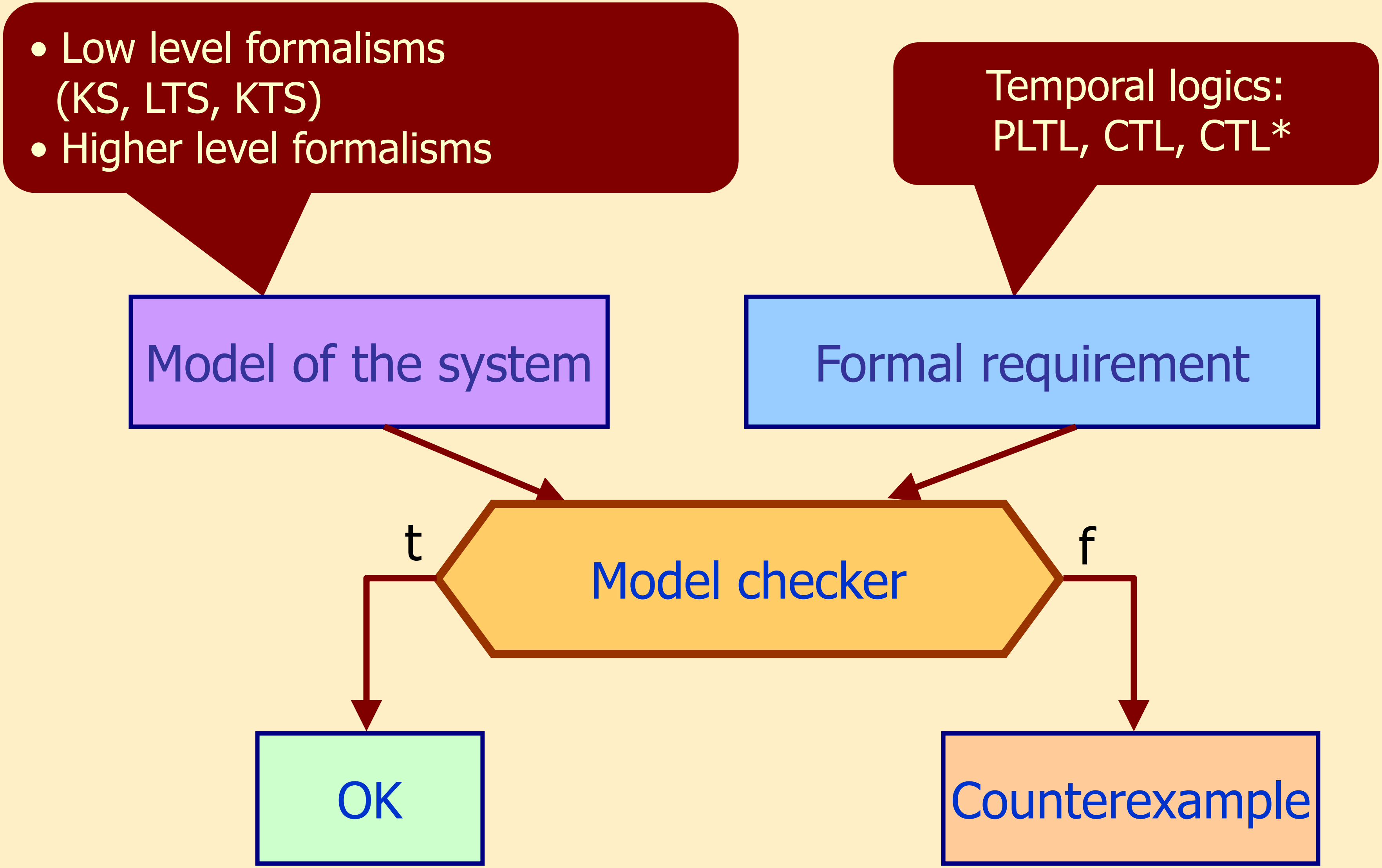
Efficient Techniques for Model Checking: Bounded Model Checking

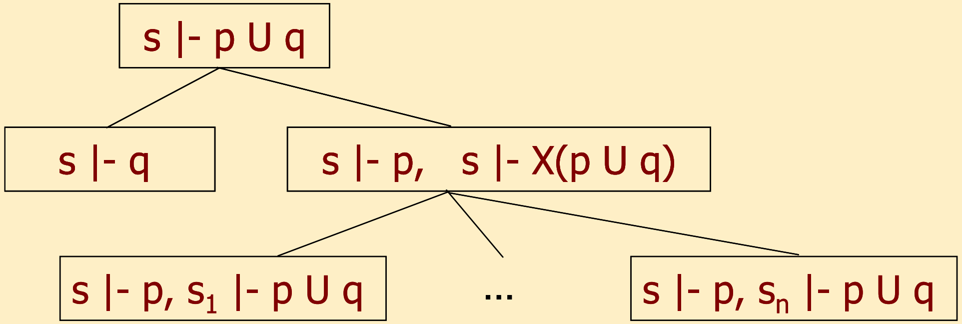
dr. István Majzik
BME Department of Measurement and Information Systems

Where are we now?

