An Abstraction Technique for Testing Decomposable Systems by Model Checking

Paolo Arcaini - University of Bergamo, Italy
Angelo Gargantini - University of Bergamo, Italy
Elvinia Riccobene - University of Milan, Italy

8th International Conference on Tests & Proofs
July 24 - July 25, 2014, York, UK

Outline

1. Kripke structures with inputs
2. Model checking for test generation
   - Using counterexample/witness as test
   - State explosion problem
3. DDAP - Decomposable by Dependency Asynchronous Parallel systems
4. An abstraction for test generation using model checking for DDAPs
5. Some experiments
   - Using NuSMV – it well supports Kripke structures with inputs and for processes running in parallel

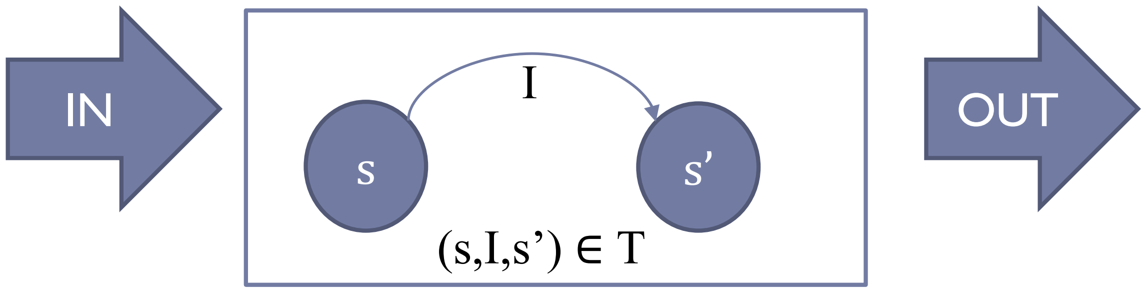
Kripke structures with inputs

- A Kripke structure with inputs is a 6-tuple

$$M = \langle S, S^0, IN, OUT, T, L \rangle$$

  - S is a set of states; $S^0 \subseteq S$ is the set of initial states;
  - IN and OUT are disjoint sets of atomic propositions;
  - $T \subseteq S \times \mathcal{P}(IN) \times S$ is the transition relation;
    - given a state s and the applied inputs I, the structure moves to a state s', such that $(s, I, s') \in T$.
  - $L: S \rightarrow \mathcal{P}(OUT)$ is the proposition labeling function.
- The set of atomic propositions is $AP = IN \cup OUT$ and CTL/LTL formulae are defined over AP
- Kripke structure with inputs differ from classical Kripke structures because the inputs are not part of the state and cannot be modified by M (but they are equivalent)

Kripke structures with inputs



- **Input sequence:** $I_0, \cdots, I_n, \cdots$ with $I_k \in \mathcal{P}(IN)$
- **Trace:**

$$s_0 \xrightarrow{I_0} s_1 \xrightarrow{I_1} s_2 \dashrightarrow s_i \xrightarrow{I_i} s_{i+1} \dashrightarrow$$

such that

- $s_0 \in S^0$ and
- $(s_i, I_i, s_{i+1}) \in T$

- **Test:** a test is a finite trace

Test generation by model checking

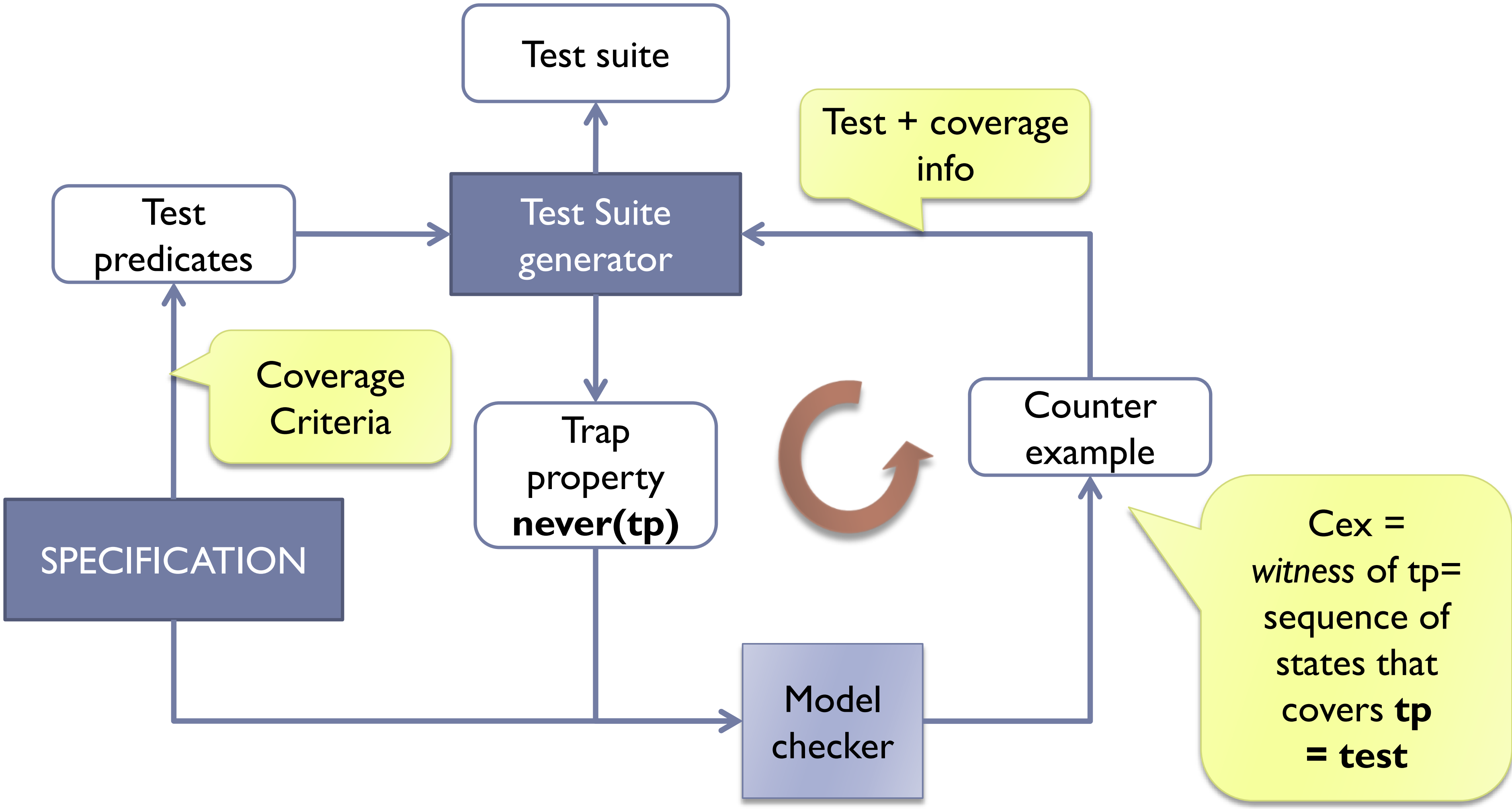
- **Test predicate**: A test predicate is a formula over the model, and determines if a particular testing goal is reached.

