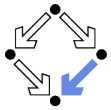
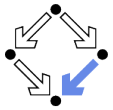
Model-Checking Concurrent Systems

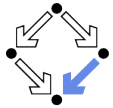
Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

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





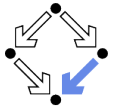
The Model Checker Spin



- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Symposium series since 1995 (31th symposium "Spin 2025").
  - 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.
  - There exist various other approaches (discussed later).
- On-the-fly model checking.
  - Reachable states of S_A are only expended on demand.
  - *Partial order reduction* to keep state space manageable.
- 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

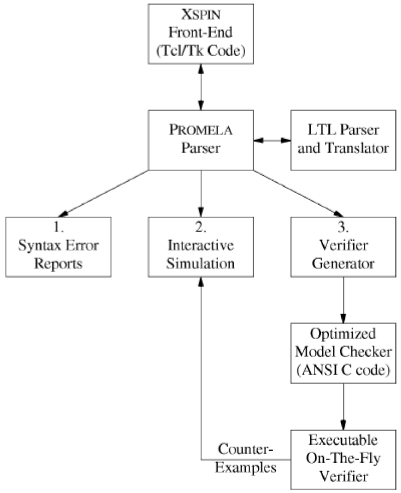


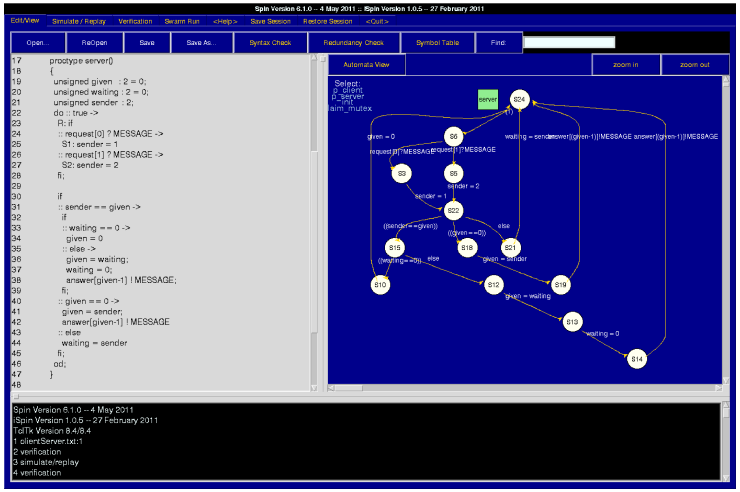
Fig. 1. The structure of SPIN simulation and verification.

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 `ispin` 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 Spin User Interface

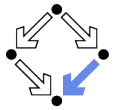


The Client/Server System in Promela

```
/* 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 mutual exclusion property */
ltl mutex { []!(client[1]@C && client[2]@C)}
```

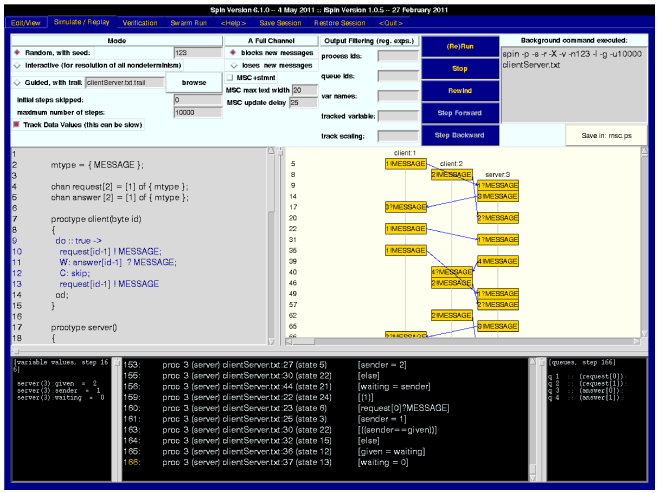
```
/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;
    W: answer[id-1]  ? MESSAGE;
    C: skip; // the critical region
    request[id-1] ! MESSAGE
  od;
}
```