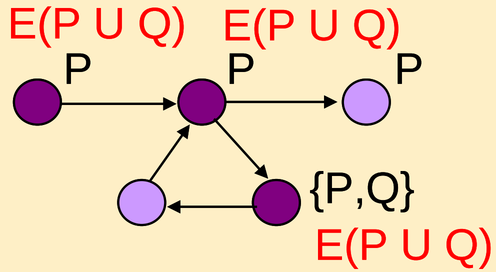


Recap: presented techniques for model checking

- LTL model checking:
  - Semantic tableaux: decomposing formulas based on the model



  - Automata theoretic approach (supplementary material)
- CTL model checking:
  - Labeling: iterative labeling of states



Overview of the presented techniques

- CTL model checking: symbolic technique

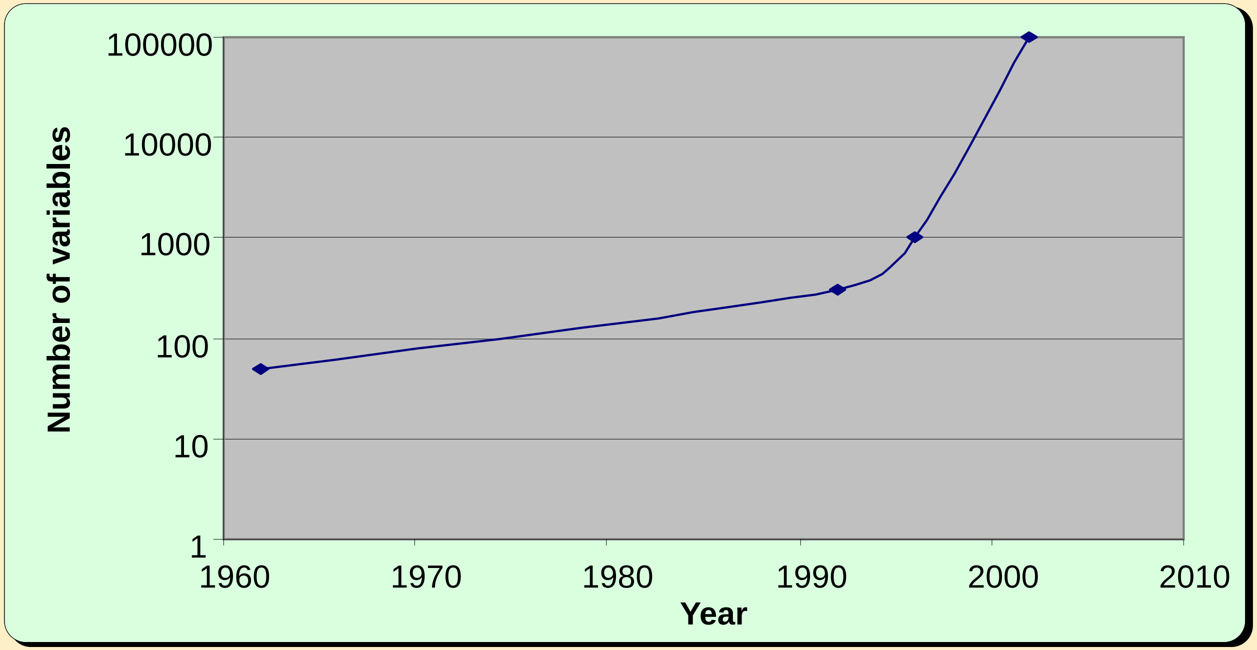
Semantics-based tehcnique	Symbolic technique
Sets of labeled states	Characteristic functions (Boolean functions): ROBDD representation
Operations over sets	Efficient operations over ROBDD

- Model checking invariants: Bounded model checking
  - Satisfiability checking for Boolean formulas with a SAT solver
  - Model checking up to a given bound:
Searching for counterexamples within a bounded length
    - A counterexample is a valid counterexample
    - If no counterexample is found, it is only a partial result

Bounded Model Checking

SAT solvers

- SAT solver:
  - Searches for a model –
    a variable assignment that makes the formula true
    Example: bitvector (1,1,0) for formula $f(x_1,x_2,x_3)=x_1 \wedge x_2 \wedge \neg x_3$
- NP-complete, but efficient algorithms exist
  - zChaff, MiniSAT, …



Goal

- Reducing the problem to a suitable problem in SAT
  - Model and temporal logic property together
    - Typically invariant properties:
      condition on all reachable states
- Using a SAT solver for model checking
  - If the property holds the SAT solver
    finds no model for the formula
  - If the property fails the model found by the SAT solver
    induces a counterexample
    - The counterexample can be used for debugging
    - An efficient technique for invariant properties

