Regular Model Checking

Tomáš Vojnar

Faculty of Information Technology
Brno University of Technology

Plan of the Lecture

- From finite-state to infinite-state model checking.
- The basic idea of regular model checking.
- Computing closures of transition relations in regular model checking.
- Regular tree model checking.
- Nondeterministic automata in regular (tree) model checking.

From Finite-state to Infinite-state Model Checking

Model Checking

[Clarke, Emerson 81], [Quielle, Sifakis 81]

- An algorithmic approach of checking whether a model M of a system satisfies a certain correctness specification φ when started from some initial state s:

$$M, s \models \varphi$$

- Typically based on a **systematic exploration of the state space** of M.

- Models of systems
  - can be built in various specialised modelling languages (process algebras, Petri nets, Promela, SMV, ...), or
  - source descriptions of analysed systems (in C, Java, Verilog, VHDL, ...) can directly be used.

- Correctness specifications:
  - formulae in temporal logics (LTL, CTL, CTL*, μ-calculus, ...),
  - assertions in the source code (`assert()`), progress labels, ...

Model Checking

- Advantages:
  - highly automatable,
  - can provide counterexamples (diagnostic/debugging information).

- The biggest problem is the **state explosion problem**.
  - Efficient storage of state spaces (hierarchical storage of states, BDDs, ...).
  - State space reductions (symmetries, partial-order reduction, ...).
  - Abstraction, counterexample-guided abstraction refinement (CEGAR).
  - Compositional methods, assume-guarantee reasoning.

- Supported by many tools, including industrial-strength tools (Spin, SMV, RuleBase, Blast, JPF, Slam, ...).

- Traditional model checking concentrated on systems with large, but finite state spaces, but many systems are infinite-state.

Sources of Infinity

- Unbounded communication queues (channels), unbounded waiting queues.
- Unbounded push-down stacks: recursion.
- Unbounded counters, unbounded capacity of places in Petri nets.
- Continuous variables: time, temperature, ...
- Unbounded dynamic creation of threads, dynamic allocation of memory structures (lists, trees, ...).
- Parameterisation: parametric bounds of queues, counters, ..., parametric numbers of components or 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: $x \in \{5, 6, 7, ...\} \rightsquigarrow x \geq 5$,
- 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** (sometimes not even semi-decidable).

- There may be identified (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 are used:
  - termination is not guaranteed,
  - an indefinite answer may be returned, or
  - an intervention of the user is needed.

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 $\leadsto$ a word w over a suitable alphabet Σ,
  - a set of configurations $\leadsto$ 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) $\leadsto$ a regular 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 $\leadsto$ check that $\tau^*(Init) \cap Bad = \emptyset$,
  - implies a need to compute $\tau^*(Init)$.

Regular Model Checking: Applicability

- Communication protocols.
  - FIFO channels systems / cyclic rewrite systems.
- Sequential programs with recursive procedure calls.
  - Pushdown systems / prefix rewrite systems.
- Counter systems, Petri nets.
  - Various unbounded/parameterised systems may be (automatically) translated to counter systems.
- Programs with (unbounded) dynamic linked data structures: lists, cyclic lists, shared lists. [Bouajjani, Habermehl, Vojnar, Moro 05]
- Parameterized networks of identical processes: mutual exclusion protocols, cache coherence protocols, ..., pipelined microprocessors. [Charvát, Smrčka, Vojnar 14].

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

- ...

Regular Model Checking: Applicability

- Communication protocols.
  - FIFO channels systems / cyclic rewrite systems.
- Sequential programs with recursive procedure calls.
  - Pushdown systems / prefix rewrite systems.
- Counter systems, Petri nets.
  - Various unbounded/parameterised systems may be (automatically) translated to counter systems.
- Programs with (unbounded) dynamic linked data structures: lists, cyclic lists, shared lists. [Bouajjani, Habermehl, Vojnar, Moro 05]
- Parameterized networks of identical processes: mutual exclusion protocols, cache coherence protocols, ..., pipelined microprocessors. [Charvát, Smrčka, Vojnar 14].

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

- ...

Example: the Szymanski's Protocol

❖ A typical example of a parameterized protocol: the mutual exclusion protocol for N processes due to Szymanski—the pseudocode for process i (a bit idealised):

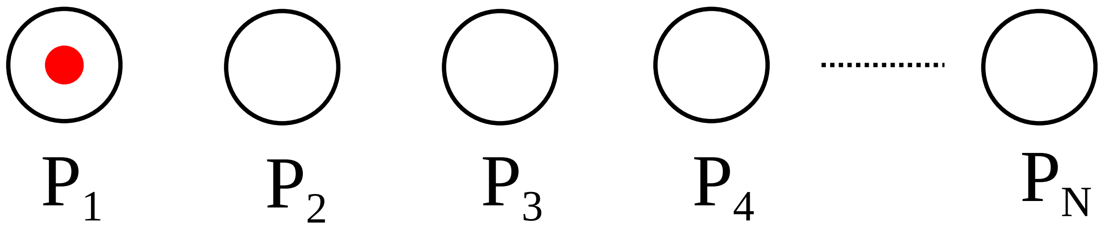
1: await $\forall j: j \neq i \Rightarrow \neg s_j$;
2: $w_i,\ s_i := true,\ true$;
3: if $\exists j: j \neq i \Rightarrow (pc_j \neq 1 \wedge \neg w_j)$
then $s_i := false$; goto 4;
else $w_i := false$; goto 5;
4: await $\exists j: j \neq i \Rightarrow (s_j \wedge \neg w_j)$
then $w_i,\ s_i := false,\ true$;
5: await $\forall j: j \neq i \Rightarrow \neg w_j$;
6: await $\forall j: j < i \Rightarrow \neg s_j$;
7: $s_i := false$; goto 1;

Too complex to be used as a running example...

