ \mathbf{U}\ H))$	always, if F holds, then G holds until H holds	precedence

Typically, there are at most two levels of nesting of temporal operators.

Examples

- Mutual exclusion: $\Box\neg(pc_1 = C \wedge pc_2 = C)$.
  - Alternatively: $\neg\Diamond(pc_1 = C \wedge pc_2 = C)$.
  - Never both components are simultaneously in the critical region.
- No starvation: $\forall i : \Box(pc_i = W \Rightarrow \Diamond pc_i = R)$.
  - Always, if component i waits for a response, it eventually receives it.
- No deadlock: $\Box\neg\forall i : pc_i = W$.
  - Never all components are simultaneously in a wait state W.
- Precedence: $\forall i : \Box(pc_i \neq C \Rightarrow (pc_i \neq C\ \mathbf{U}\ lock = i))$.
  - Always, if component i is out of the critical region, it stays out until it receives the shared lock variable (which it eventually does).
- Partial correctness: $\Box(pc = L \Rightarrow C)$.
  - Always if the program reaches line L, the condition C holds.
- Termination: $\forall i : \Diamond(pc_i = T)$.
  - Every component eventually terminates.

Classifying System Properties

- Safety Properties:
  - A safety property is a property such that, if it is violated by a run, it is already violated by some finite prefix of the run.
    - This finite prefix cannot be extended in any way to a complete run satisfying the property.
  - Example: $\Box F$.
    - The violating run $F \rightarrow F \rightarrow \neg F \rightarrow \ldots$ has the prefix $F \rightarrow F \rightarrow \neg F$ that cannot be extended in any way to a run satisfying $\Box F$.
- Liveness Properties:
  - A liveness property is a property such that every finite prefix can be extended to a complete run satisfying this property.
    - Only a complete run itself can violate that property.
  - Example: $\Diamond F$.
    - Any finite prefix p can be extended to a run $p \rightarrow F \rightarrow \ldots$ which satisfies $\Diamond F$.

Every system property P is a conjunction $S \wedge L$ of some safety property S and some liveness property L (both may be just "true").

Verifying Liveness

Example: verify that eventually some state property holds.

var $x := 0, y := 0$
loop || **loop**
$x := x + 1$ $\quad$ $y := y + 1$

$State = \mathbb{N} \times \mathbb{N}; Label = \{p, q\}.$
$I(x, y) :\Leftrightarrow x = 0 \wedge y = 0.$
$R(l, \langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow$
$(l = p \wedge x' = x + 1 \wedge y' = y) \vee (l = q \wedge x' = x \wedge y' = y + 1).$

- Prove $\langle I, R\rangle \models \Diamond x = 1$.
  - $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \ldots$
  - This run violates (as the only one) $\Diamond x = 1$.
  - Thus the system as a whole does not satisfy $\Diamond x = 1$.

For verifying liveness properties, "unfair" runs have to be ruled out.

Weak Fairness

- Weak Fairness
  - A run $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ is weakly fair to a transition l, if
    - if transition l is eventually permanently enabled in the run,
    - then transition l is executed infinitely often in the run.

      $(\exists i : \forall j \geq i : Enabled_R(l, s_j)) \Rightarrow (\forall i : \exists j \geq i : l_j = l).$
- LTL formulas may explicitly specify weak fairness constraints.
  - Let E_l denote the enabling condition of transition l.
  - Let X_l denote the predicate "transition l is executed".
  - Define $WF_l :\Leftrightarrow (\Diamond\Box E_l) \Rightarrow (\Box\Diamond X_l)$.

    If l is eventually enabled forever, it is executed infinitely often.
  - Prove $\langle I, S\rangle \models (WF_l \Rightarrow P)$.

    Property P is only proved for runs that are weakly fair to l.

A weak requirement to the fairness of a system.

Example

var $x := 0, y := 0$
loop || **loop**
$x := x + 1$ $\quad$ $y := y + 1$

$State = \mathbb{N} \times \mathbb{N}; Label = \{p, q\}.$
$I(x, y) :\Leftrightarrow x = 0 \wedge y = 0.$
$R(l, \langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow$
$(l = p \wedge x' = x + 1 \wedge y' = y) \vee (l = q \wedge x' = x \wedge y' = y + 1).$

- Prove $\langle I, R\rangle \models WF_p \Rightarrow \Diamond x = 1$.
  - Run $[x = 0, y = 0] \rightarrow [x = 0, y = 1] \rightarrow [x = 0, y = 2] \rightarrow \ldots$
  - Run is not weakly fair to p, need not be considered.

Violating run can be ruled out by demanding weak fairness.

