Regular Model Checking

for Formal Verification of Infinite-State Systems

Tomáš Vojnar

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Sources of Infinity

- Unbounded communication queues (channels), unbounded waiting queues.
- Unbounded push-down stacks: recursion.
- Unbounded counters, unbounded capacity of places in Petri nets.
- Unbounded string variables.
- Continuous variables: time, temperature, ...
- Unbounded dynamic spawning of threads, dynamic memory allocation:
  - dynamic linked (circular/shared/nested/...) lists, trees, skip-lists, ...
- Parameterisation:
  - parametric bounds of queues, counters, ...,
  - parametric networks of processes.

Model Checking Infinite-State Systems

❖ Cut-offs: safe, finite bounds on the sources of infinity such that when a system is verified up to these bounds, the results may be generalised.

❖ Abstraction:

- predicate abstraction: $x \in \{5, 6, 7, ...\} \leadsto x \geq 5$,
- abstractions for parameterised networks of processes: 0-1-∞ abstraction, ...

❖ Symbolic methods: finite representation of infinite sets of states using

- logics,
- grammars,
- automata, ...

❖ Automated induction, ...

Decidability Issues

- Formal verification of infinite state systems is usually **undecidable**.

- There exist (sub)classes of systems for which various problems are decidable:
  - push-down systems—model checking LTL is even polynomial for a fixed formula,
  - lossy channel systems—reachability, safety, inevitability, and (fair) termination are decidable (though non-primitive recursive),
  - various parameterised systems for which finite cut-offs exist,
  - ...

- Otherwise, semi-algorithmic solutions can be used:
  - termination is not guaranteed,
  - an indefinite answer may be returned, or
  - a help from the user is needed: invariants, predicates, ...

Regular Model Checking

The Basic Idea

Regular Model Checking

[Pnueli et al. 97], [Wolper, Boigelot 98], [Bouajjani, Nilsson, Jonsson, Touili 00]

- A **generic** framework for verification of infinite-state systems:
  - a configuration $\rightsquigarrow$ a word w over a suitable alphabet Σ,
  - a set of configurations $\rightsquigarrow$ a regular language:
    - usually described by a finite-state automaton A,
    - two distinguished sets of configurations:
      - initial configurations $Init$ and
      - bad configurations Bad,
  - an action (transition) $\rightsquigarrow$ a rational relation τ:
    - usually described by a finite-state transducer T,
    - sometimes, more general, regularity-preserving relations are used.
      - Implemented, e.g., as specialised operations on automata.
- Safety verification $\rightsquigarrow$ check that $\tau^*(Init) \cap Bad = \emptyset$,
  - implies a need to compute $\tau^*(Init)$ or its sufficiently precise approximation.

Regular Model Checking: Applications

- Communication protocols.
  - Lossy/non-lossy FIFO channels systems / cyclic rewrite systems.
- Sequential programs with recursive procedure calls.
  - Push-down systems / prefix rewrite systems.
- Counter systems, Petri nets.
  - Various systems may be (automatically) translated to counter systems.
- String manipulating programs. [Yu, Alkhalaf, Bultan, Ibarra et al 08–17]
- Programs with (unbounded) dynamic linked data structures:
  - lists, cyclic lists, shared lists. [Bouajjani, Habermehl, V., Moro 05]
- Parameterized networks of processes:
  - mutual exclusion and cache coherence protocols, ..., [many of the mentioned works]

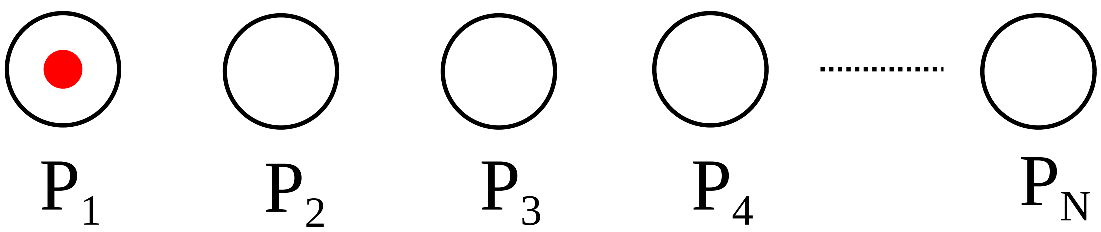
$$q_1 q_2 \cdots q_{i-1} q_i q_{i+1} \cdots q_j \cdots q_n \mapsto q_1 q_2 \cdots q_{i-1} q_i' q_{i+1} \cdots q_j' \cdots q_n$$

  - pipelined microprocessors. [Charvát, Smrčka, V. 14–19]

Example: A Simple Token Passing

- A simple protocol in a linear process network:
  - a parametric number of processes,
  - a process does or does not have a token,
  - a process that has a token passes it to the right.