- Example:
  - Conditional statement

    `if C then …`

  - If one wants to cover a case in which `C` is true
  - LTL test predicate: **F**(C)

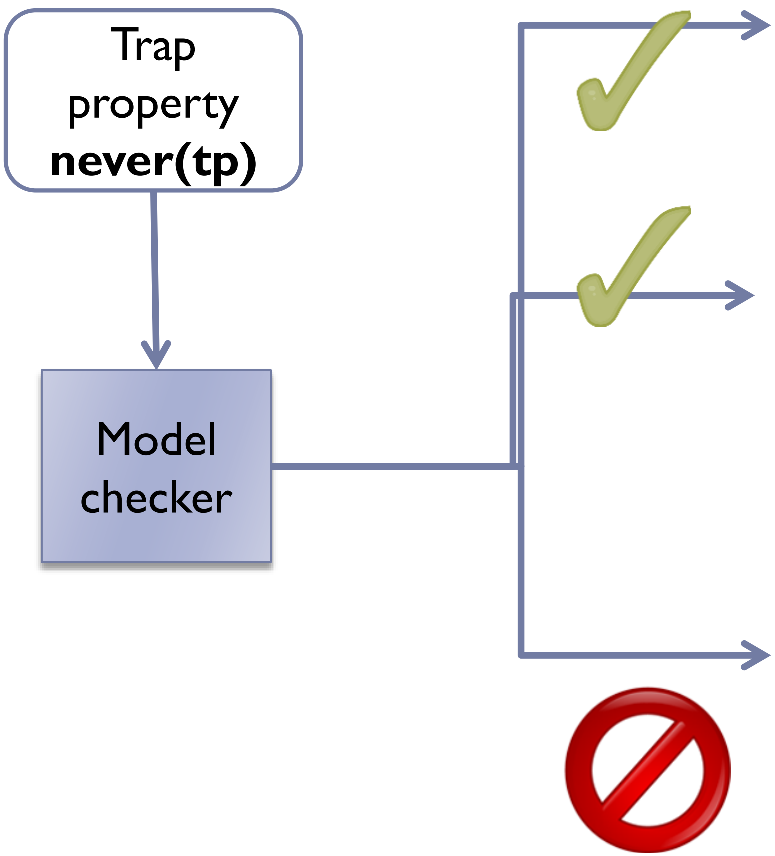
MC for test generation



Test generation by model checking

- Model checking for model-based tests generation is a well established *research* technique
  - [FAW09] reviewed 140 papers
  - [GH99] and [ABM98] have around 400 citations
  - several notations, systems, coverage criteria (data flow, structural, mutation, …) and using several model checkers
    - [FAW09] Fraser, Ammann, and Wotawa. *Testing with Model Checkers: A Survey. Journal for Software Testing,* Verification and Reliability, 2009
    - [GH99] Gargantini, Heitmeyer. *Using model checking to generate tests from requirements specifications.* FSE/ESEC, 1999
    - [ABM98] Ammann, Black, Majurski. *Using model checking to generate tests from specifications.* Formal Engineering Methods, 1998
- Several commercial tools are based on model checking techniques (like mathworks)

Some limits



1. Trap property proved false
   - Cex found
2. Trap property proved true
   - Test predicate infeasible
3. MC does neither complete the proof nor finds the counter example
   - Out of memory (state explosion problem)

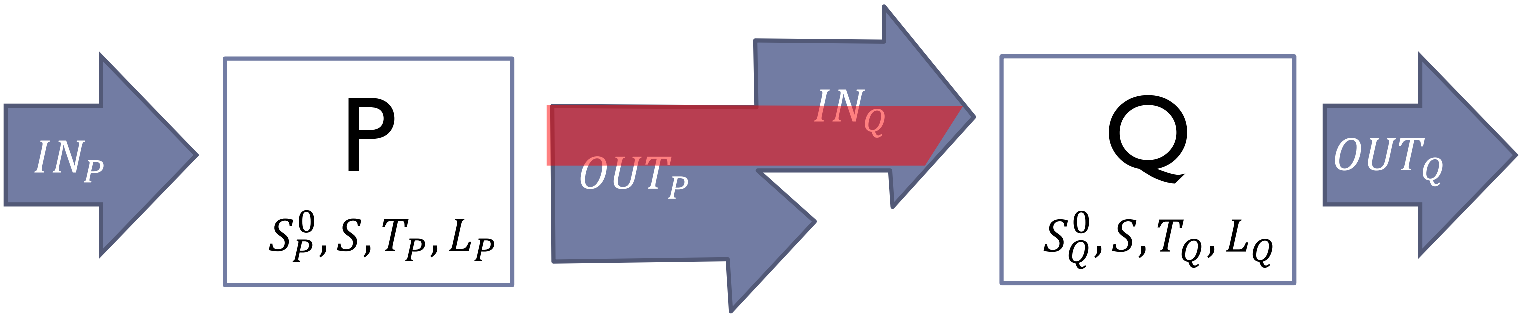
Main problem: scalability

- Model checker (symbolically) explores the entire state space
- It suffers from the state explosion problem
  - A combinatorial blow up of the state-space
  - It limits its usability

- Are there particular classes of systems which can be abstracted for test generation?
  - *Sequential nets of abstract state machines*, ABZ 2012
  - *with information passing*, Science of Computer Programming, 2014
  - Running in parallel?