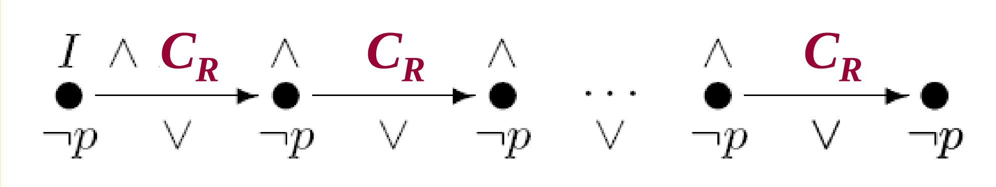
The basics of bounded model checking

- We do not handle the state space all in one
- We perform checking by restricting the length of paths
  - Partial verification: checking only up to a given bound in path length
  - The bound can be iteratively increased
  - In certain cases, the state space has a diameter – the length of the longest loop-free path
- The bound can be estimated:
  - Based on intuition about the problem
  - Based on worst-case execution time

Informal introduction

- How do we describe a path?
  - Starting from the initial states: characteristic function I(s)
  - „Unrolling”: along potential transitions
    - Transition relation (where can we progress): characteristic function $C_R(s,s')$
    - Transition between s and s': $C_R(s,s')$
    - Transition between s' and s'': $C_R(s',s'')$
    - …
    - Simpler notation: Upper index instead of primes: $C_R(s^0,s^1)$, $C_R(s^1,s^2)$ …
- How do we describe the property?
  - Invariant: condition on all states – a predicate p(s)
- The characterization of a counterexample (with conjunction):
  - Starting from the initial state: I(s)
  - „Stepping” along the transition relation: $C_R(s,s')$
  - To a counterexample (somewhere p(s) fails): $\neg p(s)$ disjunction on states of the path

A model of this formula corresponds to a counterexample!



Notations

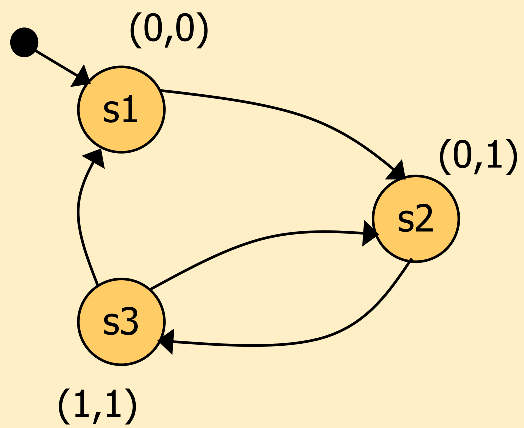
- Kripke structure M=(S,R,L)
- Logical formulas:
  - I(s): the characteristic formula of initial states in n variables
    - Background: Encoding states with a bit vector of length n
  - $C_R(s,s')$: the characteristic formula of transitions in 2n variables
    - The individual transitions are combined with disjunction
  - path(): characteristic function of paths of length k in (k+1)n variables

$$\mathrm{path}(s^0, s^1, \ldots, s^k) = \bigwedge_{0 \le i < k} C_R(s^i, s^{i+1})$$

Upper indices instead of primes

  - p(s): characteristic function of the property
    - Based on the labeling L
    - In general: can be constructed based on the state variables

Examples: encoding a model

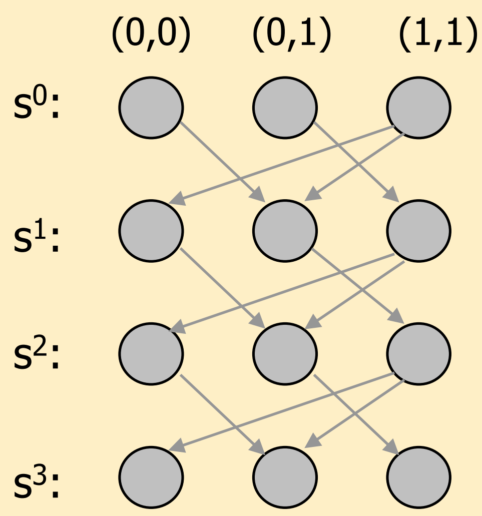


Initial states:

$I(x,y) = (\neg x \wedge \neg y)$

Transition relation:

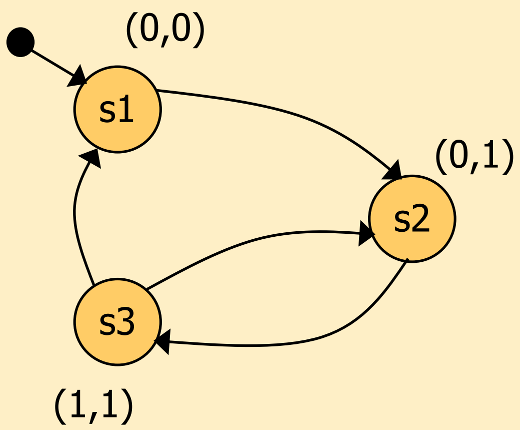
$$\begin{aligned} C_R(x,y,x',y') = & (\neg x \wedge \neg y \wedge \neg x' \wedge y') \vee \\ & \vee (\neg x \wedge y \wedge x' \wedge y') \vee \\ & \vee (x \wedge y \wedge \neg x' \wedge y') \vee \\ & \vee (x \wedge y \wedge \neg x' \wedge \neg y') \end{aligned}$$