- Initially, a token is in the left-most process.

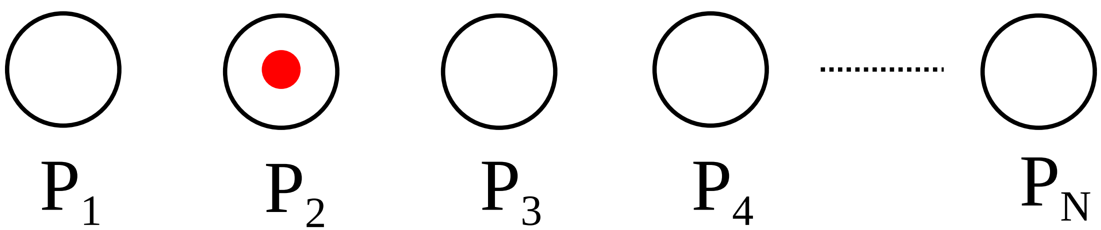


- Check that the token cannot disappear nor duplicate.

Example: A Simple Token Passing

- A simple protocol in a linear process network:
  - a parametric number of processes,
  - a process does or does not have a token,
  - a process that has a token passes it to the right.

- Initially, a token is in the left-most process.

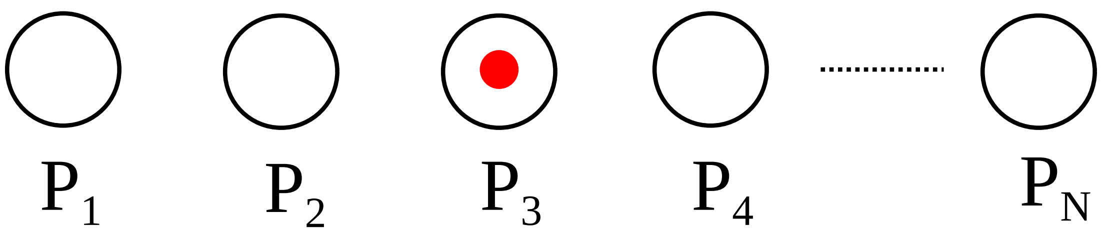


- Check that the token cannot disappear nor duplicate.

Example: A Simple Token Passing

- A simple protocol in a linear process network:
  - a parametric number of processes,
  - a process does or does not have a token,
  - a process that has a token passes it to the right.

- Initially, a token is in the left-most process.

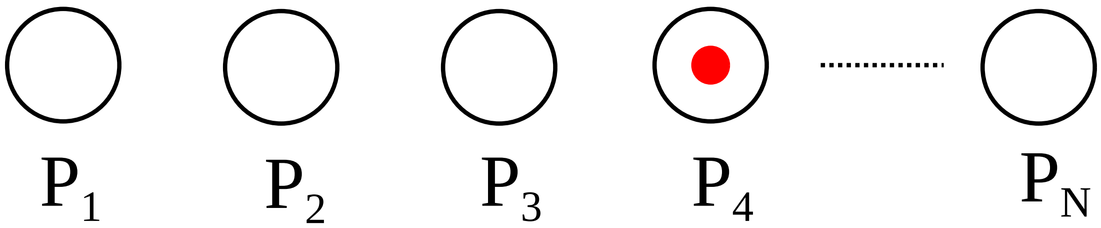


- Check that the token cannot disappear nor duplicate.

Example: A Simple Token Passing

- A simple protocol in a linear process network:
  - a parametric number of processes,
  - a process does or does not have a token,
  - a process that has a token passes it to the right.

- Initially, a token is in the left-most process.

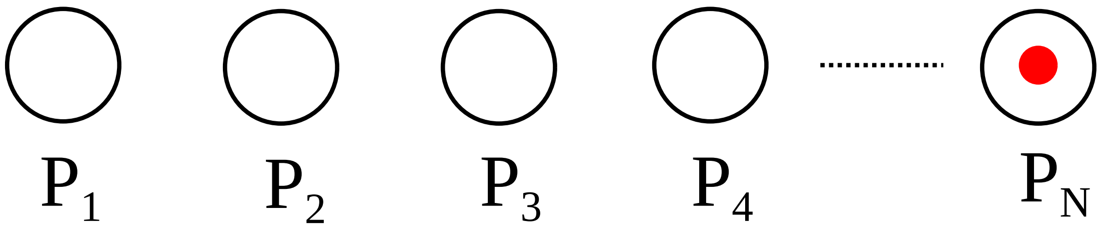


- Check that the token cannot disappear nor duplicate.