The Client/Server System in Promela

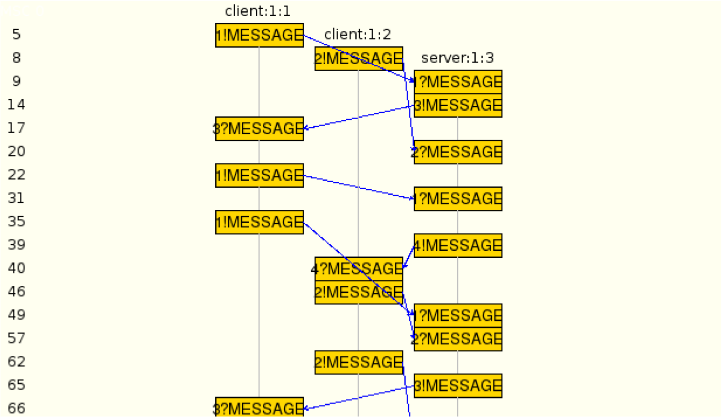


```
/* the server process type */
proctype server()
{
  /* three variables of two bit each */
  unsigned given   : 2 = 0;
  unsigned waiting : 2 = 0;
  unsigned sender  : 2;

  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;
}
```

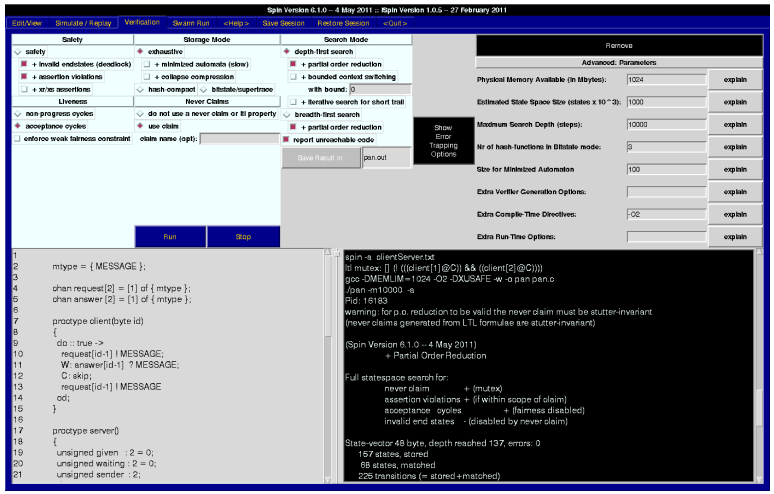

The Spin Simulation View



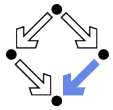
Simulating the System Execution in Spin



The Spin Verification View

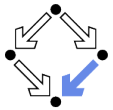


Specifying a System Property in Spin



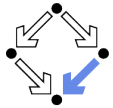
```
The easiest way to specify an LTL property is to specify it inline.
The formula is specified globally (i.e., outside all proctype or
init declarations) with the following syntax:

        ltl [ name ] '{'  formula '}'

... The names of operators can either be abbreviated with the symbols
shown above, or spelled out in full (as always, eventually, until,
implies, and equivalent. The alternative operators weakuntil, stronguntil,
and release (for the V operator, see above), are also supported. This
means that the following two are equivalent:

        ltl p1 { []<> p }
        ltl p2 { always eventually p }
        ltl p3 { eventually (a > b) implies len(q) == 0 }

There is just one restriction: you cannot use the predefined operators
empty, nempty, full, nfull in inline ltl formula...
```


Spin LTL



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

Operands (opd):
 true, false, user-defined names starting with a lower-case letter,
 or embedded expressions inside curly braces, e.g.,: { a+b>n }.

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)
 W  (the temporal operator weak 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

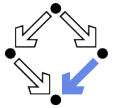


```
(a+b > c)
(len(q) < 5)
(process@Label)
(process[pid]@Label)
```

- PROMELA conditions with references to *global* system variables.
  - C-like boolean expressions.
  - 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*.
    - Process `init` receives identifier 0.
    - Processes instantiated with `run` receive identifiers $1, 2, \ldots$.

Atomic predicates can describe arbitrary state conditions.

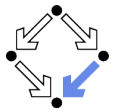
The Spin Verification Options



- Verification mode
  - Safety: predefined safety properties.
  - Liveness: predefined liveness properties.
  - Use Claim: user-defined (LTL) properties.
    - Needs liveness option acceptance cycles.
- Storage mode
  - Exhaustive: full error search.
    - May require (too) much memory.
  - Hash-compact and bitstate/supertrace: probabilistic search.
    - Less memory consuming but not guaranteed to find all errors.
    - Number of reported hash collisions should not be too high.
- Advanced Parameters
  - Physical Memory Size: increase, if checker runs out of memory.
  - Maximum Search Depth: reduce to find shorter counterexamples.

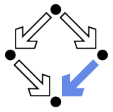
By default, select "acceptance cycles", "use claim", and "exhaustive".