Unrolling for 3 steps from the initial states:

$$\begin{aligned} I(x^0,y^0) \wedge path(s^0,s^1,s^2,s^3) = \\ = I(x^0,y^0) \wedge \\ C_R(x^0,y^0, x^1,y^1) \wedge \\ C_R(x^1,y^1, x^2,y^2) \wedge \\ C_R(x^2,y^2, x^3,y^3) \end{aligned}$$

Examples: encoding a model

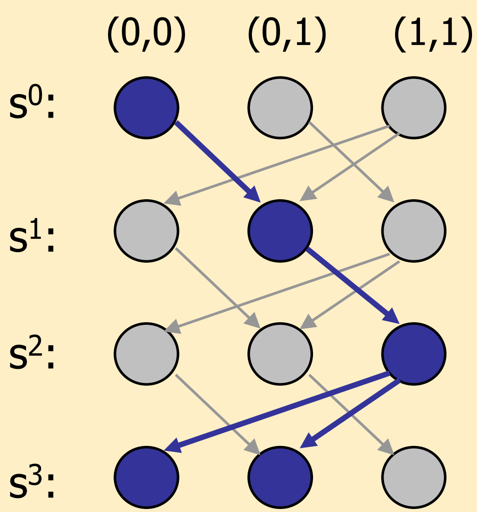


Initial states:

$I(x,y) = (\neg x \wedge \neg y)$

Transition relation:

$$
\begin{aligned}
C_R(x,y,x',y') = & (\neg x \wedge \neg y \wedge \neg x' \wedge y') \vee \\
& \vee (\neg x \wedge y \wedge x' \wedge y') \vee \\
& \vee (x \wedge y \wedge \neg x' \wedge y') \vee \\
& \vee (x \wedge y \wedge \neg x' \wedge \neg y')
\end{aligned}
$$



Unrolling for 3 steps from the initial states:

$$
\begin{aligned}
I(x^0,y^0) \wedge path(s^0,s^1,s^2,s^3) = \\
= I(x^0,y^0) \wedge \\
C_R(x^0,y^0, x^1,y^1) \wedge \\
C_R(x^1,y^1, x^2,y^2) \wedge \\
C_R(x^2,y^2, x^3,y^3)
\end{aligned}
$$

Formalizing the problem

- Invariant p(s) to prove: Each path from the initial states ends in a state where p(s) holds

$$\forall i:\ \forall s^0, s^1, \ldots, s^i:\ (I(s^0) \wedge \mathrm{path}(s^0, s^1, \ldots, s^i) \Rightarrow\ p(s^i))$$

- If p(s) fails at some point then there exists an i such that the followng formula is satisfiable:

$$I(s^0) \wedge \mathrm{path}(s^0, s^1, \ldots, s^i) \wedge \neg p(s^i)$$

  The model can be found by the SAT solver!
  - That is, values for the (i+1)·n variables that define the path $(s^0,s^1,\ldots,s^i)$

- First idea: for i=0,1,2,..., check whether for a path of length i the following formula can hold:

$$I(s^0) \wedge \mathrm{path}(s^0, s^1, \ldots, s^i) \wedge \neg p(s^i)$$

Elements of the algorithm

- Iteration: i=0,1,2,... on the length of paths
- We are investigating loop free paths: lfpath

Expressed in terms of the state variables

$$\text{lfpath}(s^0, s^1, \ldots, s^k) = \text{path}(s^0, s^1, \ldots, s^k) \wedge \bigwedge_{0 \le i < j \le k} s^i \ne s^j$$

- Termination condition during the iteration:
  - There is no loop free path with length i from the initial state, that is, the following is not satisfied

$$I(s^0) \wedge \text{lfpath}(s^0, s^1, \ldots, s^i)$$

  - There is no loop free path with length i (from anywhere) to a bad state (where p(s) fails), that is, the following is not satisfied

$$\text{lfpath}(s^0, s^1, \ldots, s^i) \wedge \neg p(s^i)$$

- If the iteration stops, then p(s) holds invariably

The algorithm

$i = 0$

while True do

  if not $\text{SAT}(I(s^0) \wedge \text{lfpath}(s^0, s^1, \ldots, s^i))$

    or not $\text{SAT}((\text{lfpath}(s^0, s^1, \ldots, s^i) \wedge \neg p(s^i))$

    then return True

  if $\text{SAT}(I(s^0) \wedge \text{path}(s^0, s^1, \ldots, s^i) \wedge \neg p(s^i))$