Example: A Simple Token Passing

- A simple protocol in a linear process network:
  - a parametric number of processes,
  - a process does or does not have a token,
  - a process that has a token passes it to the right.

- Initially, a token is in the left-most process.

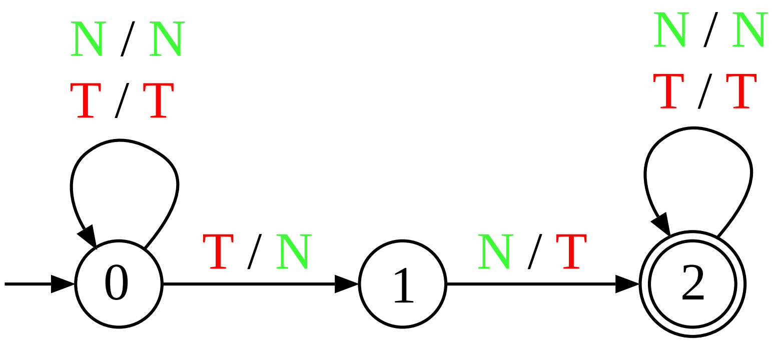


- Check that the token cannot disappear nor duplicate.

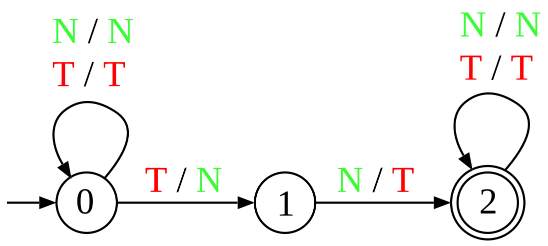
Example: A Simple Token Passing

❖ An **encoding** of the simple token passing protocol for the needs of RMC:

- the alphabet: $\Sigma = \{T, N\}$,
- configurations: words from Σ^*, e.g., $N\ N\ T\ N$,
- initial configurations: $T\ N^*$ (a regular language),
- bad configurations: $N^* + (T + N)^*\ T\ N^* T\ (T + N)^*$ (a regular language),
- transitions—in the form of a finite-state transducer:



Example: A Simple Token Passing



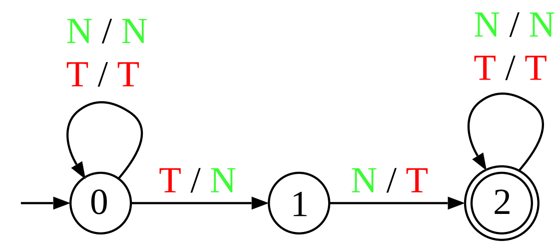
- An application of the transducer on a single configuration:

$T\ N\ N\ N \xrightarrow{\tau} N\ T\ N\ N \xrightarrow{\tau} N\ N\ T\ N \xrightarrow{\tau} N\ N\ N\ T$

Example: A Simple Token Passing

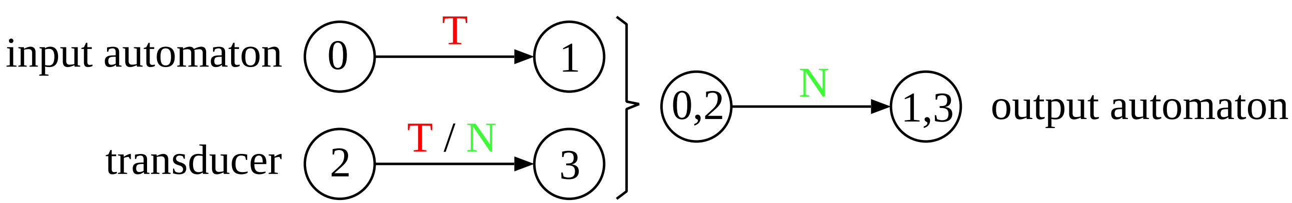
- An application of the transducer on a single configuration:

$T\ N\ N\ N \stackrel{\tau}{\to} N\ T\ N\ N \stackrel{\tau}{\to} N\ N\ T\ N \stackrel{\tau}{\to} N\ N\ N\ T$