DDAP systems

- Decomposable by Dependency Asynchronous Parallel systems (DDAP) systems.
- A DDAP system is composed of two subsystems,
  1. running asynchronously in parallel,
  2. (part of) the inputs of the dependent subsystem are provided by the other subsystem

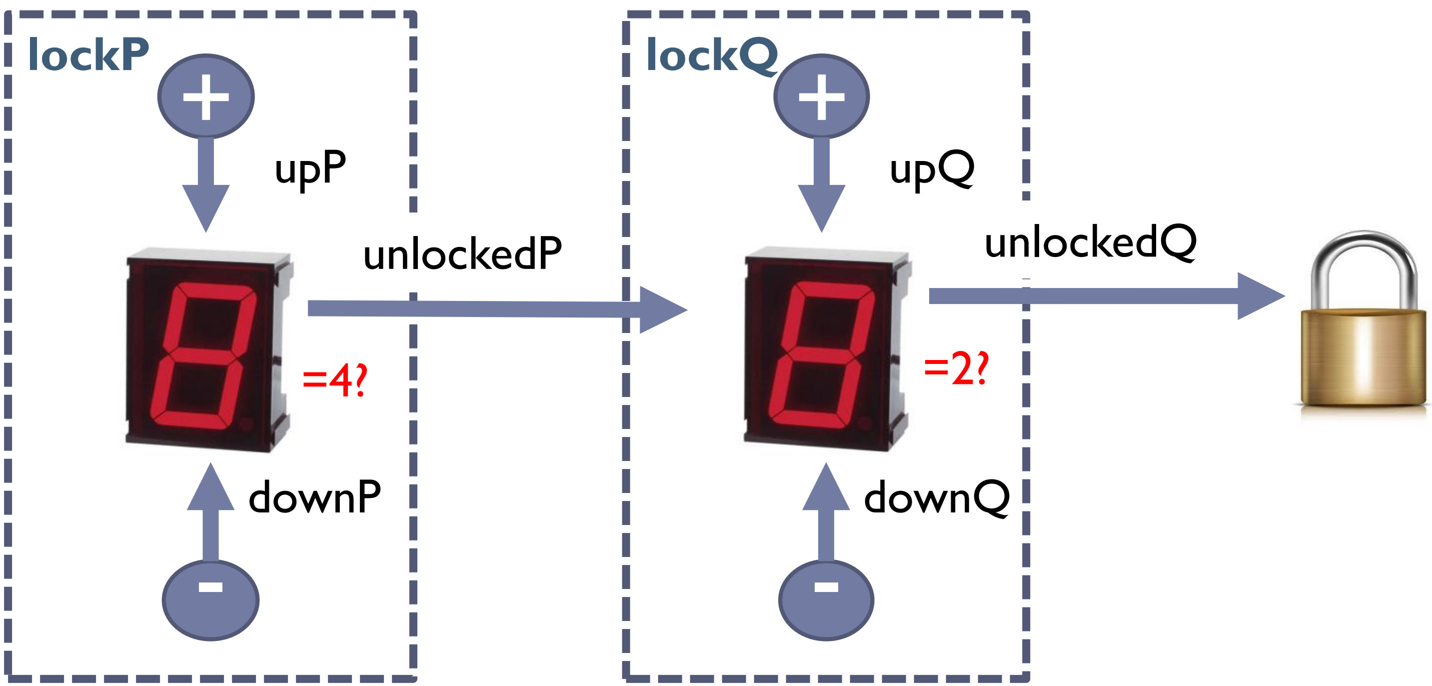


Q depends on P: dependency set D

$D = OUT_P \cap IN_Q$ $\quad D \neq \emptyset$

DDAP example

- A safelock composed by two locks working in sequence
  - Each combination digit is a lock
  - It becomes unlocked if the two locks are unlocked
  - The combination is 42



For a DDAP K = $\langle P, Q \rangle$

- **input set** is the union of the inputs (except D):
  - $IN_K = IN_P \cup IN_Q \backslash D$
- **input sequence:**
  - $J_0, \cdots, J_n, \cdots$ with $J_k \in \mathcal{P}(IN_K)$
- **trace:**

  $(p_0, q_0) \xrightarrow{J_0} (p_1, q$

  either the component P moves from p_i to p_{i+1} and Q remains still in state $q_i = q_{i+1}$, or component Q moves from q_i to q_{i+1} and P remains still in state $p_i = p_{i+1}$