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 can pass it to the right.

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

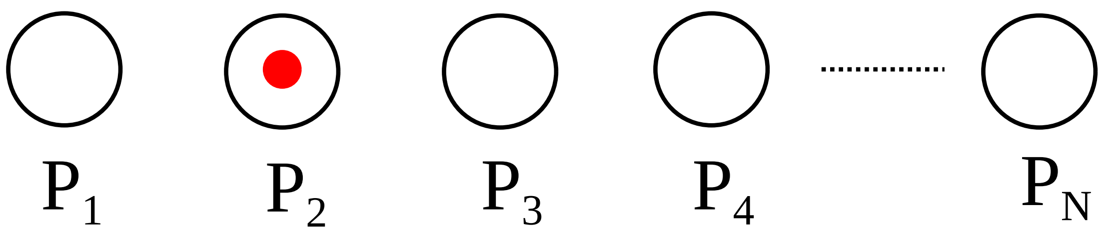


- Check that the token cannot disappear nor duplicate.

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 can pass it to the right.

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

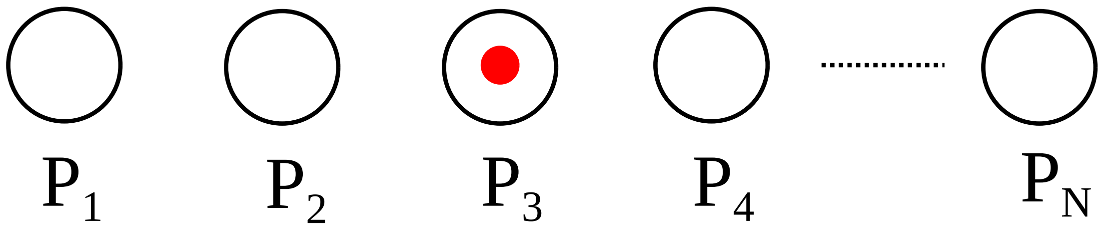


- Check that the token cannot disappear nor duplicate.

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 can pass it to the right.

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

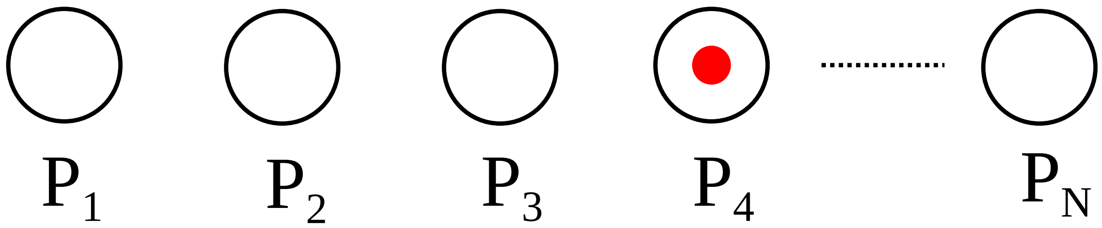


- Check that the token cannot disappear nor duplicate.

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 can pass it to the right.

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



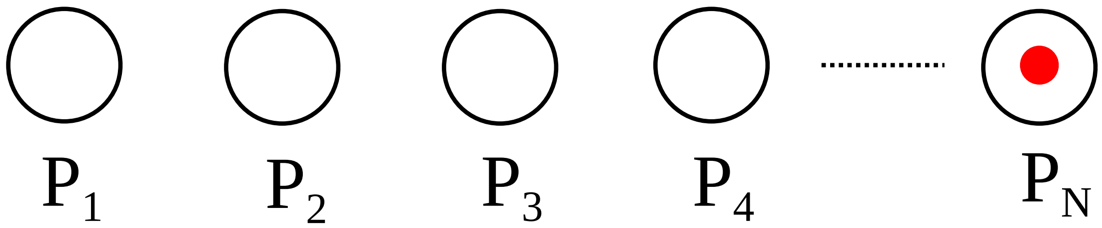
- Check that the token cannot disappear nor duplicate.

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 can pass it to the right.

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

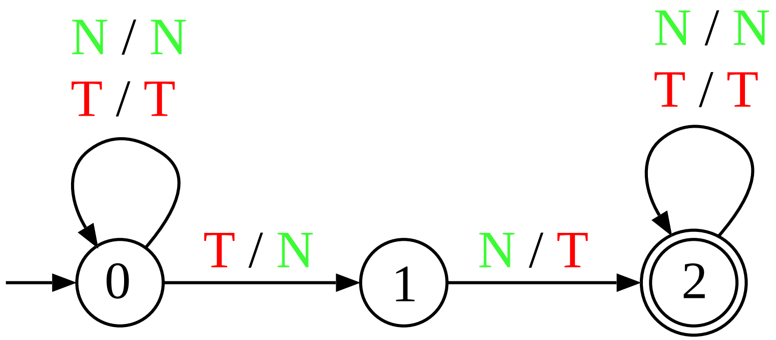


❖ Check that the token cannot disappear nor duplicate.