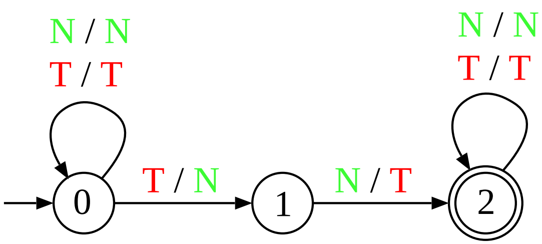


- An application of the transducer on all initial configurations:

$$T\ N^* \stackrel{\tau}{\to} N\ T\ N^* \stackrel{\tau}{\to} N\ N\ T\ N^* \stackrel{\tau}{\to} N\ N\ N\ T\ N^* \stackrel{\tau}{\to} \ldots$$



Example: A Simple Token Passing

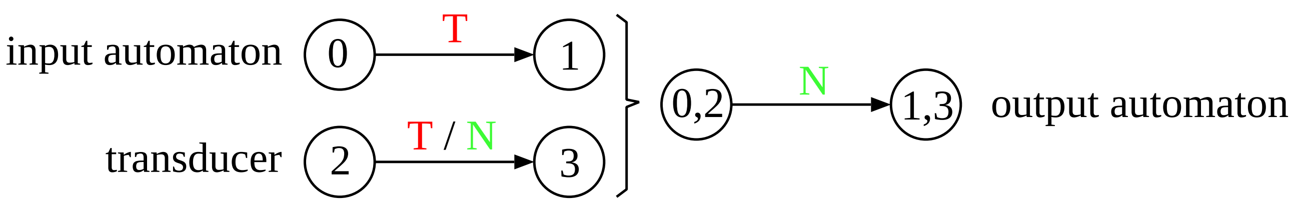


❖ An application of the transducer on a single configuration:

$T\ N\ N\ N \xrightarrow{\tau} N\ T\ N\ N \xrightarrow{\tau} N\ N\ T\ N \xrightarrow{\tau} N\ N\ N\ T$

❖ An application of the transducer on all initial configurations:

$$T\ N^* \xrightarrow{\tau} N\ T\ N^* \xrightarrow{\tau} N\ N\ T\ N^* \xrightarrow{\tau} N\ N\ N\ T\ N^* \xrightarrow{\tau} \dots$$



❖ A simple iterative computation of all reachable configurations will **never converge** to the desired set $N^*\ T\ N^*$.

- Need special (accelerated) ways for computing/over-approximating $\tau^*(Init)$.

Regular Model Checking

Computing Closures

RMC: Computing Closures

The task: compute/over-approximate $\tau^*(Init)$.

❖ Problems to face:

- Non-regularity / non-constructibility of $\tau^*(Init)$.
- Termination of the constructions.
- State explosion in the automata / transducers.

RMC: Computing Closures

The task: compute/over-approximate $\tau^*(Init)$.

❖ Problems to face:

- Non-regularity / non-constructibility of $\tau^*(Init)$.
- Termination of the constructions.
- State explosion in the automata / transducers.

❖ Solutions:

- Specialised constructions: LCS, PDS, classes of arithmetical relations, lists, ...
- General-purpose constructions:
  - widening by extrapolating repeated patterns, [Bouajjani, Touili], [Wolper, Boigelot, Legay]
  - merging states wrt the history of their creation, [Abdulla, Nilsson, Jonsson, d'Orso]
  - widening by merging states wrt their fw/bw languages, [Yu, Alkhalaf, Bultan, Ibarra]
  - refinable abstraction by state merging, [Bouajjani, Habermehl, V.]
  - automata learning, [Habermehl, V.], [Vardhan, Sen, Viswanathan, Agha], [Chen, Hong, Lin, Rümmer]
  - ...

Abstract Regular Model Checking

- Given a relation τ, and two automata I (initial states) and B (bad states), check:

$$\tau^*(I) \cap B = \emptyset$$

1. Define a finite-range abstraction α on automata s.t. $L(A) \subseteq L(\alpha(A))$.
2. Compute iteratively $(\alpha \circ \tau)^*(I)$.
3. If $(\alpha \circ \tau)^*(I) \cap B = \emptyset$, then answer YES.

Abstract Regular Model Checking

❖ Given a relation τ, and two automata I (initial states) and B (bad states), check:

$$\tau^*(I) \cap B = \emptyset$$

1. Define a finite-range abstraction α on automata s.t. $L(A) \subseteq L(\alpha(A))$.
2. Compute iteratively $(\alpha \circ \tau)^*(I)$.
3. If $(\alpha \circ \tau)^*(I) \cap B = \emptyset$, then answer YES.