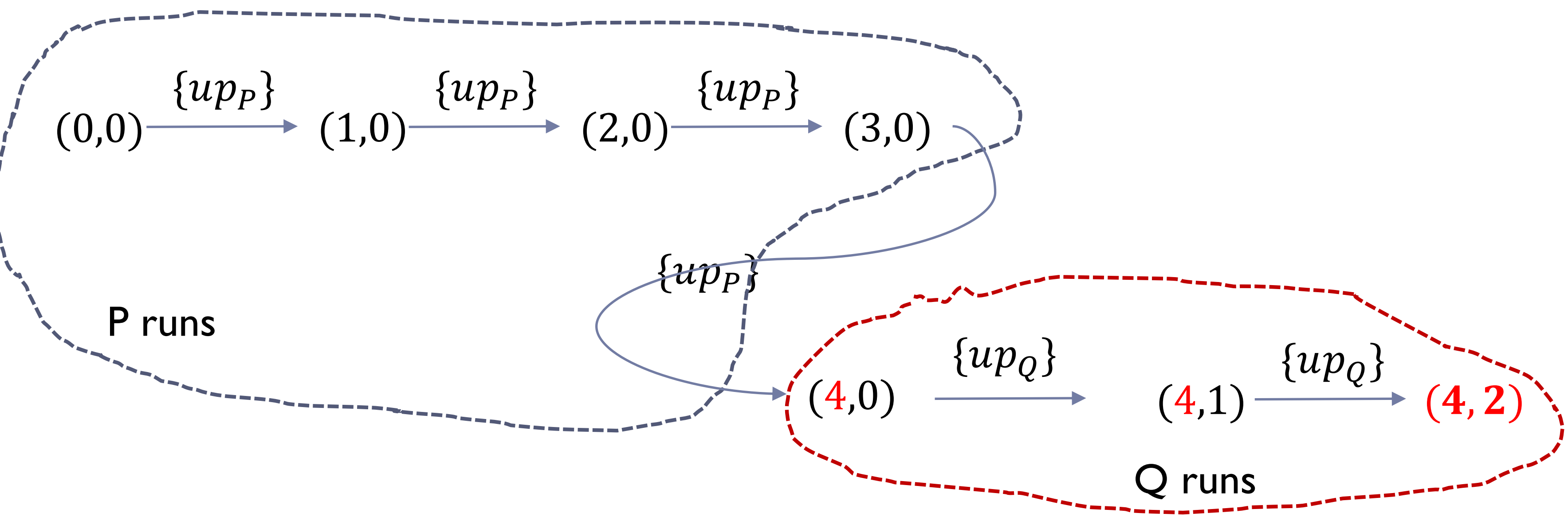
  - such that
    - $p_0 \in S_P^0$ and $q_0 \in S_Q^0$
    - $(p_i, J_i \cap IN_P, p_{i+1}) \in T_P \wedge q_i = q_{i+1} \oplus$

$p_i = p_{i+1} \wedge \left(q_i, J_i \cap IN_Q \cup L(p_i) \cap D, q_{i+1}\right) \in T_Q$

When Q moves, it reads some of its inputs from the outputs of P

Safelock trace example

- $IN_{SafeLock}$ = {upP, downP, upQ , downQ }
- Trace in which the lock is unlocked:



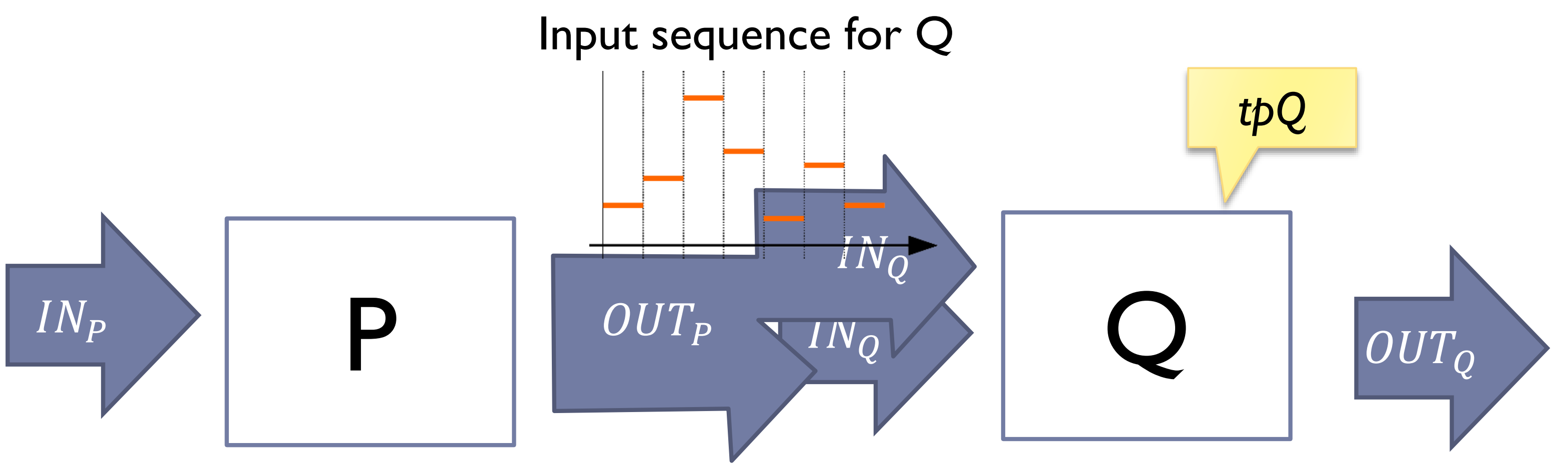
red state when the lock is unlocked

Test Generation for DDAP systems

- We propose an abstraction that exploits dependency between inputs and outputs to decompose the complete system
- The proposed test generation approach consists in generating two tests, one over Q and one P, and merging them later.
  - Since model checkers suffer exponentially from the size of the system, decomposition brings an exponential gain and allows to test large systems.
- Assume that the test predicate refers to Q
  - If it refers to P, COI abstraction is enough

Step 1: build a test for Q

- Given a test predicate *tpQ* for Q
- Consider only Q and ignore P
- Compute the necessary input sequence to obtain the desired test case (witness for *tpQ*)