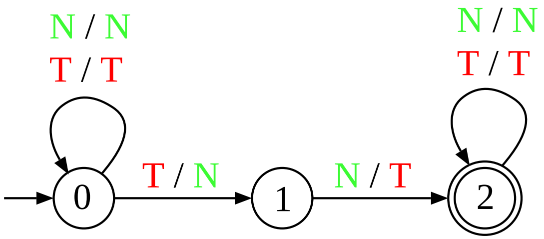
Example: A Simple Token Passing

❖ An **encoding** of the simple token passing protocol for the needs of regular model checking:

- the alphabet: $\Sigma = \{T, N\}$,
- all configurations: words from Σ^*,
- 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:



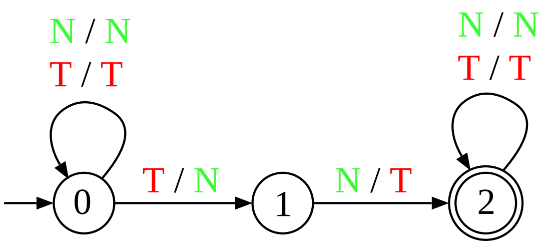
Example: A Simple Token Passing



- An application of the transducer on a sample 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

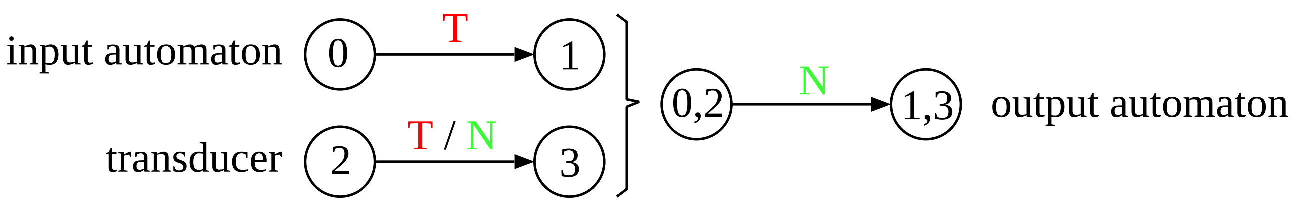


- An application of the transducer on a sample configuration:

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

- An application of the transducer on all initial configurations:

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

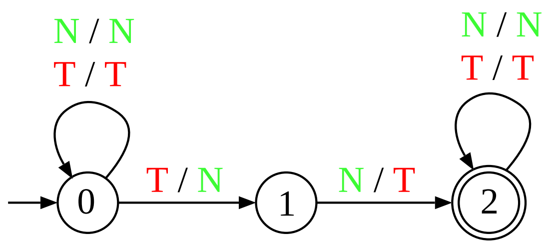


- A simple iterative computation of all reachable configurations will **never converge** to the desired set $N^*\ T\ N^*$.

Example: A Simple Token Passing

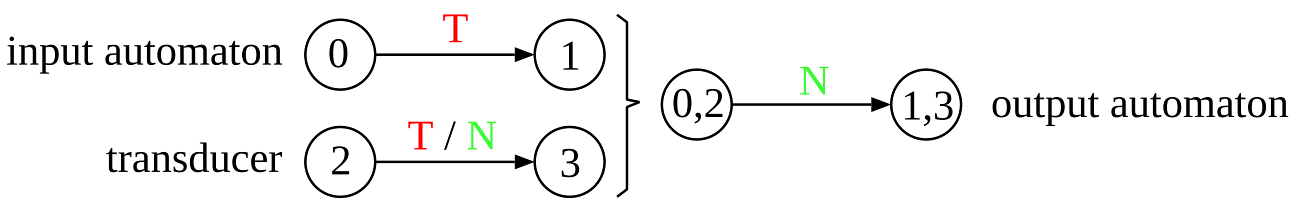
❖ An application of the transducer on a sample configuration:

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



❖ An application of the transducer on all initial configurations:

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



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

- We need special (accelerated) ways for computing $\tau^*(Init)$.

Regular Model Checking:

Computing Closures

RMC: Computing Closures

The task: compute $\tau^*(Init)$.

❖ Problems to face:

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

RMC: Computing Closures

The task: compute $\tau^*(Init)$.

❖ Problems to face:

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

❖ Solutions:

- Special purpose constructions: LCS, PDS, classes of arithmetical relations, ...
- General purpose constructions:
  - extrapolation (widening) [Bouajjani, Touili], [Wolper, Boigelot, Legay],
  - merging states wrt. the history of their creation, [Abdulla, Nilsson, Jonsson, d'Orso]
  - abstract regular model checking, [Bouajjani, Habermehl, Vojnar]
  - learning of automata, [Habermehl, Vojnar], [Vardhan, Sen, Viswanathan, Agha]
  - ...

