Automatic Verification of Parameterized Data Structures

Jyotirmoy V. Deshmukh, E. Allen Emerson and Prateek Gupta

The University of Texas at Austin

Outline

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

Motivation

- Data structures: basic building blocks of software systems.
- Methods: programs operating on data structures.
- Traditional approach: check correctness up to bounded size.
- *Parameterized verification*: correctness for arbitrarily large sizes.
- Parameterized verification faces several difficulties!

Verifying programs operating on data structures

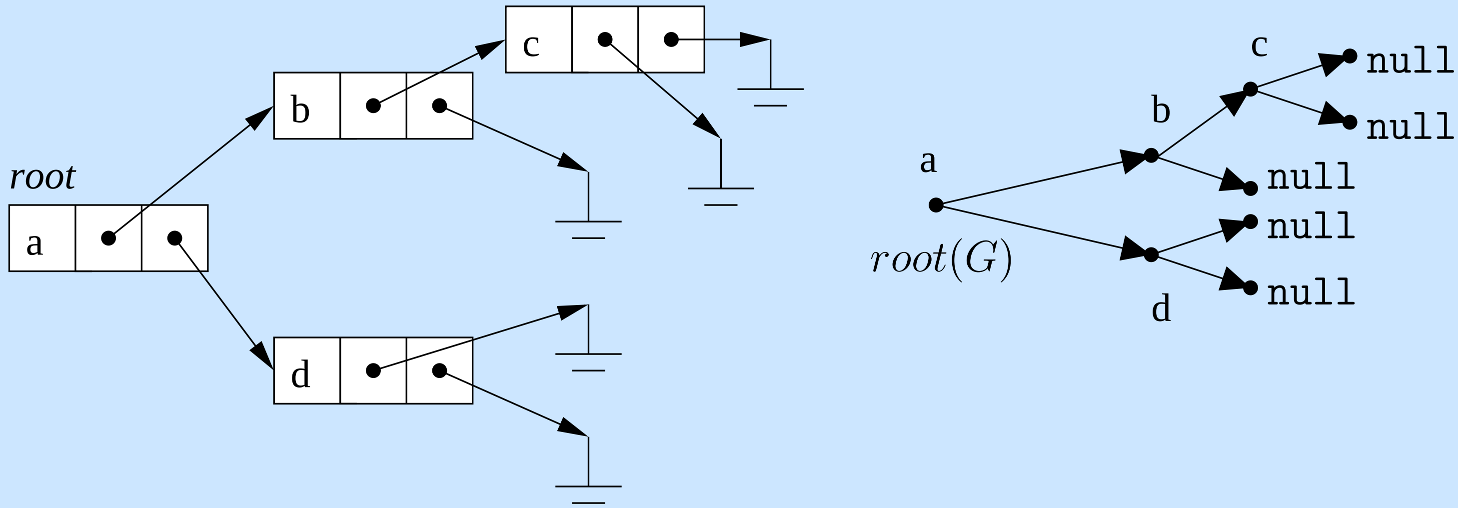
- Data structures:
  - may have arbitrarily large sizes.
  - may use pointers that range over arbitrarily large address space.
  - may use data values that range over unbounded domains.
- Parameterized correctness is generally undecidable.
- Decidable classes of programs face severe combinatorial explosion.

Potential Applications

- Verification of data structure libraries in C++, Java.
- File system manipulation routines.
- Memory management algorithms, *e.g.* garbage collection.
- Algorithms in SoC designs.