    then return $(s^0, s^1, \ldots, s^i)$

  $i = i + 1$

end

No more loop free paths from the initial states

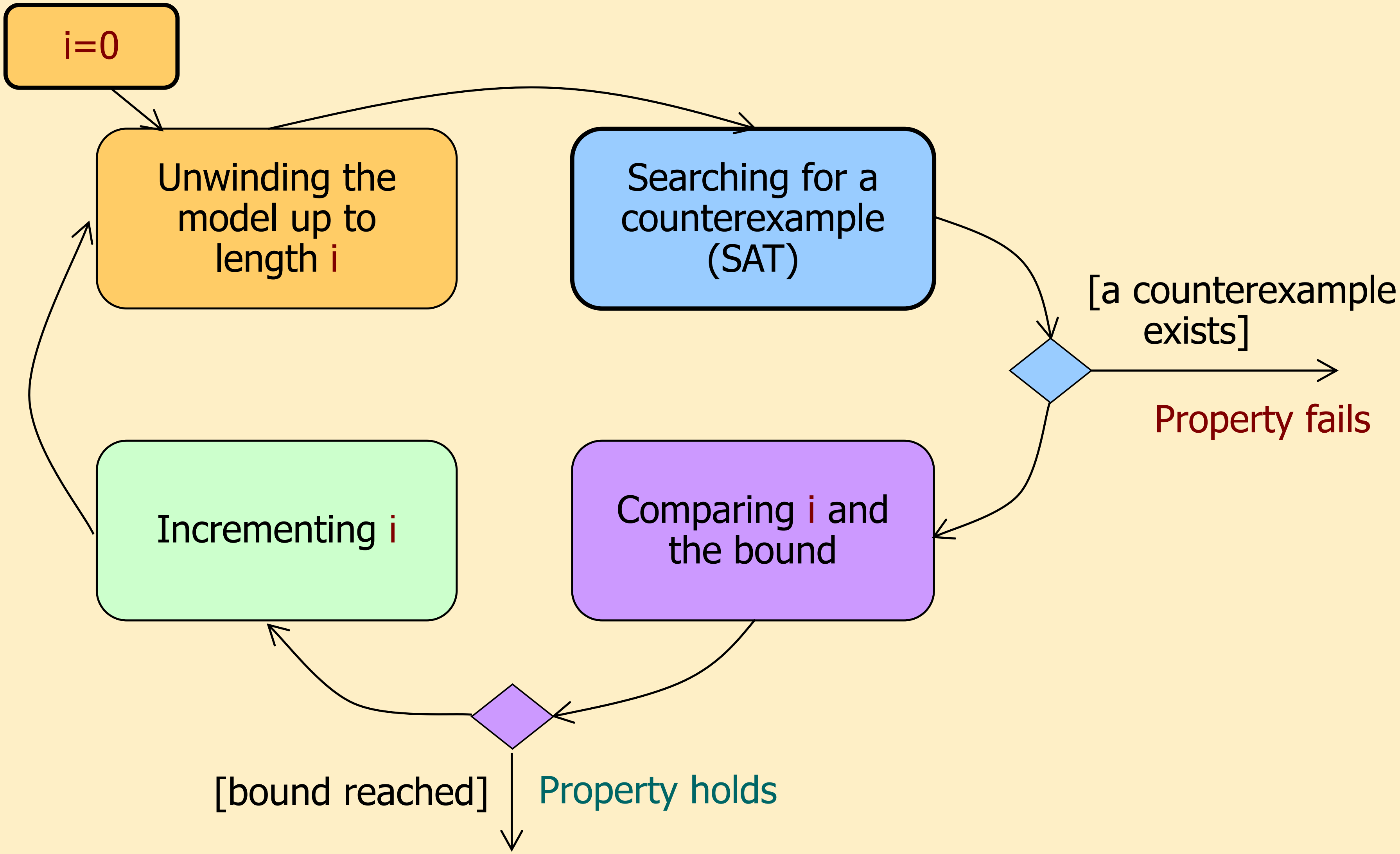
No more loop free paths to a bad state

There is a path from an initial state to an error state

iteration

- If the result is True: the invariant holds.
- If the result is a model inducing a path $(s^0, s^1, \ldots, s^i)$: it is a counterexample for the property p(s)

Bounded model checking with iteration



Refining the algorithm

- We do not start iterating from 0
  - We start with a given k,
    and try to generate the counterexample first:
    - If such a counterexample exists, we find it quickly (without iteration)!
  - We then examine whether for k+1 the iteration terminates,
    and then increase the bound
- It is not guaranteed that the length of the counterexample is minimal
  - We need some heuristic for estimating k if we aim to find a short counterexample
- Further restrictions on the input of SAT:
  - No initial states after the first (not necessarily a loop – there might be many initial states)
  - No bad states before the last state