1. A Client/Server System

2. Modeling Concurrent Systems

3. Specifying System Properties

4. Verifying System Properties

The Model Checker Spin

- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Workshop series since 1995 (12th workshop "Spin 2005").
  - ACM System Software Award in 2001.
- Spin resources:
  - Web site: http://spinroot.com.
  - Survey paper: Holzmann "The Model Checker Spin", 1997.
  - Book: Holzmann "The Spin Model Checker — Primer and Reference Manual", 2004.

Goal: verification of (concurrent/distributed) software models.

The Model Checker Spin

On-the-fly LTL model checking of finite state systems.

- System S modeled by automaton S_A.
  - Explicit representation of automaton states.
- On-the-fly model checking.
  - Reachable states of S_A are only expended on demand.
- LTL model checking.
  - Property P to be checked described in PLTL.
    - Propositional linear temporal logic.
  - Description converted into property automaton P_A.
    - Automaton accepts only system runs that do not satisfy the property.

Model checking based on automata theory.

The Spin System Architecture

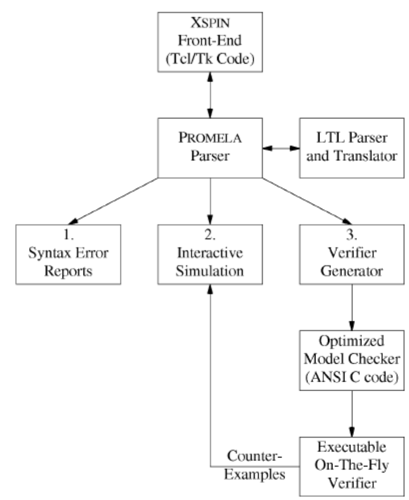


Fig. 1. The structure of SPIN simulation and verification.

Features of Spin

- System description in Promela.
  - Promela = Process Meta-Language.
    - Spin = Simple Promela Interpreter.
  - Express coordination and synchronization aspects of a real system.
  - Actual computation can be e.g. handled by embedded C code.
- Simulation mode.
  - Investigate individual system behaviors.
  - Inspect system state.
  - Graphical interface XSpin for visualization.
- Verification mode.
  - Verify properties shared by all possible system behaviors.
  - Properties specified in PLTL and translated to “never claims”.
    - Promela description of automaton for negation of the property.
  - Generated counter examples may be investigated in simulation mode.

Verification and simulation are tightly integrated in Spin.

The Client/Server System in Promela

```
/* definition of a constant MESSAGE */
mtype = { MESSAGE };

/* two arrays of channels of size 2,
   each channel has a buffer size 1 */
chan request[2] = [1] of { mtype };
chan answer [2] = [1] of { mtype };

/* the system of three processes */
init
{
  run client(1);
  run client(2);
  run server();
}
```

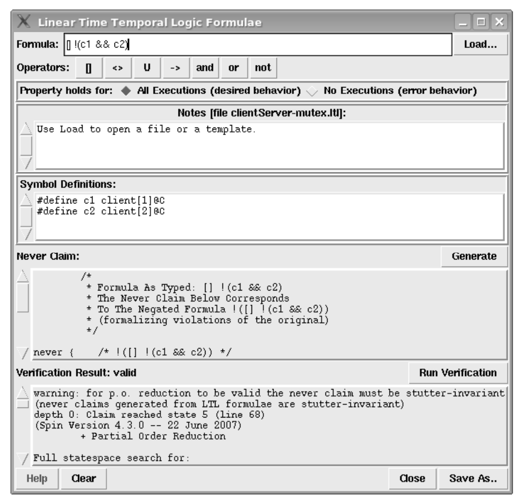
```
/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;
    W: answer[id-1]  ? MESSAGE;
    C: skip; // the critical region
    request[id-1] ! MESSAGE
  od;
}
```


The Client/Server System in Promela

```
/* the server process type */
proctype server()
{
  /* three variables of two bit each */
  unsigned given   : 2 = 0;
  unsigned waiting : 2 = 0;
  unsigned sender  : 2;

  do :: true ->

    /* receiving the message */
    R: if
    :: request[0] ? MESSAGE ->
      S1: sender = 1
    :: request[1] ? MESSAGE ->
      S2: sender = 2
    fi;
```

```
    /* answering the message */
    if
    :: sender == given ->
      if
      :: waiting == 0 ->
        given = 0
      :: else ->
        given = waiting;
        waiting = 0;
        answer[given-1] ! MESSAGE
      fi;
    :: given == 0 ->
      given = sender;
      answer[given-1] ! MESSAGE
    :: else
      waiting = sender
    fi;

  od;
}
```


Specifying a System Property



Formal specification of the “mutual exclusion” property.