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

Correspondence between a data structure and a graph



Data Structure

Corresponding Graph

Problem Definition

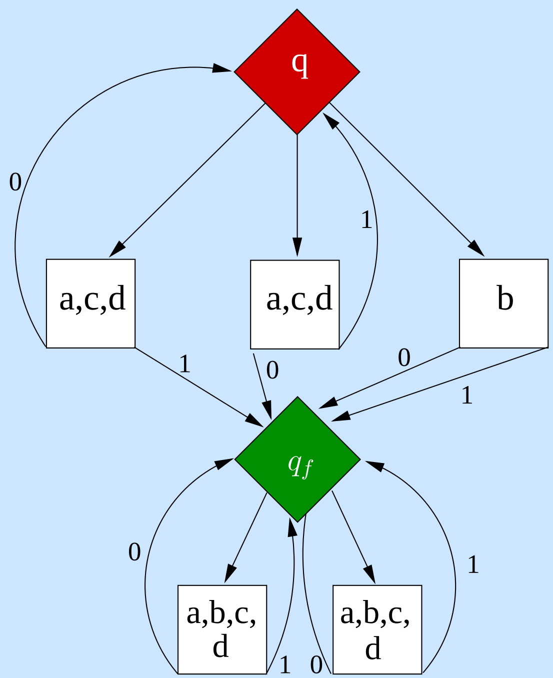
- **Given**:
  - Method $\mathcal{M}$: operates on input graph G_i to produce output graph $G_o = \mathcal{M}\,(G_i)$.
  - Property φ: some predicate on graphs.
- **Parameterized correctness**:

  For any arbitrarily large G_i, determine if: $\langle\varphi(G_i)\rangle\mathcal{M}\,\langle\varphi(G_o)\rangle$

Review: Tree automata

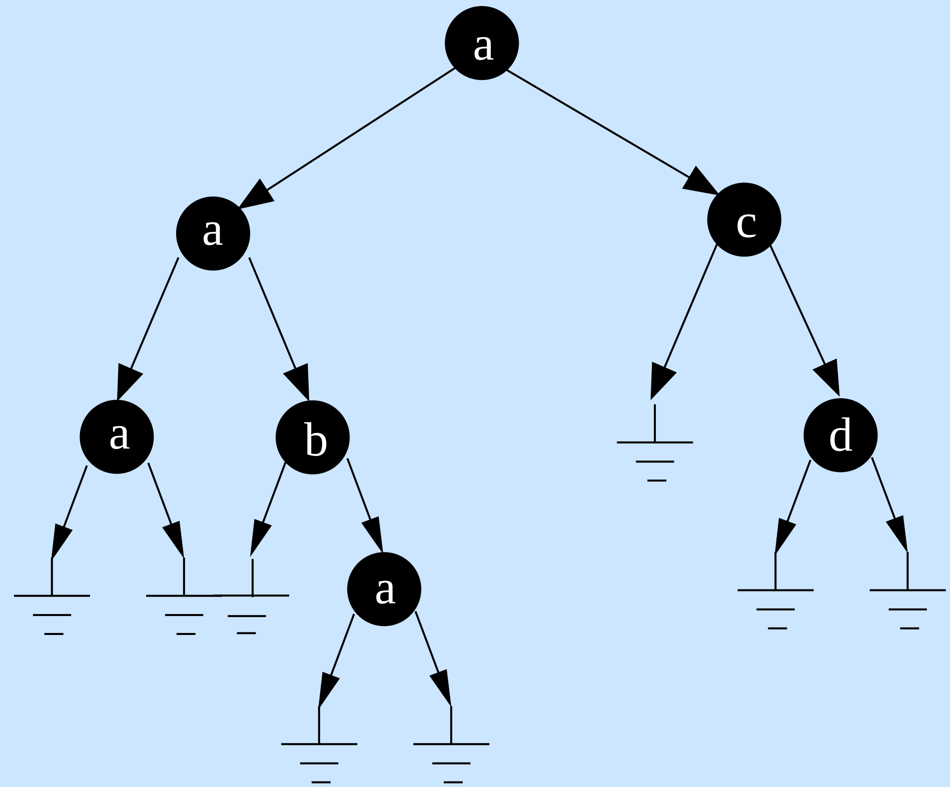
Example: Nondeterministic tree automaton for reachability (EF b)