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 function α on automata.
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 function α on automata.
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, define a refinement of α which 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 function α on automata.
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, define a refinement of α which excludes θ and goto (2).

Abstractions Based on State Collapsing

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

$\Rightarrow \quad L(A) \subseteq L(\alpha(A))$

Abstractions Based on State Collapsing

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

  $\Rightarrow \quad L(A) \subseteq L(\alpha(A))$

- We consider several different equivalence relations on automata states, including:
  - equivalence wrt. languages of words of a bounded length k:

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

    where $L(A, q)^{\leq k}$ is the set of words of length at most k accepted in A when starting from q.
  - equivalence wrt. a set of predicate languages $\mathcal{P} = \{P_1, ..., P_n\}$:

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

Abstractions Based on State Collapsing

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

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

- We consider several different equivalence relations on automata states, including:
  - equivalence wrt. languages of words of a bounded length k:

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

    where $L(A, q)^{\leq k}$ is the set of words of length at most k accepted in A when starting from q.
  - equivalence wrt. a set of predicate languages $\mathcal{P} = \{P_1, ..., P_n\}$:

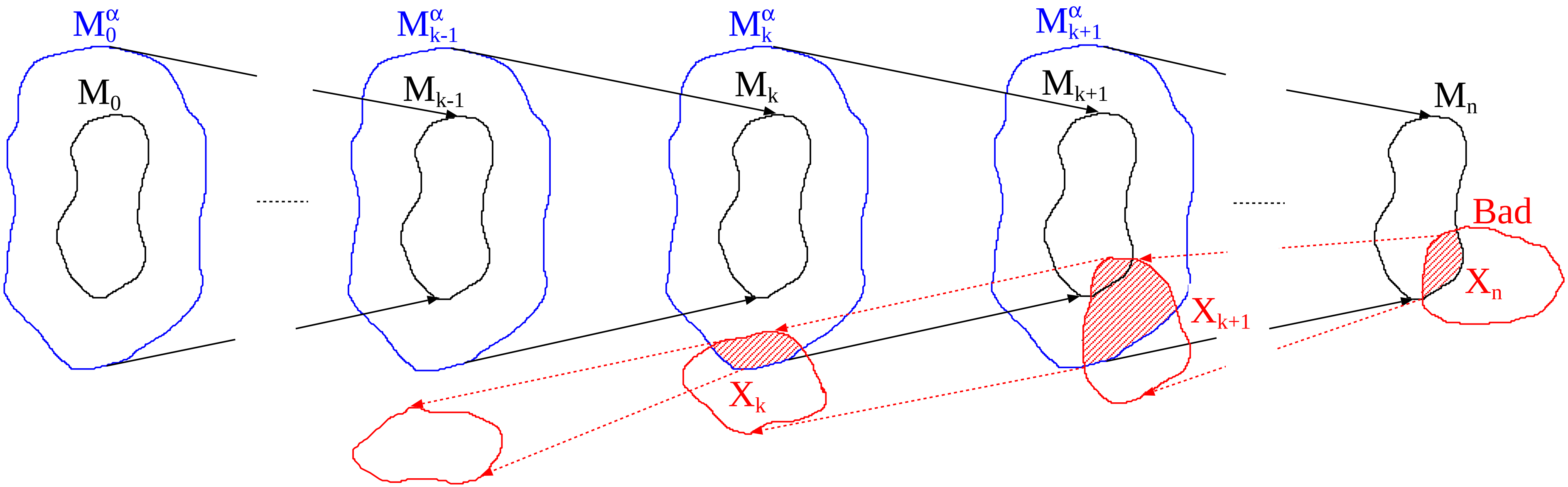
    $$q_1 \simeq_{\mathcal{P}} q_2 \text{ iff } \forall 1 \leq i \leq n : L(A, q_1) \cap P_i \neq \emptyset \Leftrightarrow L(A, q_2) \cap P_i \neq \emptyset$$

- These equivalence relations are finite-index.
  - Indeed, there are finitely many words of length up to some k as well as finitely many subsets of $\mathcal{P}$ of predicates that may hold at a certain state.

  $\Rightarrow$ The implied abstraction α has a finite image (defines a finite abstract domain).