Spin LTL

```
Grammar:
ltl ::= opd | ( ltl ) | ltl binop ltl | unop ltl

Operands (opd):
true, false, and user-defined names starting with a lower-case letter

Unary Operators (unop):
[] (the temporal operator always)
<> (the temporal operator eventually)
!  (the boolean operator for negation)

Binary Operators (binop):
U  (the temporal operator strong until)
V  (the dual of U): (p V q) means !(!p U !q))
&& (the boolean operator for logical and)
|| (the boolean operator for logical or)
/\ (alternative form of &&)
\/ (alternative form of ||)
-> (the boolean operator for logical implication)
<-> (the boolean operator for logical equivalence)
```


Spin Atomic Predicates

```
#define p (a > b)
#define q (len(q) < 5)
#define r (process@Label)
#define s (process[pid]@Label)
```

- PROMELA conditions with references to *global* system variables.
  - len(*q*): the number of messages in channel *q*.
  - *process@Label*: true if the execution of the process with process type *process* is in the state marked by *Label*.
  - *process*[*pid*]@*Label*: true if the execution of the process with type *process* and process identifier *pid* is in the state marked by *Label*.
    - First instantiated process receives process identifier 1.

Atomic predicates can describe arbitrary state conditions.

Spin Verification Output

```
(Spin Version 4.2.2 -- 12 December 2004)
+ Partial Order Reduction

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

State-vector 48 byte, depth reached 477, errors: 0
     499 states, stored
     395 states, matched
     894 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
...
0.00user 0.01system 0:00.01elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+737minor)pagefaults 0swaps
```


Verifying the System Property

```
(Spin Version 4.2.2 -- 12 December 2004)
+ Partial Order Reduction

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

State-vector 48 byte, depth reached 477, errors: 0
     499 states, stored
     395 states, matched
     894 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

...
```

No possible execution violates the "mutual exclusion" property.

The Implementation of Spin

Translation of the original problem to a problem in automata theory.

- Original problem: $S \models P$.
  - $S = \langle I, R \rangle$, PLTL formula P.
  - Does property P hold for every run of system S?
- Construct system automaton S_A with language $\mathcal{L}(S_A)$.
  - A language is a set of infinite words.
  - Each such word describes a system run.
  - $\mathcal{L}(S_A)$ describes the set of runs of S.
- Construct property automaton P_A with language $\mathcal{L}(P_A)$.
  - $\mathcal{L}(P_A)$ describes the set of runs satisfying P.
- Equivalent Problem: $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.
  - The language of S_A must be contained in the language of P_A.

There exists an efficient algorithm to solve this problem.

The Model Checking Algorithm

- Problem: $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$
  - Equivalent to: $\mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$.
    - Complement $\overline{L} := \{w : w \notin L\}$.
  - Equivalent to: $\mathcal{L}(S_A) \cap \mathcal{L}(\neg P_A) = \emptyset$.
    - $\overline{\mathcal{L}(A)} = \mathcal{L}(\neg A)$.
- Equivalent Problem: $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$.
  - Define the synchronized product automaton $A \otimes B$.
    - A transition of $A \otimes B$ represents a simultaneous transition of A and B.
  - Property: $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$.
- Final Problem: $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$.
  - We have to check whether the language of this automaton is empty.
  - We have to search for a word w accepted by this automaton.
    - If no such w exists, then $S \models P$.
    - If such a w exists, it identifies a counterexample run, i.e. a run r of S such that $r \not\models P$.

Complexity of the Search

The complexity of checking $S \models P$ is as follows.

- Let $|P|$ denote the number of subformulas of P.
- $|State_{(\neg P)_A}| = O(2^{|P|})$.
- $|State_{A \otimes B}| = |State_A| \cdot |State_B|$.
- $|State_{S_A \otimes (\neg P)_A}| = O(|State_{S_A}| \cdot 2^{|P|})$
- The time complexity of the search is linear in the size of the state space of the system automaton.
  - Actually, in the number of reachable states (typically much smaller).

PLTL model checking is linear in the number of reachable states but exponential in the size of the formula.

Verifying Concurrent Systems

Other approaches/tools to system verification exist.

- Model checking:
  - Symbolic Model Checking: SMV, NuSMV, ...
    - Systems and properties are modelled as binary decision diagrams (BDDs).
  - Bounded Model Checking: NuSMV2, ...
    - Model checking is reduced to checking the satisfiability of propositional formulas.
  - Counter-Example Guided Abstraction Refinement: BLAST, SLAM.
    - Abstract model is iteratively checked and refined; in principle also applicable to infinite state systems.
- Theorem proving:
  - Theorem proving assistants: PVS, Isabelle, Coq, the RISC ProofNavigator, ...
  - Not fully automatic, system invariants have to be established.
  - Complexity of verification independent of size of state space, also applicable to infinite state systems.