The Spin Verification Output



```
(Spin Version 6.1.0 -- 4 May 2011)
 + Partial Order Reduction

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

State-vector 48 byte, depth reached 137, errors: 0
      157 states, stored
       68 states, matched
      225 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
...
pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

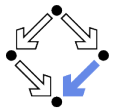

More Promela Features



Active processes, inline definitions, atomic statements, output.

```
mtype = { P, C, N }
mtype turn = P;

inline request(x, y) { atomic { x == y -> x = N } }
inline release(x, y) { atomic { x = y } }
#define FORMAT "Output: %c\n"

active proctype producer()
{
  do
  :: request(turn, P) -> printf(FORMAT, 'P'); release(turn, C);
  od
}

active proctype consumer()
{
  do
  :: request(turn, C) -> printf(FORMAT, 'C'); release(turn, P);
  od
}
```

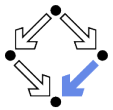

More Promela Features



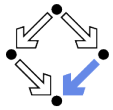
Embedded C code.

```
/* declaration is added locally to proctype main */
c_state "float f" "Local main"

active proctype main()
{
  c_code { Pmain->f = 0; }
  do
    :: c_expr { Pmain->f <= 300 };
       c_code { Pmain->f = 1.5 * Pmain->f ; };
       c_code { printf("%4.0f\n", Pmain->f); };
  od;
 }
```

Can embed computational aspects into a Promela model (only works in verification mode where a C program is generated from the model).



The Basic Approach

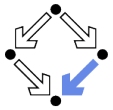


Translation of the original problem to a problem in automata theory.

- 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.

Finite State Automata

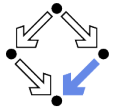


A (variant of a) labeled transition system in a finite state space.

- Take finite sets *State* and *Label*.
  - The state space *State*.
  - The alphabet *Label*.
- A (finite state) automaton $A = \langle I, R, F \rangle$ over *State* and *Label*:
  - A set of initial states $I \subseteq State$.
  - A labeled transition relation $R \subseteq Label \times State \times State$.
  - A set of final states $F \subseteq State$.
    - Büchi automata: F is called the set of accepting states.

We will only consider infinite runs of Büchi automata.

Runs and Languages



- An infinite run $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \ldots$ of automaton A:
  - $s_0 \in I$ and $R(l_i, s_i, s_{i+1})$ for all $i \in \mathbb{N}$.
  - Run r is said to read the infinite word $w(r) := \langle l_0, l_1, l_2, \ldots \rangle$.
- $A = \langle I, R, F \rangle$ accepts an infinite run r:
  - Some state $s \in F$ occurs infinitely often in r.
  - This notion of acceptance is also called Büchi acceptance.
- The language $\mathcal{L}(A)$ of automaton A:
  - $\mathcal{L}(A) := \{w(r) : A \text{ accepts } r\}$.
  - The set of words which are read by the runs accepted by A.
- Example: $\mathcal{L}(A) = (a^*bb^*a)^*a^\omega + (a^*bb^*a)^\omega \; [= (b^*a)^\omega]$.
  - $w^i = ww \ldots w$ (i occurrences of w).
  - $w^* = \{w^i : i \in \mathbb{N}\} = \{\langle\rangle, w, ww, www, \ldots\}$.
  - $w^\omega = wwww \ldots$ (infinitely often).
  - An infinite repetition of an arbitrary number of b followed by a.

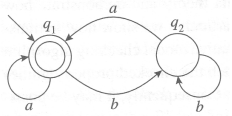
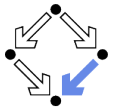


Figure 9.1
A finite automaton.

Edmund Clarke: "Model Checking", 1999.

A Finite State System as an Automaton



The automaton $S_A = \langle I, R, F \rangle$ for a finite state system $S = \langle I_S, R_S \rangle$:

- $State := State_S \cup \{\iota\}$.
  - The state space $State_S$ of S is finite; additional state ι ("iota").
- $Label := \mathbb{P}(AP)$.
  - Finite set AP of atomic propositions.
    All PLTL formulas are built from this set only.
  - Powerset $\mathbb{P}(S) := \{s : s \subseteq S\}$.
  - Every element of *Label* is thus a set of atomic propositions.
- $I := \{\iota\}$.
  - Single initial state ι.
- $R(l, s, s') :\Leftrightarrow l = L(s') \wedge (R_S(s, s') \vee (s = \iota \wedge I_S(s')))$.
  - $L(s) := \{p \in AP : s \models p\}$.
  - Each transition is labeled by the set of atomic propositions satisfied by the successor state.
- $F := State$.
  - Every state is accepting.

A Finite State System as an Automaton

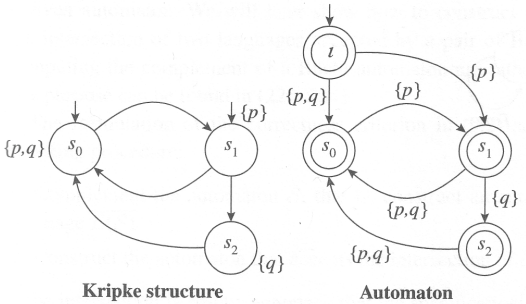


Figure 9.2
Transforming a Kripke structure into an automaton.

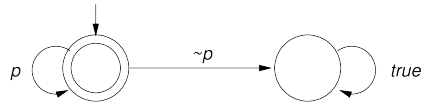
Edmund Clarke et al: "Model Checking", 1999.