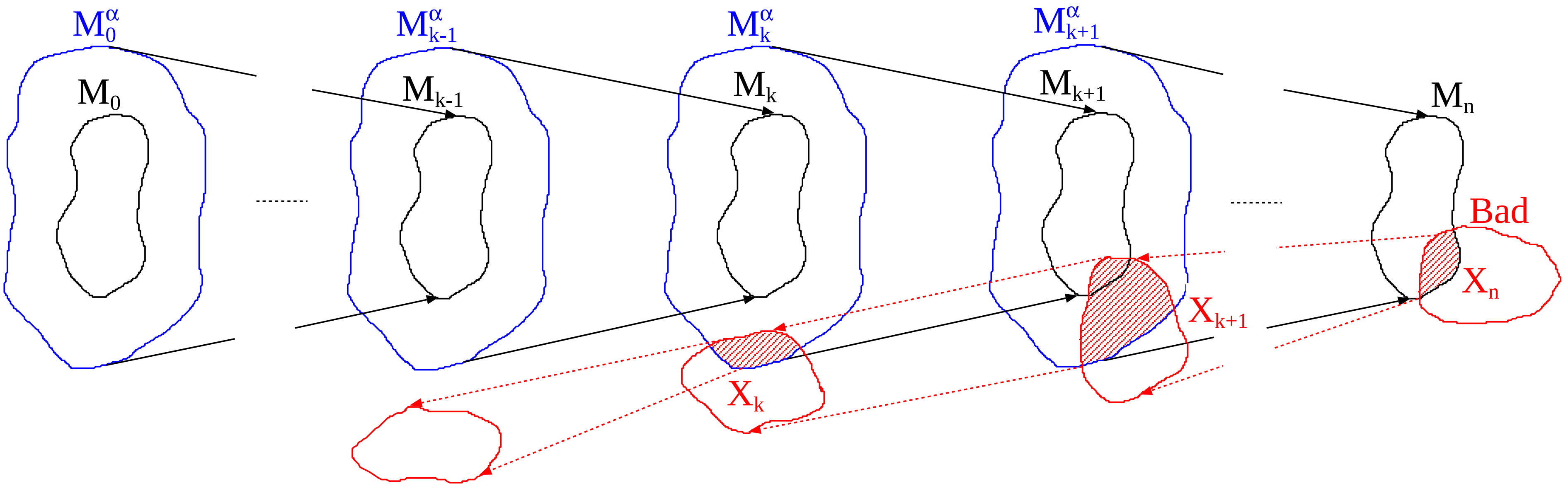
  $\Rightarrow$ Abstract fixpoint computations always terminate.

Counterexample-Guided Refinement



- For abstraction based on bounded length languages, increment the bound.
- For predicate automata abstraction, take $\mathcal{P}' = \mathcal{P} \cup \{L(X_k, q) \mid q \text{ is a state in } X_k\}$.

Counterexample-Guided Refinement



- For abstraction based on bounded length languages, increment the bound.
- For predicate automata abstraction, take $\mathcal{P}' = \mathcal{P} \cup \{L(X_k, q) \mid q \text{ is a state in } X_k\}$.

Theorem:

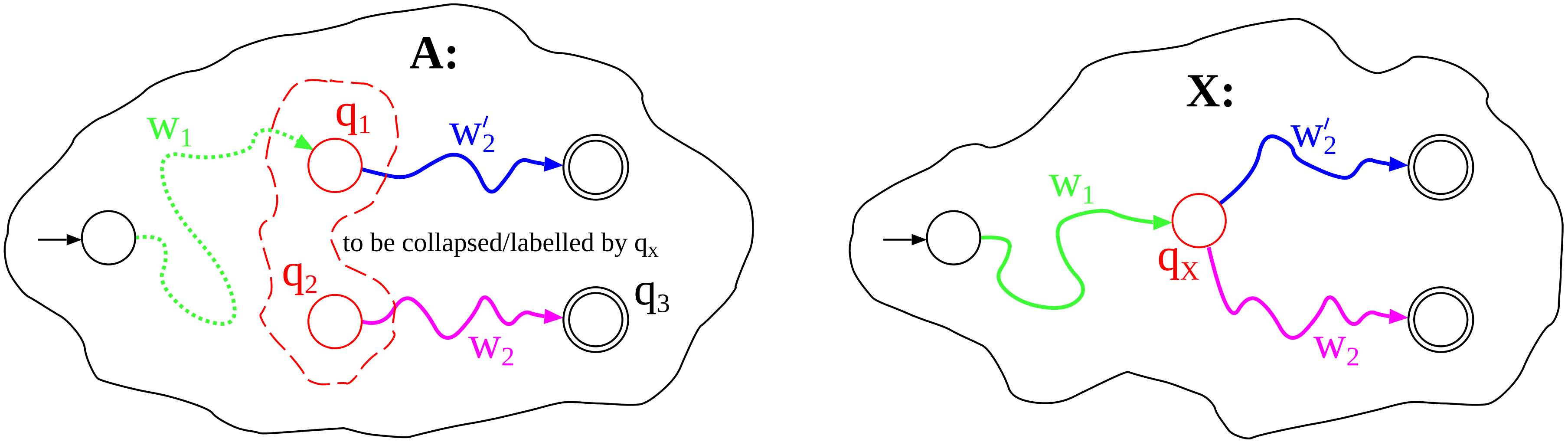
Let A and X be two finite automata, and let $\mathcal{P}$ be a finite set of predicate languages such that $\forall q \in Q_X .\ L(X, q) \in \mathcal{P}$.
Then, if $L(A) \cap L(X) = \emptyset$, we have $L\big(\alpha_{\mathcal{P}}(A)\big) \cap L(X) = \emptyset$ too.

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 and Programs with 1-Selector-Linked Structures

[Bouajjani, Habermehl, Moro, Vojnar 05]

- Heap configurations encoded as words:
  - Uninterrupted list segments of length n: sequences of n symbols $\rightarrow$, divided by $|$.
  - A null successor: $\perp$.
  - Variables: put a variable into the word on the place it points to.
  - Two special sections of the word for null and undefined variables.
  - Marker pairs (m_{from},m_{to}) encode non-linear configurations: sharing and cicles.