The refined algorithm

$i = k$

while True do

if $\text{SAT}(I(s^0) \wedge \text{path}(s^0, s^1, \dots, s^i) \wedge \neg \bigwedge_{j=0}^{i} (p(s^j)))$

then return $(s^0, s^1, \dots, s^i)$

if not $\text{SAT}(I(s^0) \wedge \bigwedge_{j=1}^{i+1} (\neg I(s^j)) \wedge \text{lfpath}(s^0, s^1, \dots, s^{i+1}))$

or not $\text{SAT}((\text{lfpath}(s^0, s^1, \dots, s^{i+1}) \wedge \bigwedge_{j=0}^{i} p(s^j) \wedge \neg p(s^{i+1}))$

then return True

$i = i + 1$

end

Starting value

There is a path of length i from an inital state to a bad state

There is no cylce free path of length i+1 where only the first state is initial

There is no path of length i+1 where only the last state is bad

Summary: BMC

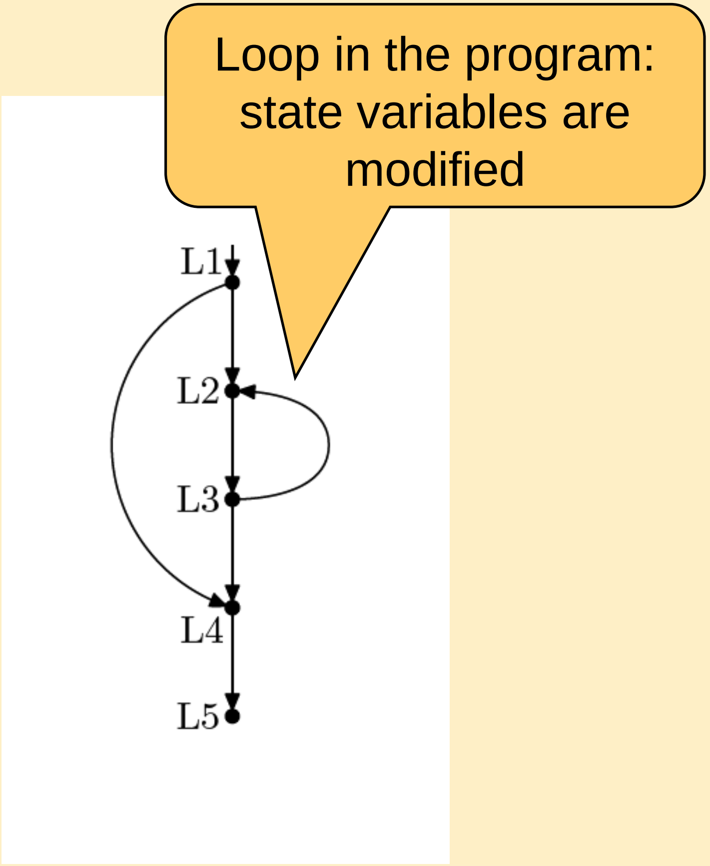
- Efficient for checking invariant poperties
- Sound method using loop free paths
  - If there is a counterexample up to a certain bound, it will be found
  - A counterexample found is a valid counterexample
- Handling the state space
  - SAT solver: symbolic technique using formulas
  - For up to a given unrolling a partial result is obtained
- Finding the shortest counterexampe
  - Can be used for test generation
- Automatic method
  - The bound can be determined heuristically (the diameter of the state space)
- Tools:
  - E.g. Symbolic Analysis Laboratory (SAL): sal-bmc, sal-atg

The results of Intel (hardware models)

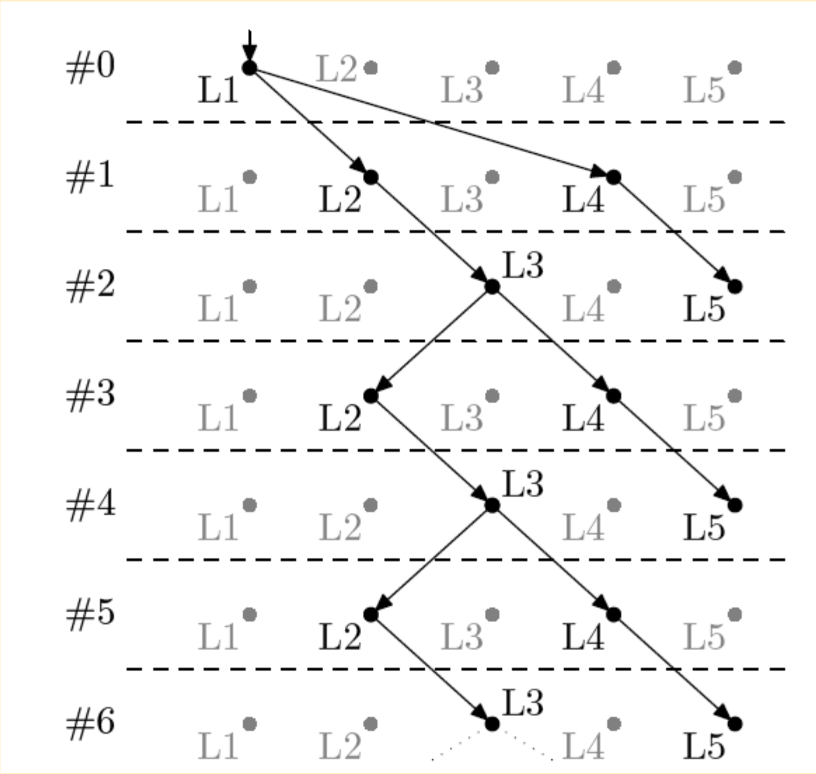
Model	k	Forecast (BDD)	Thunder (SAT)
Circuit 1	5	114	2.4
Circuit 2	7	2	0.8
Circuit 3	7	106	2
Circuit 4	11	6189	1.9
Circuit 5	11	4196	10
Circuit 6	10	2354	5.5
Circuit 7	20	2795	236
Circuit 8	28	—	45.6
Circuit 9	28	—	39.9
Circuit 10	8	2487	5
Circuit 11	8	2940	5
Circuit 12	10	5524	378
Circuit 13	37	—	195.1
Circuit 14	41	—	—
Circuit 15	12	—	1070