$\Sigma = \{a, b, c, d\}$ $Q = \{q, q_f\}$ $q_0 = q$ $\Phi : \{c(q) = red(1), c(q_f) = green(2)\}$

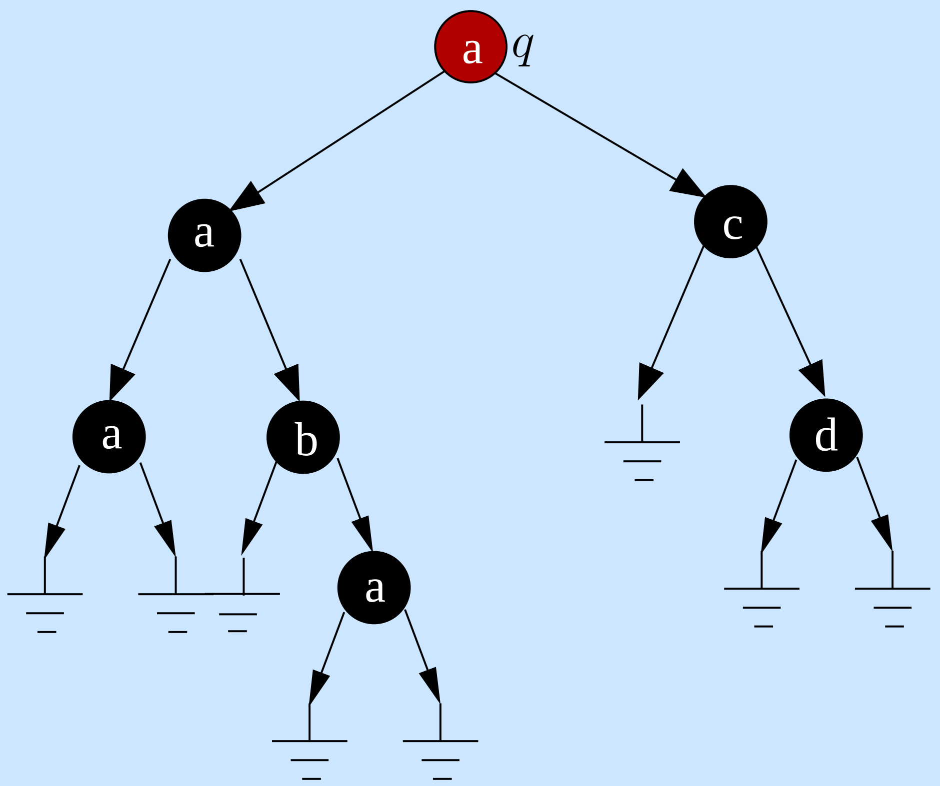


Transition diagram for δ