RMC and Programs with 1-Selector-Linked Structures

[Bouajjani, Habermehl, Moro, Vojnar 05]

- Heap configurations encoded as words:
  - Uninterrupted list segments of length n: sequences of n symbols $\rightarrow$, divided by $|$.
  - A null successor: $\perp$.
  - Variables: put a variable into the word on the place it points to.
  - Two special sections of the word for null and undefined variables.
  - Marker pairs (m_{from},m_{to}) encode non-linear configurations: sharing and cicles.
- Program statements translated automatically to transducers.

RMC and Programs with 1-Selector-Linked Structures

[Bouajjani, Habermehl, Moro, Vojnar 05]

- Heap configurations encoded as words:
  - Uninterrupted list segments of length n: sequences of n symbols $\rightarrow$, divided by $|$.
  - A null successor: $\perp$.
  - Variables: put a variable into the word on the place it points to.
  - Two special sections of the word for null and undefined variables.
  - Marker pairs (m_{from},m_{to}) encode non-linear configurations: sharing and cicles.
- Program statements translated automatically to transducers.
- To stay with a finite number of markers:
  - When they are not-needed, they are re-claimed by shifting the appropriate parts of the words such that they merge.
  - A transducer can encode a single step of the shifting, ARMC used to compute the effect of iterating this step.
  - Merging cannot be implemented as a regular relation (and hence a transducer)!

List Reversion: An Example of a Run

1: $x = null;$

2: while $(l \,!= null)$ $\{$ $//$ i.e. if $(l \,!= null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y;\}$ // i.e. $l = y$; goto 2;

7: $l = x;$

$$1 \quad | \quad xy \quad | \quad\quad | \quad l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot \quad |$$

List Reversion: An Example of a Run

1: $x = null;$

2: while $(l \mathrel{!=} null)$ $\{$ $//$ i.e. if $(l \mathrel{!=} null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$

List Reversion: An Example of a Run

1: $x = null;$

2: while $(l \,!= null)$ $\{$ // i.e. if $(l \,!= null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$

List Reversion: An Example of a Run

1: $x = null;$
2: while $(l\ != null)$ { $//$ i.e. if $(l\ != null)$ goto 3; else goto 7;
3: $\quad y = l \rightarrow next;$
4: $\quad l \rightarrow next = x;$
5: $\quad x = l;$
6: $\quad l = y;\}$ // i.e. $l = y$; goto 2;
7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$
3	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$

List Reversion: An Example of a Run

1: $x = null;$

2: while $(l \; != null)$ $\{$ $//$ i.e. if $(l \; != null)$ goto 3; else goto 7;

3: $\quad y = l \to next;$

4: $\quad l \to next = x;$

5: $\quad x = l;$

6: $\quad l = y;$ $\}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \to\to\to\to\to\to \bot$
2	y	x	$l \to\to\to\to\to\to \bot$
3	y	x	$l \to \; \to\to\to\to\to \bot$

List Reversion: An Example of a Run

1: $x = null;$

2: while $(l \mathrel{!}= null)$ { // i.e. if $(l \mathrel{!}= null)$ goto 3; else goto 7;

3: $y = l \rightarrow next;$

4: $l \rightarrow next = x;$

5: $x = l;$

6: $l = y;$ } // i.e. $l = y$; goto 2;

7: $l = x;$

$$
\begin{array}{c|c|c|l|}
1 & xy & & l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot \\
2 & y & x & l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot \\
3 & y & x & l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot \\
4 & & x & l \rightarrow y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot
\end{array}
$$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \; != null)$ { $//$ i.e. if $(l \; != null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y;$ } // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
3	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
4		x	$l \rightarrow y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \,!{=}\, null)$ $\{$ $//$ i.e. if $(l \,!{=}\, null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y;\}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$	
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$	
3	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$	
4		x	$l \rightarrow y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$	
5		x	$l \rightarrow \perp \mid y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$	

List Reversion: An Example of a Run

1: $x = null$
2: while $(l \,!= null)$ { // i.e. if $(l \,!= null)$ goto 3; else goto 7;
3: $\quad y = l \to next;$
4: $\quad l \to next = x;$
5: $\quad x = l;$
6: $\quad l = y;$ } // i.e. $l = y$; goto 2;
7: $l = x;$

1	xy		$l \to\to\to\to\to\to \perp$
2	y	x	$l \to\to\to\to\to\to \perp$
3	y	x	$l \to\to\to\to\to\to \perp$
4		x	$l \to y \to\to\to\to\to \perp$
5		x	$l \to \perp \mid y \to\to\to\to\to \perp$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l\ != null)$ $\{$ $//$ i.e. if $(l\ != null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
3	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
4		x	$l \rightarrow y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
5		x	$l \rightarrow \bot \mid y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	
6			$xl \rightarrow \bot \mid y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$	

List Reversion: An Example of a Run

1: $x = null$

2: while $(l\ != null)$ $\{$ $//$ i.e. if $(l\ != null)$ goto 3; else goto 7;