Step1. Formally: a witness for *tpQ*

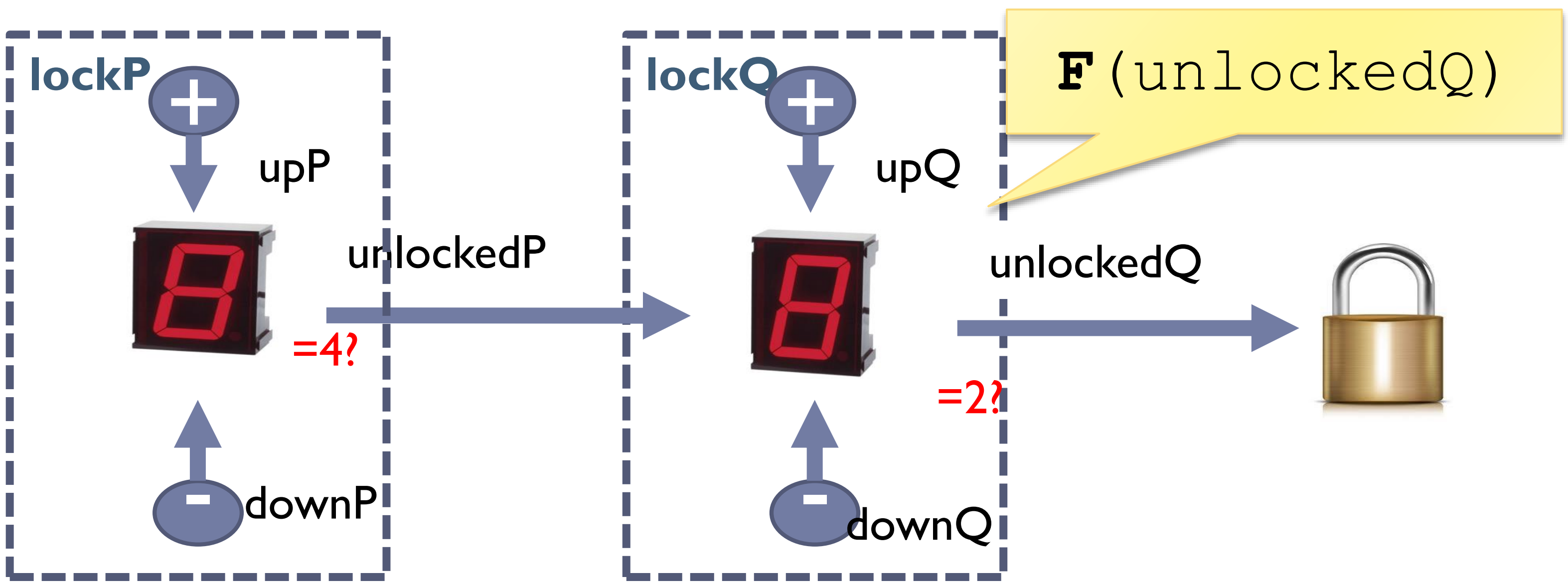
- We compute its witness by asking the model checker for a counterexample for the trap property ¬*tpQ*
- The witness is a finite trace of Q, $testQ$:

$$q_0 \xrightarrow{IQ_0} q_1 \xrightarrow{IQ_1} q_2 \dashrightarrow \quad \xrightarrow{IQ_{m-1}} q_m$$

  - $IQ_j \subseteq IN_Q$ is the set of inputs of Q applied at state q_j
  - Parts of inputs come from P (those in the dependency set): $IQ_j \cap D$

Example SafeLock – Step 1

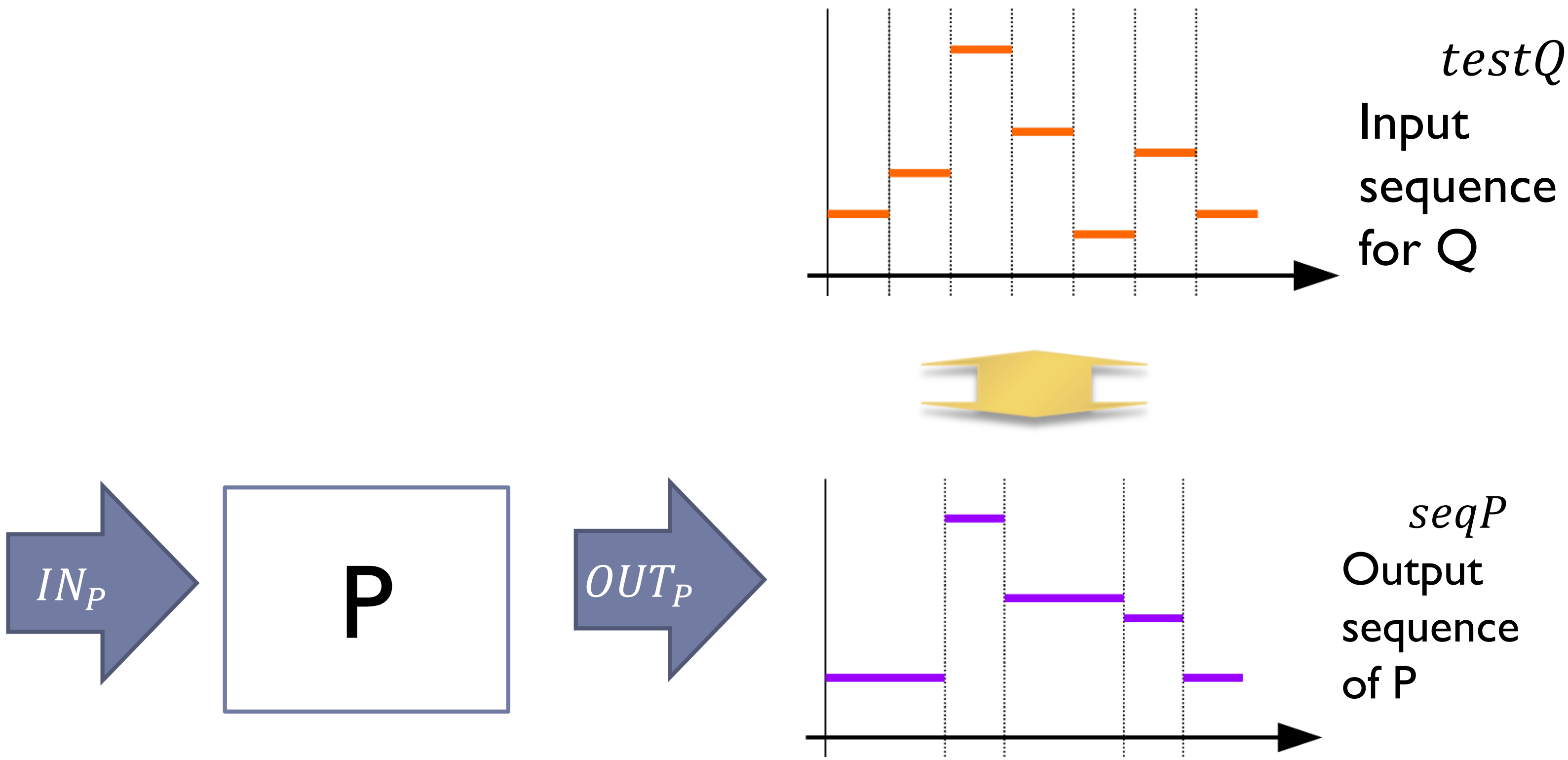
- Test goal: the lock becomes unlocked
- Ignore lockP and build a test for lockQ



$$(0) \xrightarrow{\{unlockedP,\, up_Q\}} (1) \xrightarrow{\{unlockedP,\, up_Q\}} (2)$$