4. Otherwise, let θ be the computed symbolic path from I to B.
5. Check if θ includes a concrete counterexample.
   - If yes, then answer NO.
   - Otherwise, refine α s.t. it excludes θ and goto (2).

Abstract Regular Model Checking

❖ Given a relation τ, and two automata I (initial states) and B (bad states), check:

$$\tau^*(I) \cap B = \emptyset$$

$\Longrightarrow$ *Counter-Example Guided Abstraction Refinement (CEGAR) loop*

1. Define a finite-range abstraction α on automata s.t. $L(A) \subseteq L(\alpha(A))$.
2. Compute iteratively $(\alpha \circ \tau)^*(I)$.
3. If $(\alpha \circ \tau)^*(I) \cap B = \emptyset$, then answer YES.
4. Otherwise, let θ be the computed symbolic path from I to B.
5. Check if θ includes a concrete counterexample.
    - If yes, then answer NO.
    - Otherwise, refine α s.t. it excludes θ and goto (2).

Abstractions Based on State Collapsing

- We abstract automata by collapsing their states that are equal wrt some criterion s.t.:

$$L(A) \subseteq L(\alpha(A)).$$

- Various equivalences on automata states can be used, e.g.:

  - Equivalence wrt languages of words of a bounded length k:

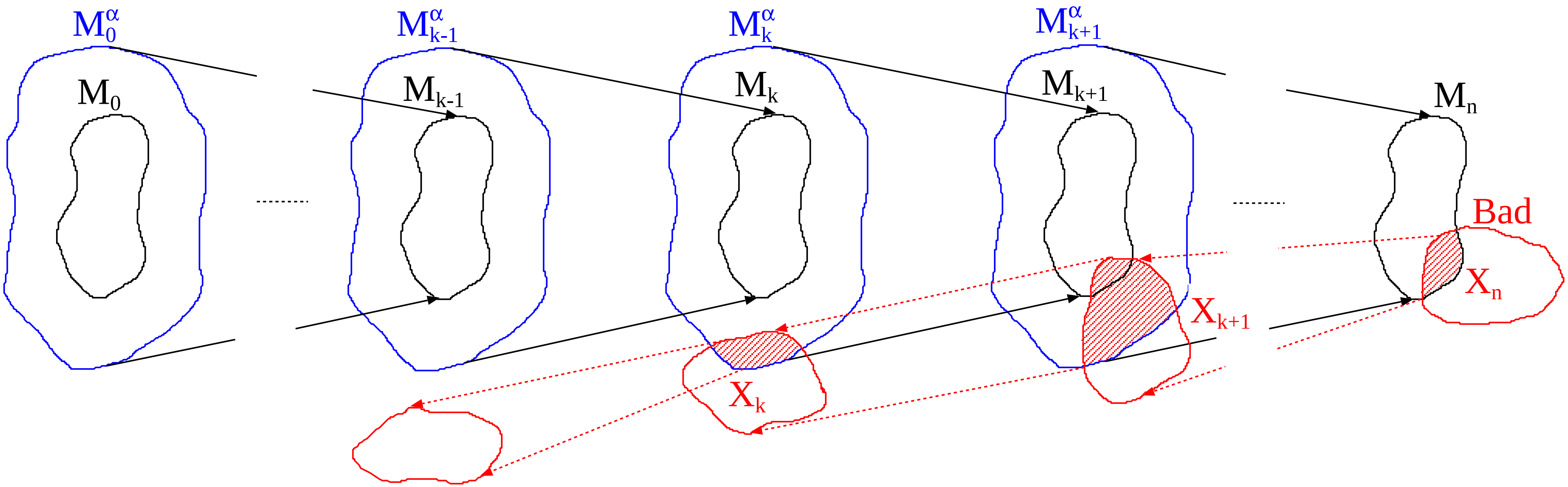
    $$q_1 \simeq_k q_2 \text{ iff } L(A, q_1)^{\leq k} = L(A, q_2)^{\leq k}$$

    $L(A, q)^{\leq k}$: the set of words of length at most k accepted in A from q.

  - Equivalence wrt a set of predicate languages $\mathcal{P} = \{P_1, ..., P_n\}$:

    $$q_1 \simeq_{\mathcal{P}} q_2 \text{ iff } \forall 1 \leq i \leq n : L(A, q_1) \cap P_i \neq \emptyset \Leftrightarrow L(A, q_2) \cap P_i \neq \emptyset$$

Counterexample-Guided Refinement



❖ For abstraction based on bounded length languages: increment the bound.