If $r = s_0 \to s_1 \to s_2 \to \ldots$ is a run of S, then S_A accepts the labelled version $r_l := \iota \overset{L(s_0)}{\to} s_0 \overset{L(s_1)}{\to} s_1 \overset{L(s_2)}{\to} s_2 \overset{L(s_3)}{\to} \ldots$ of r.

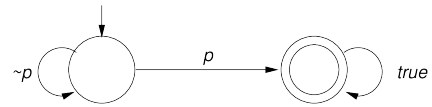
A System Property as an Automaton

Also an PLTL formula can be translated to a finite state automaton.

- We need the automaton P_A for a PLTL property P.
  - Requirement: $r \models P \Leftrightarrow P_A$ accepts r_l.
  - A run satisfies property P if and only if automaton A_P accepts the labeled version of the run.
- Example: $\Box p$.

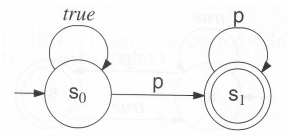


- Example: $\Diamond p$.



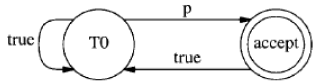
Further Examples

- Example: $\Diamond\Box p$.



Gerard Holzmann: "The Spin Model Checker", 2004.

- Example: $\Box\Diamond p$.



Gerard Holzmann: "The Model Checker Spin", 1997.

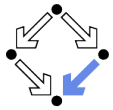
Arbitrary PLTL formulas can be converted to automata (we omit the details).

System Properties

- State equivalence: $L(s) = L(t)$.
  - Both states have the same labels.
  - Both states satisfy the same atomic propositions in AP.
- Run equivalence: $w(r_l) = w(r'_l)$.
  - Both runs have the same sequences of labels.
  - Both runs satisfy the same PLTL formulas built over AP.
- Indistinguishability: $w(r_l) = w(r'_l) \Rightarrow (r \models P \Leftrightarrow r' \models P)$
  - PLTL formula P cannot distinguish between runs r and r' whose labeled versions read the same words.
- Consequence: $S \models P \Leftrightarrow \mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.
  - If every run of S satisfies P, then every word $w(r_l)$ in $\mathcal{L}(S_A)$ equals some word $w(r'_l)$ in $\mathcal{L}(P_A)$, and vice versa.

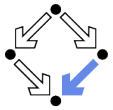
A skeleton of the correctness proof of the problem reduction.

The Next Steps



- 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: $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$.
    - $\overline{\mathcal{L}(P_A)} = \mathcal{L}((\neg P)_A)$.
- Equivalent Problem: $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$.
  - We will introduce 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 look for a word w accepted by this automaton.
    - If no such w exists, then $S \models P$.
    - If such a $w = w(r_l)$ exists, then r is a counterexample, i.e. a run of S such that $r \not\models P$.

Synchronized Product of Two Automata

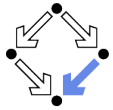


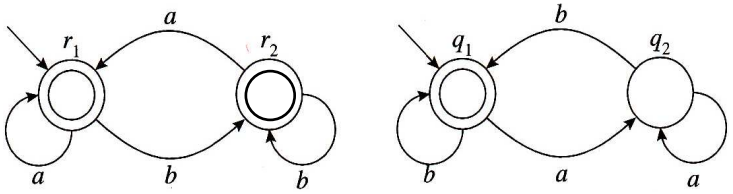
Given two finite automata $A = \langle I_A, R_A, State_A \rangle$ and $B = \langle I_B, R_B, F_B \rangle$.

- Synchronized product $A \otimes B = \langle I, R, F \rangle$.
  - $State := State_A \times State_B$.
  - $Label := Label_A = Label_B$.
  - $I := I_A \times I_B$.
  - $R(I, \langle s_A, s_B \rangle, \langle s'_A, s'_B \rangle) :\Leftrightarrow R_A(I, s_A, s'_A) \wedge R_B(I, s_B, s'_B)$.
  - $F := State_A \times F_B$.

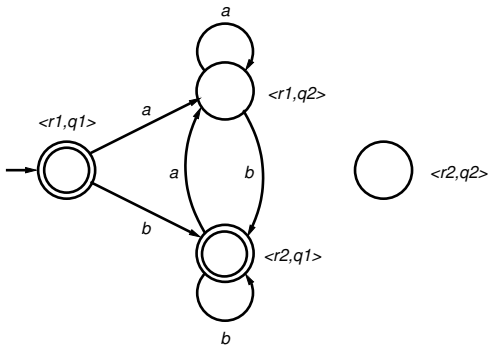
Special case where all states of automaton A are accepting.

Synchronized Product of Two Automata

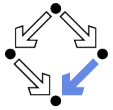




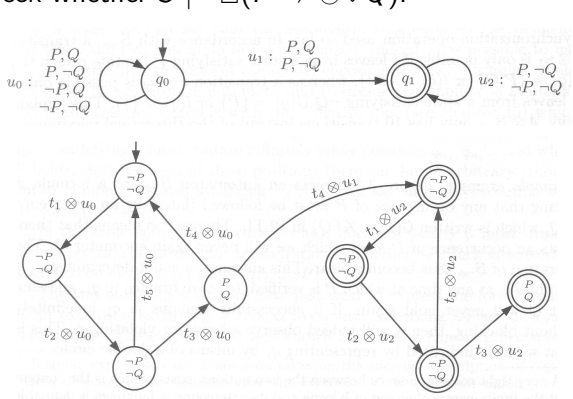
Edmund Clarke: "Model Checking", 1999.

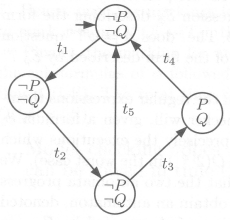


Example



Check whether $S \models \Box(P \Rightarrow \bigcirc\Diamond Q)$.

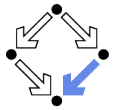




B. Berard et al: "Systems and Software Verification", 2001.

The product automaton accepts a run, thus the property does not hold (shown S has no ι but t_i labeled with properties of predecessor state).

Checking Emptiness



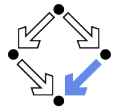
How to check whether $\mathcal{L}(A)$ is non-empty?

- If $\mathcal{L}(A)$ is non-empty, A accepts some run r.
  - r represents a counterexample for the property to be checked.
- Since r is accepted, it contains infinitely many occurrences of some accepting state s.

  $r = \iota \to \ldots \to s \to \ldots \to s \to \ldots \to s \to \ldots$
- Since the state space is finite, r must contain a cycle $s \to \ldots \to s$.
  - Finite prefix $\iota \to \ldots \to s$.
  - Infinite repetition of cycle $s \to \ldots \to s$.
- We have to search for such an acceptance cycle.
  - An accepting state s that is reachable from itself.

The search for an acceptance cycle in the reachability graph is the core problem of PLTL model checking; it can be solved by *depth-first search*.

Basic Structure of Depth-First Search



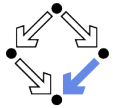
Visit all states of the reachability graph of an automaton $\langle \{\iota\}, R, F \rangle$.