3: $\quad y = l \to next;$

4: $\quad l \to next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \to\to\to\to\to\to \bot$
2	y	x	$l \to\to\to\to\to\to \bot$
3	y	x	$l \to\to\to\to\to\to \bot$
4		x	$l \to y \to\to\to\to\to \bot$
5		x	$l \to \bot \mid y \to\to\to\to\to \bot$
6			$xl \to \bot \mid y \to\to\to\to\to \bot$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \,!= null)$ $\{$ $//$ i.e. if $(l \,!= null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

1	xy		$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
2	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
3	y	x	$l \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
4		x	$l \rightarrow y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
5		x	$l \rightarrow \bot \mid y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
6			$xl \rightarrow \bot \mid y \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
2			$x \rightarrow \bot \mid ly \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \bot$
			$etc.$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \mathrel{!}= null)$ $\{$ $//$ i.e. if $(l \mathrel{!}= null)$ goto 3; else goto 7;

3: $\quad y = l \to next;$

4: $\quad l \to next = x;$

5: $\quad x = l;$

6: $\quad l = y;\}$ // i.e. $l = y$; goto 2;

7: $l = x;$

$$3 \quad | \qquad | \qquad | \quad x \to\to\to \bot \mid l y \to\to\to \bot \quad |$$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l\ != null)$ { // i.e. if $(l\ != null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y;$ } // i.e. $l = y$; goto 2;

7: $l = x;$

$$3 \quad | \quad\quad | \quad\quad | \quad x \rightarrow\rightarrow\rightarrow \bot \mid ly \rightarrow\rightarrow\rightarrow \bot \quad |$$

$$4 \quad | \quad\quad | \quad\quad | \quad x \rightarrow\rightarrow\rightarrow \bot \mid l \rightarrow y \rightarrow\rightarrow \bot \quad |$$

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \,!= null)$ $\{$ $//$ i.e. if $(l \,!= null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y;$ $\}$ // i.e. $l = y$; goto 2;

7: $l = x;$

$$3 \quad | \quad | \quad | \quad x \rightarrow\rightarrow\rightarrow \perp | \, ly \rightarrow\rightarrow\rightarrow \perp \quad |$$

$$4 \quad | \quad | \quad | \quad x \rightarrow\rightarrow\rightarrow \perp | \, l {\rightarrow} y \rightarrow\rightarrow \perp \quad |$$

List Reversion: An Example of a Run

```
1:  x = null
2:  while (l != null) { // i.e. if (l != null) goto 3; else goto 7;
3:      y = l → next;
4:      l → next = x;
5:      x = l;
6:      l = y; } // i.e. l = y; goto 2;
7:  l = x;
```

$$3 \quad | \quad | \quad | \quad x \to\to\to \bot \mid ly \to\to\to \bot \quad |$$

$$4 \quad | \quad | \quad | \quad x \to\to\to \bot \mid l{\to}y \to\to \bot \quad |$$

$$5 \quad | \quad | \quad | \quad x m_t \to\to\to \bot \mid l{\to}\, m_f \mid y \to\to \bot \quad |$$

- Marker pairs (m_{from},m_{to}) allow us to encode:
  - non-linear configurations: in particular, sharing and circles,
  - when they are not-needed, they are re-claimed by shifting the appropriate parts of the words (non-regular!).

List Reversion: An Example of a Run

1: $x = null$

2: while $(l \,!= null)$ $\{$ // i.e. if $(l \,!= null)$ goto 3; else goto 7;

3: $\quad y = l \rightarrow next;$

4: $\quad l \rightarrow next = x;$

5: $\quad x = l;$

6: $\quad l = y; \}$ // i.e. $l = y$; goto 2;

7: $l = x;$

$$3 \quad | \quad\quad | \quad\quad | \quad x \rightarrow\rightarrow\rightarrow \bot \mid ly \rightarrow\rightarrow\rightarrow \bot \quad\quad |$$

$$4 \quad | \quad\quad | \quad\quad | \quad x \rightarrow\rightarrow\rightarrow \bot \mid l{\rightarrow}y \rightarrow\rightarrow \bot \quad\quad |$$

$$5 \quad | \quad\quad | \quad\quad | \quad xm_t \rightarrow\rightarrow\rightarrow \bot \mid l{\rightarrow}\, m_f \mid y \rightarrow\rightarrow \bot \quad |$$

$$5 \quad | \quad\quad | \quad\quad | \quad l \rightarrow x \rightarrow\rightarrow\rightarrow \bot \mid y \rightarrow\rightarrow \bot \quad\quad |$$