❖ For predicate automata abstraction: take as predicates languages of all states of the last non-empty intersection of the forward and backward run:

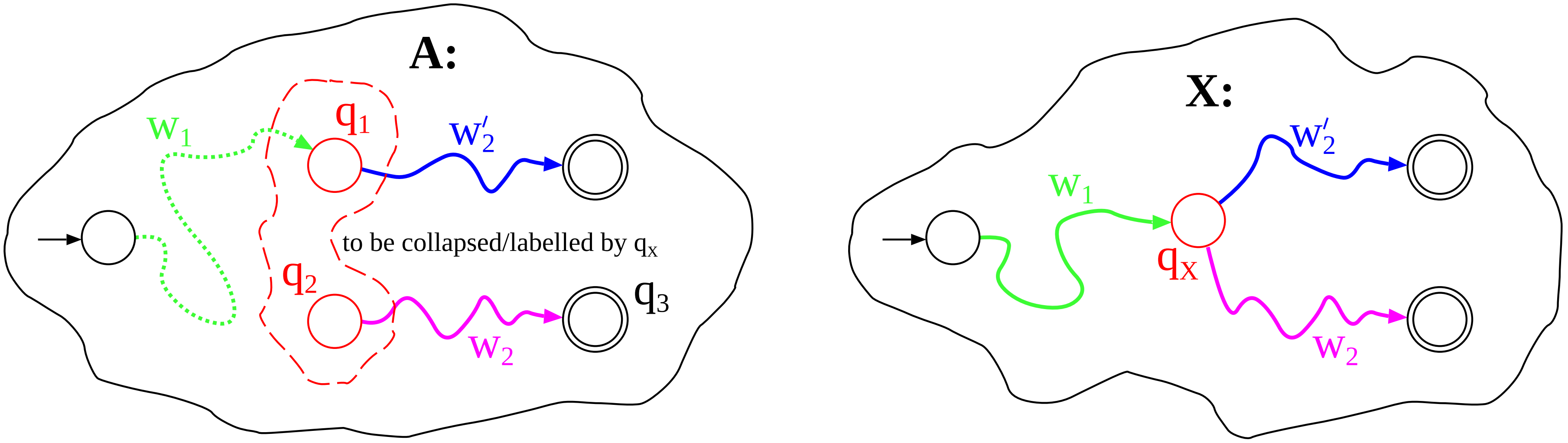
$$\mathcal{P}' = \mathcal{P} \cup \{L(X_k, q) \mid q \text{ is a state in } X_k\}.$$

Predicate Automata Abstraction: Refinement

Theorem:

Let A and X be two finite automata, and let $\mathcal{P}$ be a finite set of predicate languages such that $\forall q \in Q_X .\ L(X, q) \in \mathcal{P}$.
Then, if $L(A) \cap L(X) = \emptyset$, we have $L\big(\alpha_{\mathcal{P}}(A)\big) \cap L(X) = \emptyset$ too.

❖ Proof sketch: Assume $w \notin L(A) \wedge w \in L\big(\alpha_{\mathcal{P}}(A)\big) \cap L(X)$ with a minimum number of *jumps* needed to accept it in A – the last jump being $q_1 \rightsquigarrow q_2$ from where w_2 is accepted.



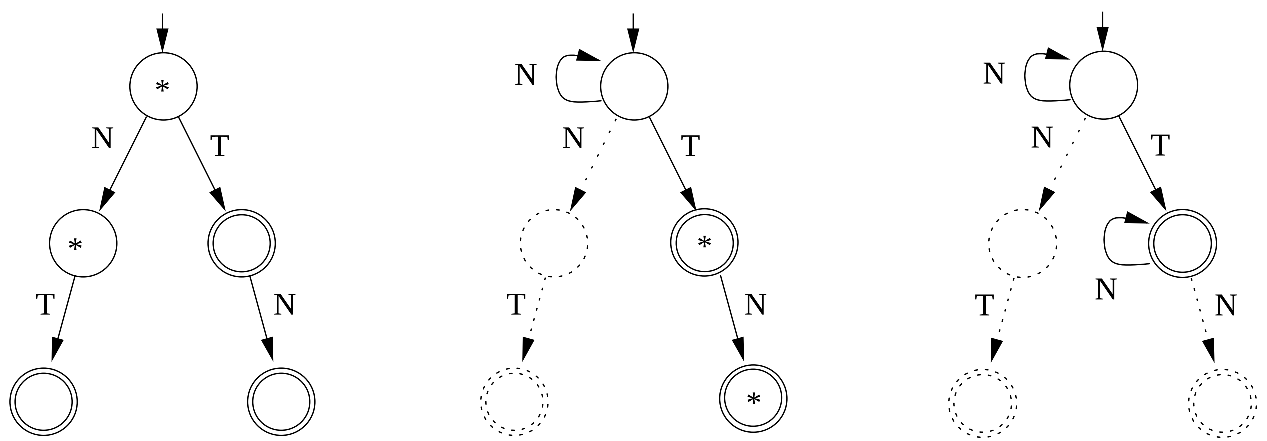
For $w_1 w_2'$, an even smaller number of jumps is needed which is a contradiction.