```
global
  StateSpace V := {}
  Stack D := ⟨⟩

proc main()
  push(D, ι)
  visit(ι)
  pop(D)
end
```

```
proc visit(s)
  V := V ∪ {s}
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      visit(s')
      pop(D)
    end
  end
end
```

State space V holds all states visited so far; stack D holds path from initial state to currently visited state.

Searching for a State



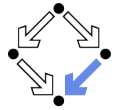
Algorithm to check violation of assertion f (i.e., $\Box f$ with state formula f).

```
global
  StateSpace V := {}
  Stack D := ⟨⟩

proc main()
  // r becomes true, iff
  // counterexample run is found
  push(D, ι)
  r := search(ι)
  pop(D)
end
```

```
function search(s)
  V := V ∪ {s}
  if s ⊭ f then
    print D
    return true
  end
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      r := search(s')
      pop(D)
      if r then return true end
    end
  end
  return false
end
```

Stack D can be used to print counterexample run.

Searching for an Acceptance Cycle

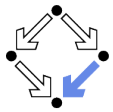


Algorithm to check whether a general temporal formula is violated.

```
global
  StateSpace V := {}
  Stack D := ⟨⟩
  Stack C := ⟨⟩

function searchCycle(s)
  for ⟨l, s, s'⟩ ∈ R do
    if has(D, s') then
      print D; print C; print s'
      return true
    else if ¬has(C, s') then
      push(C, s');
      r := searchCycle(s')
      pop(C);
      if r then return true end
    end
  end
  return false
end
```