List Reversion: An Example of a Run

1: $x = null$
2: while $(l \mathrel{!=} null)$ { // i.e. if $(l \mathrel{!=} null)$ goto 3; else goto 7;
3: $\quad y = l \rightarrow next;$
4: $\quad l \rightarrow next = x;$
5: $\quad x = l;$
6: $\quad l = y;$ } // i.e. $l = y$; goto 2;
7: $l = x;$

$$3 \quad | \quad | \quad | \quad x \rightarrow\rightarrow\rightarrow \perp \mid ly \rightarrow\rightarrow\rightarrow \perp \quad |$$

$$4 \quad | \quad | \quad | \quad x \rightarrow\rightarrow\rightarrow \perp \mid l \rightarrow y \rightarrow\rightarrow \perp \quad |$$

$$5 \quad | \quad | \quad | \quad xm_t \rightarrow\rightarrow\rightarrow \perp \mid l \rightarrow m_f \mid y \rightarrow\rightarrow \perp \quad |$$

$$5 \quad | \quad | \quad | \quad l \rightarrow x \rightarrow\rightarrow\rightarrow \perp \mid y \rightarrow\rightarrow \perp \quad |$$

$$etc.$$

$$8 \quad | \quad | \quad y \quad | \quad xl \rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow \perp$$

List Reversion: Verification

- Initial configurations: $Init = (1 \mid xy \mid \mid l \rightarrow\rightarrow^* \perp \mid)$.

List Reversion: Verification

- Initial configurations: $Init = (1 \mid xy \mid \mid l \to\to^* \perp \mid)$.
- ARMC can be used to overapproximate reachable configurations at any line: including the postcondition $\tau^*(Init) = (8 \mid \mid y \mid xl \to\to^* \perp \mid)$ and loop invariants.

List Reversion: Verification

❖ Initial configurations: $Init = (1 \mid xy \mid \mid l \to\to^* \perp \mid)$.

❖ ARMC can be used to overapproximate reachable configurations at any line: including the postcondition $\tau^*(Init) = (8 \mid \mid y \mid xl \to\to^* \perp \mid)$ and 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.

List Reversion: Verification

- Initial configurations: $Init = (1 \mid xy \mid \mid l \rightarrow\rightarrow^* \perp \mid)$.
- ARMC can be used to overapproximate reachable configurations at any line: including the postcondition $\tau^*(Init) = (8 \mid \mid y \mid xl \rightarrow\rightarrow^* \perp \mid)$ and 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 that can be checked:
  - The result is a single, unshared, acyclic list.
  - The list is really reversed, no elements are lost/added.
  - For that, one may use special markers injected into the initial configuration, e.g.: $bgn\ l \rightarrow^* fst \rightarrow snd \rightarrow^* end \rightarrow \perp$ leads to $end\ l \rightarrow^* snd \rightarrow fst \rightarrow^* bgn \rightarrow \perp$
  - Note that injection at random positions can be used, and the verification then checks correctness for all possible positions of the markers.
  - One can also add a test harness: additional code which generates the input data structures and/or checks the output.

Regular Tree Model Checking

Regular Tree Model Checking

[Pnueli, Shahar 00], [Bouajjani, Touili 02], [Abdulla, d'Orso et al 02, 05]
[Bouajjani, Habermehl, Rogalewicz, Vojnar 05]

❖ A generalisation of RMC to systems with a tree-like topology of configurations:

- a configuration $\leadsto$ a tree (term) t over a suitable *ranked* alphabet Σ,
- a set of configurations $\leadsto$ a regular *tree* language
  - usually described by a finite-state *tree* automaton A.
- an action (transition) $\leadsto$ a regular (regularity-preserving) *tree* relation τ
  - usually described by a finite-state *tree* transducer T.

Regular Tree Model Checking

- Safety verification $\leadsto$ check that $\tau^*(Init) \cap Bad = \emptyset$,
  - implies a need to compute $\tau^*(Init)$.

- Computing closures in RTMC—generalisations of:
  - extrapolation (widening), [Bouajjani, Touili]
  - merging of states wrt. the history of their creation, [Abdulla, d'Orso, Legay, Rezine]
  - abstract regular tree model checking: [Bouajjani, Habermehl, Rogalewicz, Vojnar]
    - finite-*height* abstraction,
    - predicate *tree* automata abstraction.

RTMC: Applicability

- Verification of parameterised networks with a tree-like topology:
  - mutual exclusion, leader election, ...

- Verification of programs with complex dynamic linked data structures:
  - programs with doubly-linked lists, lists of lists, trees, skip-lists, trees with linked leaves ..., i.e., not only trees!,
  - configurations encoded into trees:
    - tree backbones and routing expressions, [Bouajjani, Habermehl, Rogalewicz, Vojnar ’06]
    - tuples of (nested) tree automata linked via references from leaves to roots – (boxed) forest automata: [Habermehl, Holík, Šimáček, Rogalewicz, Vojnar ’11]
      - less general – finite number of “far” pointers (e.g., not handles trees of linked leaves),
      - more scalable,
      - implemented in the Forester tool.

Nondeterministic Automata in Regular (Tree) Model Checking

AR(T)MC and Nondeterministic Automata?

- AR(T)MC based on deterministic (tree) automata:
  - easy minimisation leading to a unique canonical form,
  - easy language inclusion testing,
  - BUT determinisation costs time and makes automata grow.

- What about nondeterministic automata in AR(T)MC?
  - Almost everything works like in the deterministic case (abstraction, transduction).
  - No determinisation in the computation loop.
  - But, there are tasks to solve:
    - How to check language inclusion?
      - antichains, simulations, congruences (the latter not tried yet),
    - How to reduce the size of nondeterministic tree automata?
      - (bi-)simulation (mediated) quotienting.