Use for software: the problem of loops

Traversing cycles might lead to new states



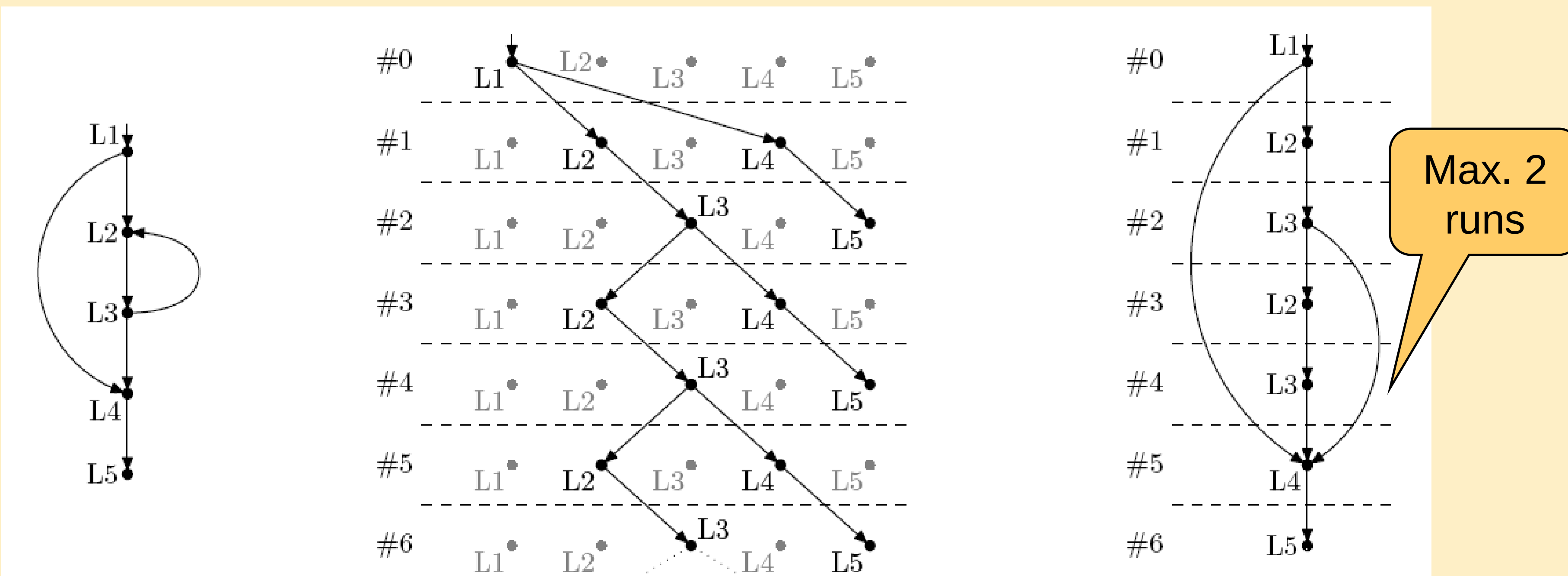
Control flow graph (CFG)



Complete unrolling

Loop unrolling

- Possibilities for unrolling the model:
  - Path enumeration:
    - Systematicall along all possible paths
  - Loop unrolling:
    - Unrolling loops for a given bound