```
proc main()
  push(D, ι); r := search(ι); pop(D)
end

boolean search(s)
  V := V ∪ {s}
  for ⟨l, s, s'⟩ ∈ R do
    if s' ∉ V
      push(D, s')
      r := search(s')
      pop(D)
      if r then return true end
    end
  end
  if s ∈ F then
    r := searchCycle(s)
    if r then return true end
  end
  return false
end
```

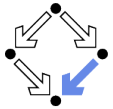
Depth-First Search for Acceptance Cycle



- At each call of *search*(s),
  - s is a reachable state,
  - D describes a path from ι to s.
- *search* calls *searchCycle*(s)
  - on a reachable accepting state s
  - in order to find a cycle from s to itself.
- At each call of *searchCycle*(s),
  - s is a state reachable from a reachable accepting state s_a,
  - D describes a path from ι to s_a,
  - $D \rightarrow C \rightarrow s$ describes a path from ι to s (via s_a).
- Thus we have found an accepting cycle $D \rightarrow C \rightarrow s'$, if
  - there is a transition $s \xrightarrow{l} s'$,
  - such that s' is contained in D.

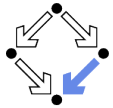
If the algorithm returns "true", there exists a violating run; the converse follows from the exhaustiveness of the search.

Implementing the Search



- The state space V,
  - is implemented by a hash table for efficiently checking $s' \notin V$.
- Rather than using explicit stacks D and C,
  - each state node has two bits d and c,
  - d is set to denote that the state is in stack D,
  - c is set to denote that the state is in stack C.
- The counterexample is printed,
  - by searching, starting with ι, the unique sequence of reachable nodes where d is set until the accepting node s_a is found, and
  - by searching, starting with a successor of s_a, the unique sequence of reachable nodes where c is set until the cycle is detected.
- Furthermore, it is not necessary to reset the c bits, because
  - *search* first explores all states reachable by an accepting state s before trying to find a cycle from s; from this, one can show that
  - called with first accepting node s_a reachable from itself, *searchCycle* needs not revisit nodes with c bits set in previous searches.
  - With this improvement, every state is only visited once.

Complexity of the Search

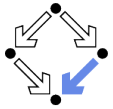


The complexity of checking $S \models P$ is as follows.

- 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 *search* is linear in the size of *State*.
  - Actually, in the number of reachable states (typically much smaller).
  - Only true for the improved variant where the c bits are not reset.
  - Then every state is visited by *searchCycle* at most once.

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

The Overall Process

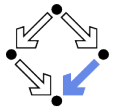


Basic PLTL model checking for deciding $S \models P$.

- Convert system S to automaton S_A.
  - Atomic propositions of PLTL formula are evaluated on each state.
- Convert negation of PLTL formula P to automaton $(\neg P)_A$.
  - How to do so, we have not yet described.
- Construct synchronized product automaton $S_A \otimes (\neg P)_A$.
  - After that, formula labels are not needed any more.
- Find acceptance cycle in reachability-graph of product automaton.
  - A purely graph-theoretical problem that can be efficiently solved.
  - Time complexity is linear in the size of the state space of the system but exponential in the size of the formula to be checked.
  - Weak scheduling fairness with k components: runtime is increased by factor $k + 2$ (worst-case, "in practice just factor 2" [Holzmann]).

The basic approach immediately leads to *state space explosion*; further improvements are needed to make it practical.

On the Fly Model Checking

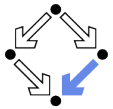


For checking $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$, it is not necessary to construct the states of S_A in advance.

- Only the property automaton $(\neg P)_A$ is constructed in advance.
  - This automaton has comparatively small state space.
- The system automaton S_A is constructed on the fly.
  - Construction is guided by $(\neg P)_A$ while computing $S_A \otimes (\neg P)_A$.
  - Only that part of the reachability graph of S_A is expanded that is consistent with $(\neg P)_A$ (i.e. can lead to a counterexample run).
- Typically only a part of the state space of S_A is investigated.
  - A smaller part, if a counterexample run is detected early.
  - A larger part, if no counterexample run is detected.

Unreachable system states and system states that are not along possible counterexample runs are never constructed.

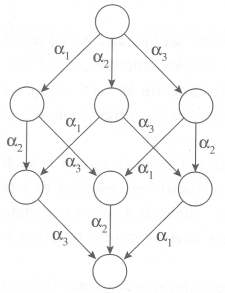
Partial Order Reduction