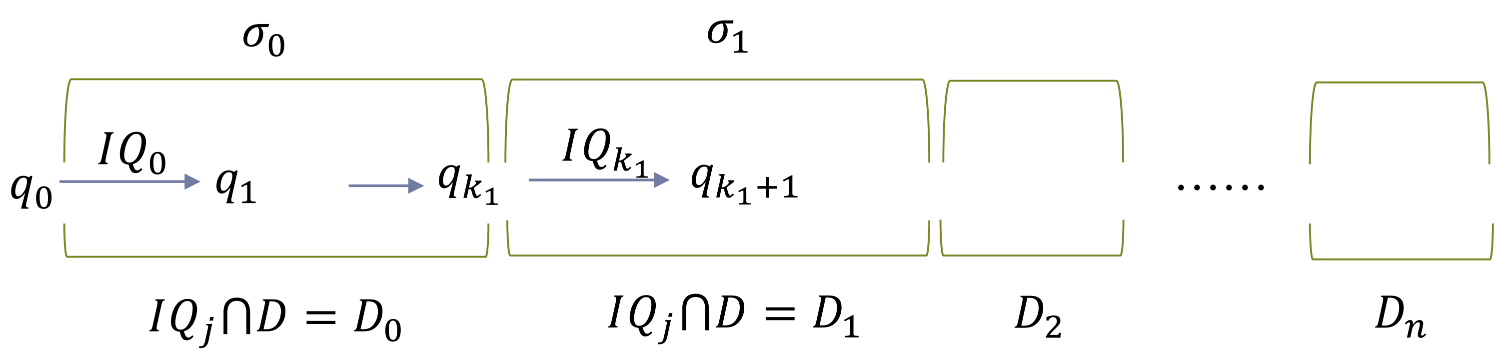
Step 2: transform the trace for P

- The input sequence for Q must be transformed to a sequence of outputs for P



Step 2. Split $testQ$

- We split the sequence $testQ$ in subsequences $\sigma_i \;\; i = 0, \ldots, n$ such that atomic propositions of the dependency set remain unchanged:



$seqP = D_0, D_1, \ldots, D_n$ constitutes the input sequence part for Q coming from P

SafeLock – Step 2

- Transform the test for lockQ to an output sequence for lockP

$$(0) \xrightarrow{\{unlockedP,\, up_Q\}} (1) \xrightarrow{\{unlockedP,\, up_Q\}} (2)$$

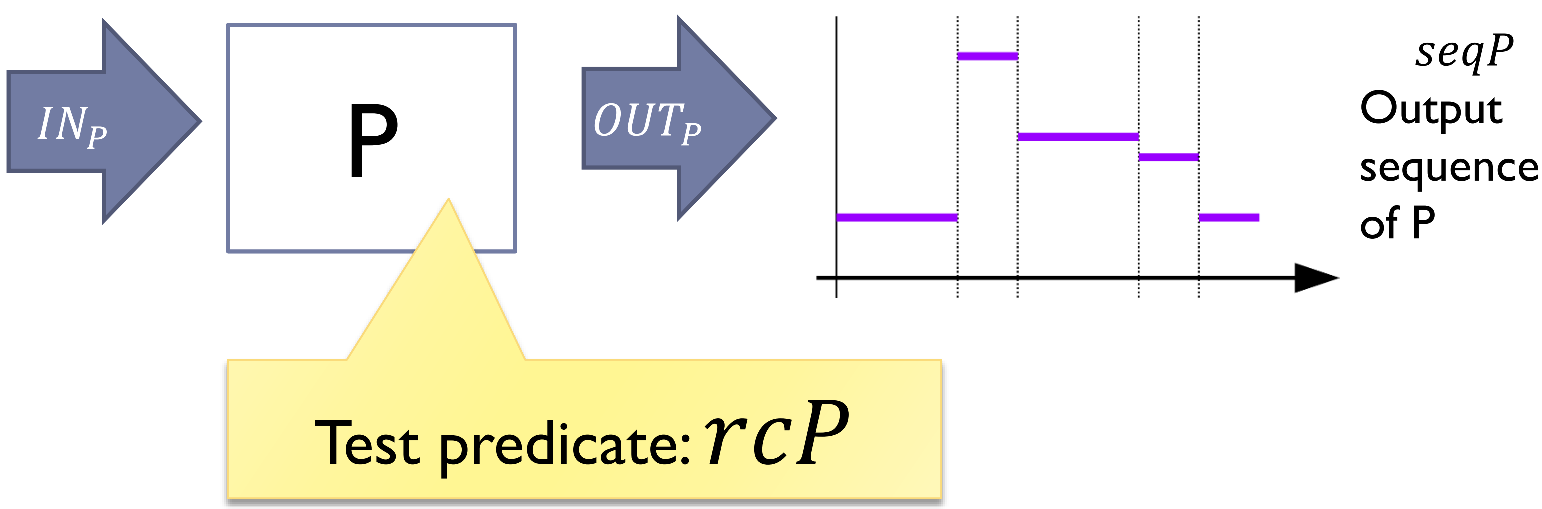
$$D_0 = \{unlockedP\}$$

$$seqP = \{unlockedP\}$$

Desired output sequence for lockP

Step 3: generate the trace for *seqP*

- To generate a trace for $seqP = D_0, D_1, \ldots, D_n$ we can build a suitable LTL property and find a witness for it



Step 3: build reachability condition

- In order to obtain the output sequence $D_0, D_1, \dots, D_n$ for P, we build the LTL formula over the AP of P

$$rcP = \mathbf{F}\left(\bigwedge_{d_0 \in D_0} d_0 \wedge \mathbf{F}\left(\cdots \mathbf{F}\left(\bigwedge_{d_n \in D_n} d_n\right)\right)\right)$$

rcP requires that n + 1 subsequent states exist, in which P produces the output values D_i requested by Q to start the computation σ_i