Software model checking

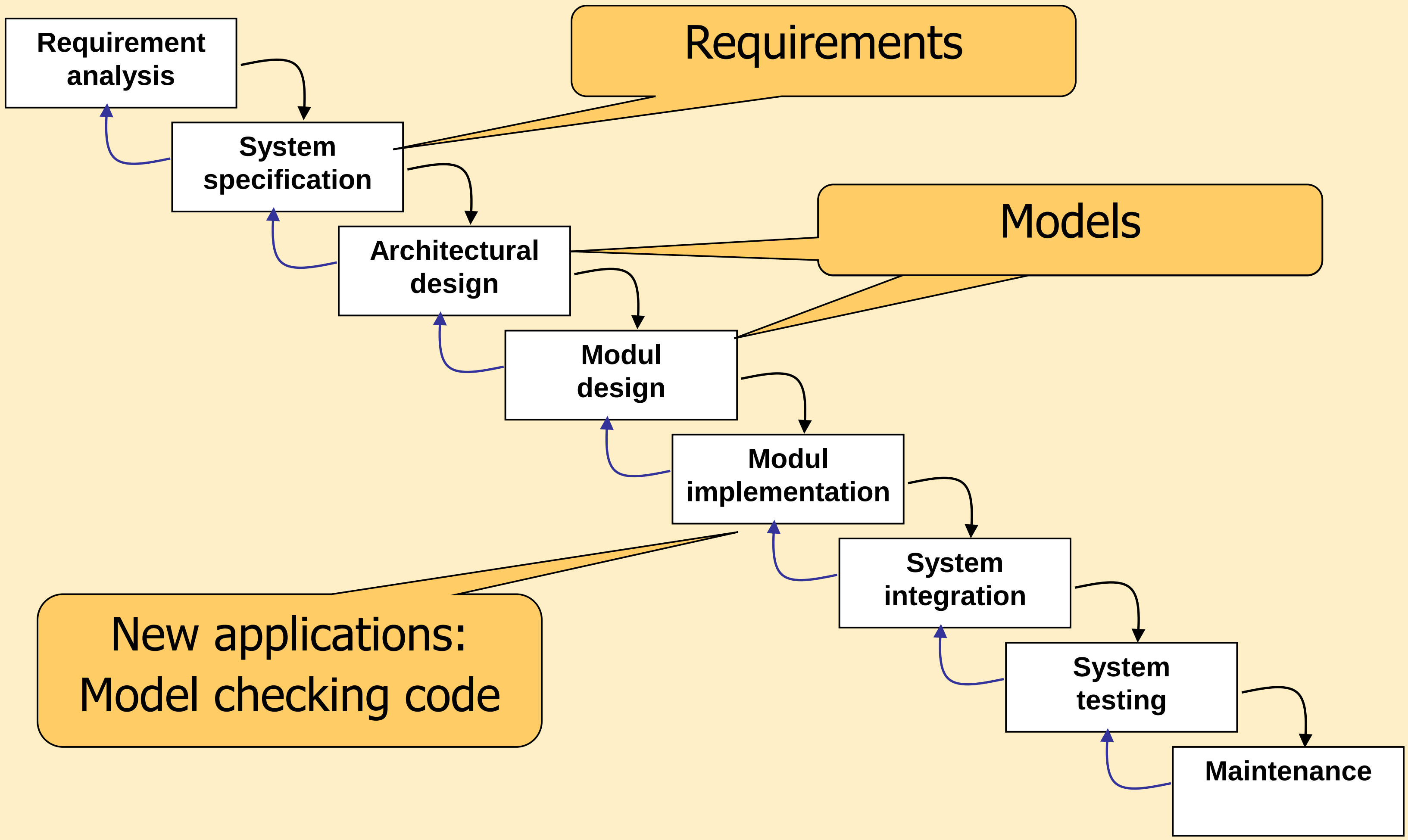
- F-SOFT (NEC):
  - Path enumeration
  - Used for unix system tuilities (e.g. pppd)
- CBMC (CMU, Oxford University):
  - Supports C, SystemC
  - Loop unrolling
  - Support for certain system libraries in Linux, Windows, MacOS
  - Handling integer arithmetic:
    - Bit level („bit-flattening", „bit-blasting")
  - CBMC with SMT solving:
    - Satisfiability Modulo Theories: extension to first order theories (e.g. integer arithmetic)
- SATURN:
  - Loop unrolling: at most 2 runs
  - Full Linux kernel verifiable: for Null pointer dereferences

Summary: efficient techniques for model checking

- Symbolic model checking
  - Charactereistic fomrulas represented as ROBDD
  - Efficient for „well structured” problems
    - E.g. identical processses in a protocol
  - Size depends on variable ordering
- Bounded model checking for invariant properties
  - Based on satisfiability solving (SAT solver)
  - Searching for counterexamples of bounded length
    - A counterexample found is a valid counterexample
    - If no counterexample found, it is only a partial result (longer counterexamples might exist)
  - Good for test generation

Properties of model checking

Model checking during the design phase



Strengths of model checking

- Possible to handle large state spaces
  - State spaces of size 10^{20}, but examples even for size 10^{100}
  - This is the state space of the system (e.g. network of automata)
  - Efficient techniques: symbolic, SAT based (bounded)
- General method
  - Software, hardware, protocols, …
- Fully automatic tool, no intuition or strong mathematical background is needed
  - Theorem proving is much harder!
- Generates a counterexample that can be used for debugging

Turing Award in 2007 for establishing model checking:
E. M. Clarke, E. A. Emerson, J. Sifakis (1981)

Weaknesses of model checking

- Scalability
  - Uses explicit state space traversal
  - Efficient techniques exist, but good scalability can not be guaranteed
- Mainly for control driven applications
  - Complex data structures induce a large state space
- Hard to generalize result
  - If the protocol is correct for 2 processes,
    is it correct for N processes?
- Formalizing requirements is hard
  - „Dialects” in temporal logic for different domains
  - E.g.: PSL (Property Specification Language, IEEE standard)