Core problem of model checking: state space explosion.

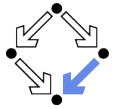
- Take asynchronous composition $S_0||S_1||\ldots||S_{k-1}$.
  - Take state s where one transition of each component is enabled.
    - Assume that the transition of one component does not disable the transitions of the other components and that no other transition becomes enabled before all the transitions have been performed.
  - Take state s' after execution of all the transitions
    - There are $k!$ paths leading from s to s'.
    - There are 2^k states involved in the transitions.

Sometimes it suffices to consider a *single path* with $k+1$ states.

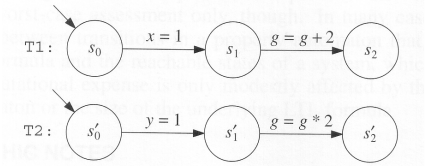


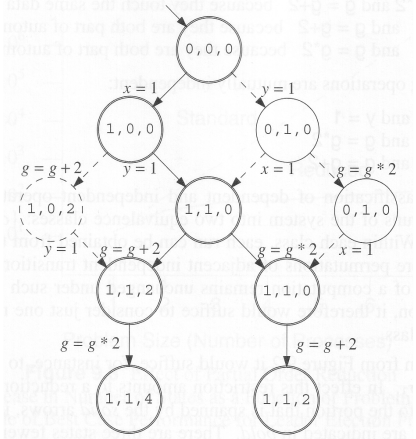
Edmund Clarke: "Model Checking", 1999.

Example



Check $(T1||T2) \models \Diamond g \geq 2$.

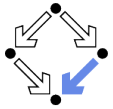




Gerard Holzmann: "The Spin Model Checker", 1999.

For checking $\Diamond g \geq 2$, it suffices to check only one ordering of the independent transitions $x = 1$ and $y = 1$ (not true for checking $\Box x \geq y$).

Example



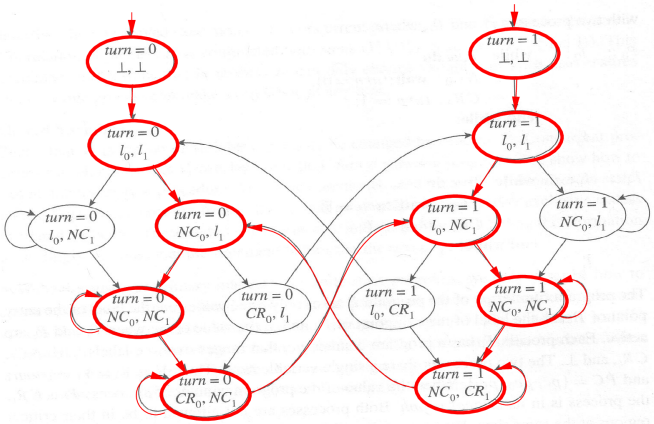
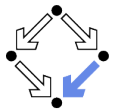


Figure 2.2
Reachable states of Kripke structure for mutual exclusion example.

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

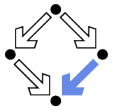
System after partial order reduction.

Other Optimizations



- Statement merging.
  - Special case of partial order reduction where a sequence of transitions of same component is combined to a single transition.
- State compression.
  - Collapse compression: each state holds pointers to component states; thus component states can be shared among many system states.
  - Minimized automaton representation: represent state set V not by hash table but by finite state automaton that accepts a state (sequence of bits) s if and only if $s \in V$.
  - Hash compact: store in the hash table a hash value of the state (computed by a different hash function). Probabilistic approach: fails if two states are mapped to the same hash value.
  - Bitstate hashing: represent V by a bit table whose size is much larger than the expected number of states; each state is then only represented by a single bit. Probabilistic approach: fails if two states are hashed to the same position in the table.

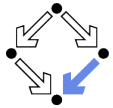
Other Approaches to Model Checking



There are fundamentally different approaches to model checking than the automata-based one implemented in Spin.

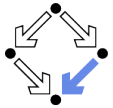
- Symbolic Model Checking (e.g. SMV, NuSMV).
  - Core: binary decision diagrams (BDDs).
    - Data structures to represent boolean functions.
    - Can be used to describe state sets and transition relations.
  - The set of states satisfying a CTL formula P is computed as the BDD representation of a fixpoint of a function (predicate transformer) F_P.
    - If all initial system states are in this set, P is a system property.
  - BDD packages for efficiently performing the required operations.
- Bounded Model Checking (e.g. NuSMV2).
  - Core: propositional satisfiability.
    - Is there a truth assignment that makes propositional formula true?
  - There is a counterexample of length at most k to a LTL formula P, if and only if a particular propositional formula $F_{k,P}$ is satisfiable.
    - Problem: find suitable bound k that makes method complete.
  - SAT solvers for efficiently deciding propositional satisfiability.

Other Approaches to Model Checking

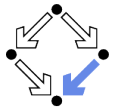


- Counter-Example Guided Abstraction Refinement (e.g. BLAST).
  - Core: model abstraction.
    - A finite set of predicates is chosen and an abstract model of the system is constructed as a finite automaton whose states represent truth assignments of the chosen predicates.
  - The abstract model is checked for the desired property.
    - If the abstract model is error-free, the system is correct; otherwise an abstract counterexample is produced.
    - It is checked whether the abstract counterexample corresponds to a real counterexample; if yes, the system is not correct.
    - If not, the chosen set of predicates contains too little information to verify or falsify the program; new predicates are added to the set. Then the process is repeated.
  - Core problem: how to refine the abstraction.
    - Automated theorem provers are applied here.

Various model checkers for software verification use this approach.



Model Checking in RISCAL



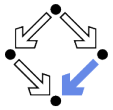
Also RISCAL includes an LTL model checker.