SafeLock – Step 3

- Transform the test for lockQ to a test property for lockP

$$seqP = \{unlockedP\}$$

Test property for lockP:

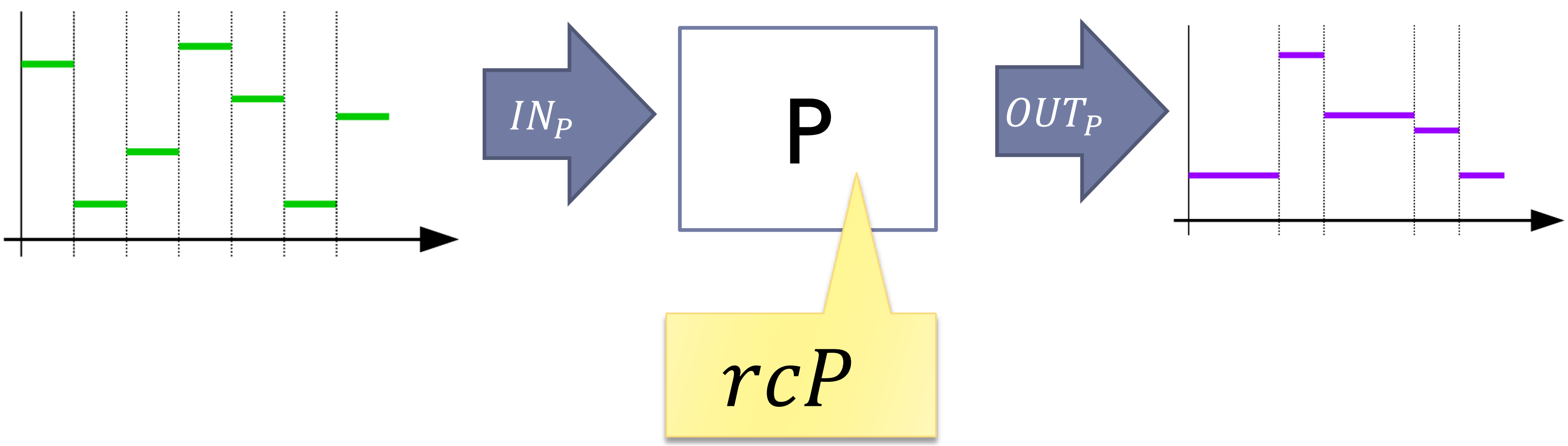
`F(unlockedP)`

Step 4. build the test for P

- The witness of rcP is $testP$

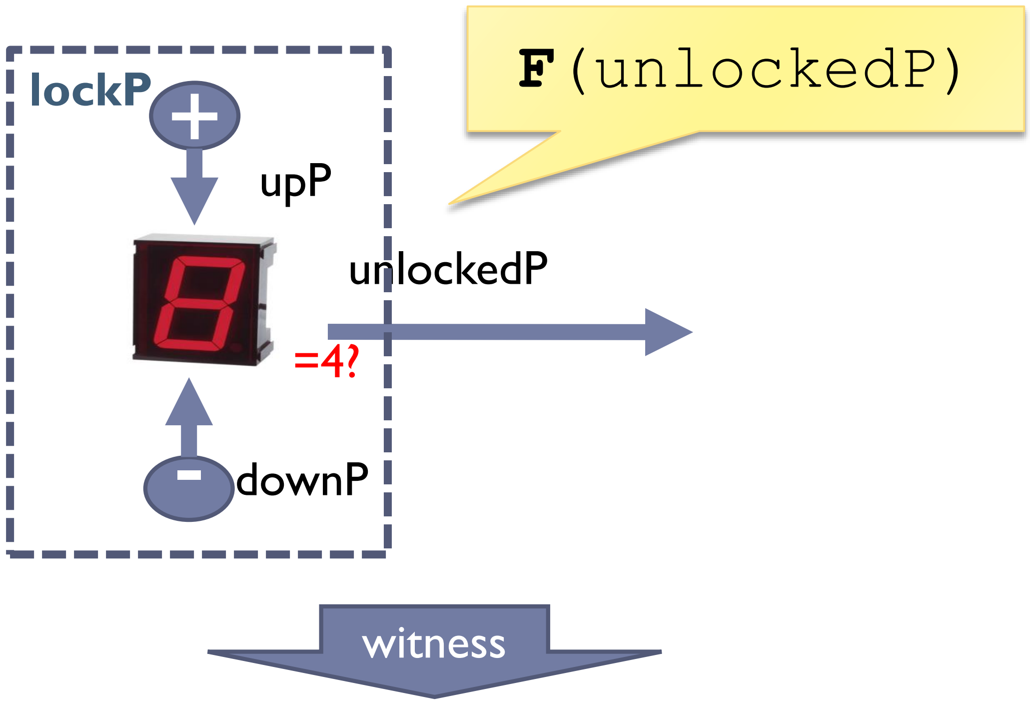
$testP$: Input sequence of P

Output sequence of P



SafeLock – Step 4

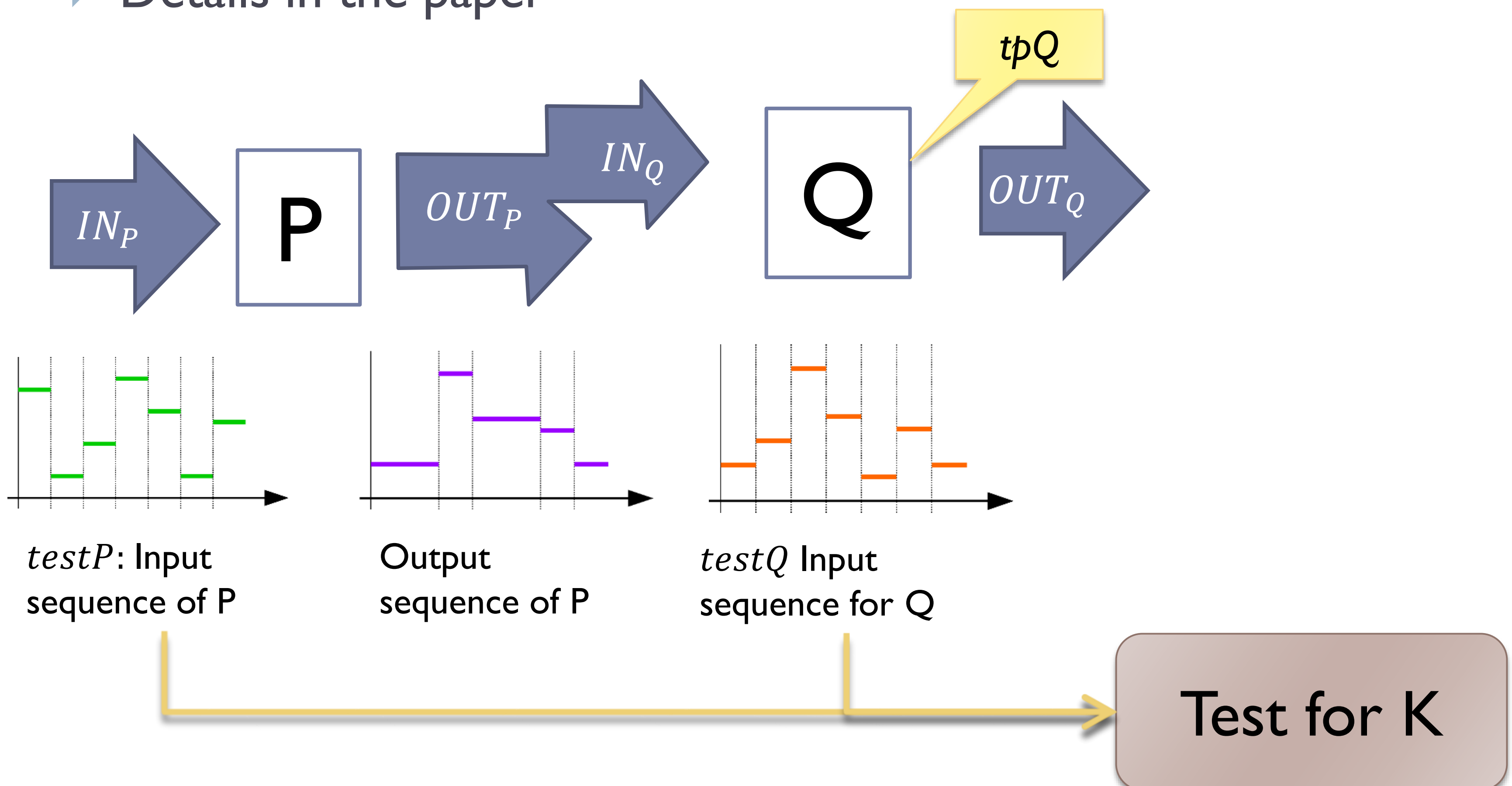
- Build a test for lockP



$$(0) \xrightarrow{\{up_P\}} (1) \xrightarrow{\{up_P\}} (2) \xrightarrow{\{up_P\}} (3) \xrightarrow{\{up_P\}} (4)$$

Step 5: build the test for K

- Merge *testP* and *testQ* in order to obtain a test for K
  - Details in the paper

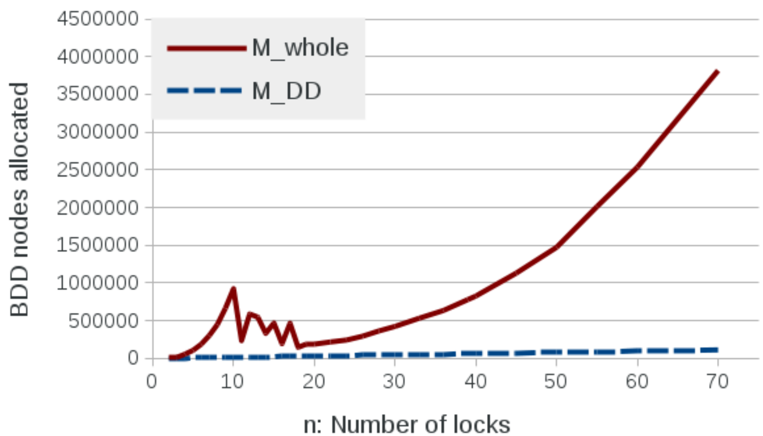


Soundness and Completeness

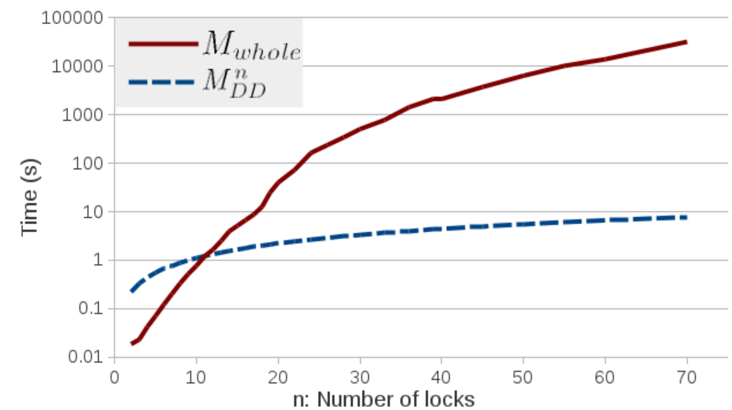
- The proposed approach is:
- **SOUND**: if a test is found, it is a valid test for K
- **INCOMPLETE**: a test that could be found using the whole system, it may not be found using the proposed decomposition

Initial Experiments for n-SafeLock

Memory



Time



- the required memory grows exponentially if we consider the whole system, whereas, using the abstraction, it grows linearly.

- The same for the time
- Except that for small N, the whole system takes less time.

Conclusions

- Systems composed by several subsystems
  - running asynchronously in parallel
  - connected together in a way that (part of) the inputs of one subsystem are provided by another subsystem.
- **Proposed abstraction:** split the systems, generate the tests and merge together.
  - Exponential gain in terms of state space
  - Proved correct (details in the paper)
    - But incomplete
  - It can be generalized to n-subcomponents