RMC by Automata Learning

❖ Use algorithms for learning automata from positive/negative samples of their languages to obtain an approximation of $\tau^*(I)$.

❖ Trakhtenbrot-Barzdin:

- based on having n-complete sets of positive and negative samples,
- can be obtained for length-preserving systems as $\tau^*(I^{\leq n})$,
- if $\tau^*(I^{\leq n}) \cap B \neq \emptyset$, error found,
- generalise the sample represented as a loop-free automaton by folding transitions back to compatible states, obtain an automaton A,



- if $\tau(L(A)) \subseteq L(A) \ \wedge \ I \subseteq L(A) \ \wedge \ L(A) \cap B = \emptyset$, verified; otherwise, increase n.

RMC by Automata Learning

- Angluin L* and variants:
  - membership query for a configuration w: check $w \in \tau^*(I^{=|w|})$.
  - equivalence query – replaced by $\tau(L(A)) \subseteq L(A) \;\wedge\; I \subseteq L(A) \;\wedge\; L(A) \cap B = \emptyset$.
- Guaranteed to terminate with the correct answer for length-preserving systems.

Regular Model Checking

String Analysis

RMC and String Analysis

[Yu, Alkhalaf, Bultan, Ibarra et al 08–17]

❖ Deterministic finite automata from MONA with transitions encoded using MTBDDs used for representing sets of strings that may appear in string variables of a program.

❖ Program statements (concatenation, replacement, ...) implemented as specialised automata operations.

❖ Non-refinable widening – collapsing states considered equal:

- states having the same language,
- states having a common access string (for non-sink states),
- closed under transitivity.

❖ Implemented in the STRANGER tool.

❖ Applied for finding XSS vulnerabilities in php-based web applications.

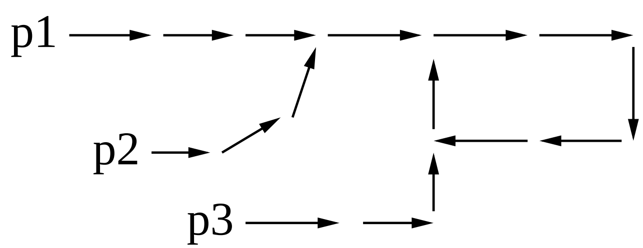
Regular Model Checking

Tricks

RMC and Programs with 1-Selector-Linked Structures

[Bouajjani, Habermehl, Moro, V. 05]

❖ Heap structures (even with 1 selector) are complex:



❖ Use pairs of from-to markers m_f/m_t:

$$p1 \to\to\to n_t \to m_t \to\to\to\to\to h_t \to m_f \mid p2 \to\to\to n_f \mid p3 \to\to\to h_f$$

❖ Pointer operations expressible by transducers up to marker elimination:

- $\mid y\ m_t \to\to \cdots \to \bot \mid x \to\to \cdots \to m_f \mid$ is changed to
  $\mid x \to\to \cdots \to y \to\to \cdots \to \bot \mid$,
- Not rational!
- Move letter-by-letter and use widening to converge: overapproximation.
- Use automata surgery instead of transducers.

RMC and Programs with 1-Selector-Linked Structures

❖ RMC can overapproximate sets of reachable configurations at any line, including loop invariants.

❖ Basic memory safety checked directly by the transducers of the program statements:

- no garbage is created,
- no null pointer dereferences,
- no undefined pointer dereferences.

❖ More complex properties can be checked from the invariants.

❖ Generating and checking code (test harness) can be added to the original code to check more complex properties: transforming the checks to error-line reachability.