```
shared system clientServer
{
  ...
  // the safety property
  ltl □ [[∀i1:Client,i2:Client with i1 ≠ N ∧ i2 ≠ N ∧ i1 < i2.
           ¬(pc[i1] = C ∧ pc[i2] = C)]];

  // the liveness property
  ltl[fairness] ∀i:Client with i ≠ N.
                  □ ([[pc[i] = S]] ⇒ (◇[[pc[i] = C]]));
  ...
}
```

- Linear temporal logic with quantifiers.
- State formulas (can be also quantified) embedded by [[ ]].

Builtin mechanism for checking liveness properties with fairness.

Model Checking in RISCAL

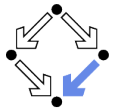


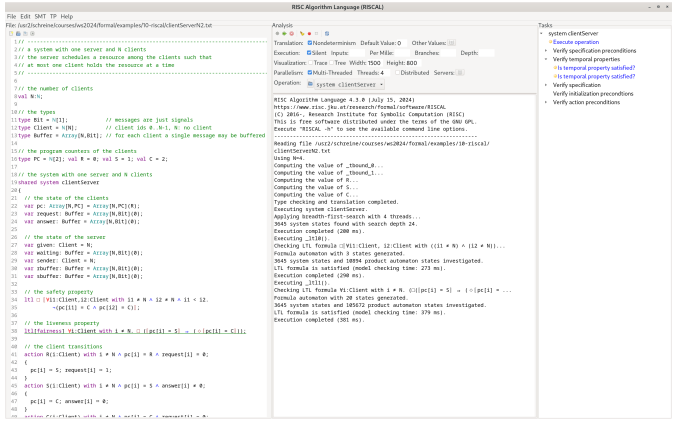
Fairness annotations of individual actions.

```
action D(i:Client) with i ≠ N ∧ sender = N ∧ rbuffer[i] ≠ 0;
fairness strong_all; // strongly fairly accept request from every client
{ sender := i; rbuffer[i] := 0; }

action A1(i:Client) with i ≠ N ∧
  sender ≠ N ∧ sender = given ∧ waiting[i] ≠ 0 ∧
  sbuffer[i] = 0;
fairness strong_all; // treat every waiting client strongly fairly
{ given := i; waiting[i] = 0; sbuffer[given] := 1; sender := N; }

action REQ(i:Client) with i ≠ N ∧ request[i] ≠ 0 ∧ rbuffer[i] = 0;
fairness strong_all; // strongly fairly forward request from every client
{ request[i] := 0; rbuffer[i] := 1; }
```

Allows much more efficient checking than by encoding fairness in LTL.

Model Checking in RISCAL





Checking with $N = 4$ clients quickly succeeds (state space explodes with LTL encoding of fairness, also overwhelms Spin).