❖ Sometimes, one may use special markers injected into random positions:

- e.g., when checking a function for reversing lists,
- one may check whether $bgn\ l \rightarrow^* fst \rightarrow snd \rightarrow^* end \rightarrow \perp$ gets transformed to $end\ l \rightarrow^* snd \rightarrow fst \rightarrow^* bgn \rightarrow \perp$.

RMC and Checking Liveness

❖ For length-preserving systems, liveness can be reduced to checking reachability:

- Choose any configuration $a_1 a_2 \ldots a_n$ as a candidate for the beginning of a loop.
- Double every symbol: $a_1 a_1 a_2 a_2 \ldots a_n a_n$.
  - In order to avoid words of the form $w.w$.
- Go on execution on the "red" symbols: $a_1 \tau'(a_1) a_2 \tau'(a_2) \ldots a_n \tau'(a_n)$.
- Check whether the system can get back to $a_1 a_1 a_2 a_2 \ldots a_n a_n$.

❖ Monitoring via some property automaton can be done within the transducer implementing the transition relation.

❖ More general approaches have been proposed, covering even the non-length preserving case. [Bouajjani, Legay, Wolper 05], [Vardhan, Sen, Viswanathan, Agha 05]

Regular Model Checking

Extensions, Improvements

RMC: Extensions, Improvements

- *Omega* regular model checking:
  - based on variants of Büchi automata,
  - systems with real-valued variables, liveness checking.
    [Boigelot, Bouajjani, Legay, Wolper]
- (Abstract) regular *tree* model checking:
  - based on variants of tree automata (TAs),
  - parametric protocols with tree topology,
  - shape analysis for programs with complex dynamic data structures.
- R(T)MC based on non-deterministic automata:
  - simulation-based minimisation,
  - antichain-based inclusion checking.

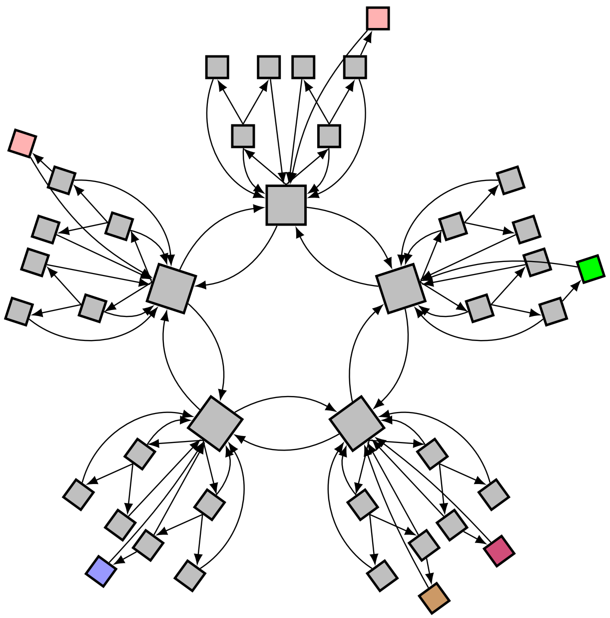
ARTMC and Shape Analysis

❖ Can handle programs with complex dynamic data structures: various forms of singly- and doubly-linked lists, trees, skip lists, structures with additional links, nested, combined, and shared structures.

❖ Based on forest automata (FAs):

- tuples of TAs,
- graphs decomposed to tuples of trees whose leaves can refer to roots,
- symbols can be nested FAs describing repeated substructures.

[Holik, Hruska, Lengal, Rogalewicz, Simacek, V.]



ARTMC: Shape Analysis with Backward Runs

[Holik, Hruska, Lengal, Rogalewicz, V. 17]

❖ Backward execution of a program along a possible counterexample trace:

- possible for classic ARTMC but originally not possible on (nested) FAs,
- required reversion of various meta-operations on FAs used in the forward analysis:
  - folding/unfolding of nested FAs, splitting/merging TAs, reordering of TAs.

❖ Rare in the area of shape analysis in general!

❖ The introduction of the backward execution enabled:

- automated checking of possibly spurious counterexamples,
- automated refinement of the abstraction used,
- for the first time allowed both fully automated sound verification and precise bug detection in programs with some complex data structures,
- the data structures could contain data from finite domains (possibly used as markers for verification of complex properties such as element preservation, sortedness, ...).

Acknowledgement

During the years, the research of the author of the slides on the various aspects of the presented subject was supported by a number of projects, the last being the following:

ROBUST: Verification and Bug Hunting for Advanced Software,
Czech Science Foundation,
project No. 17-12465S.