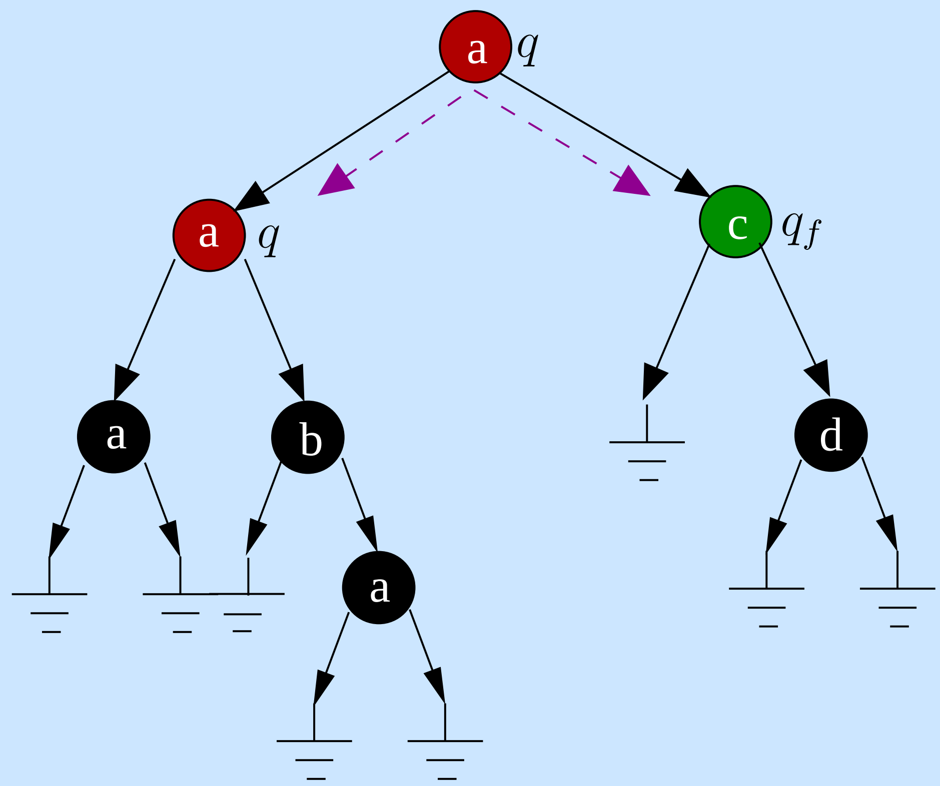
Example: accepting run of $\mathcal{A}_{reach}$ (EF b)



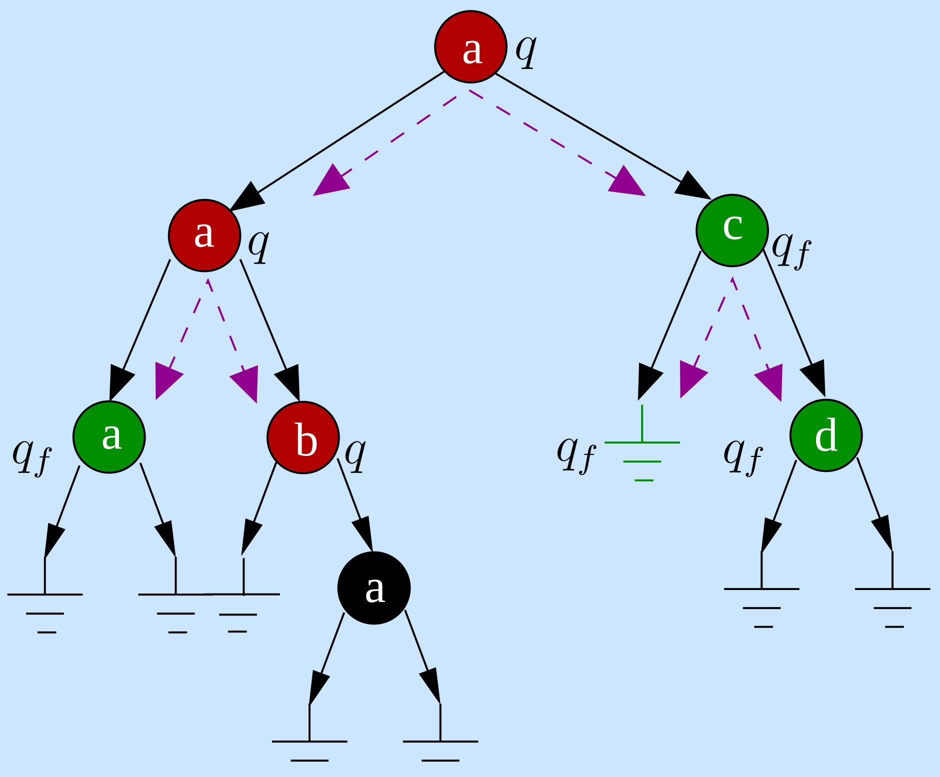
Example: accepting run of $\mathcal{A}_{reach}$ (EF b)



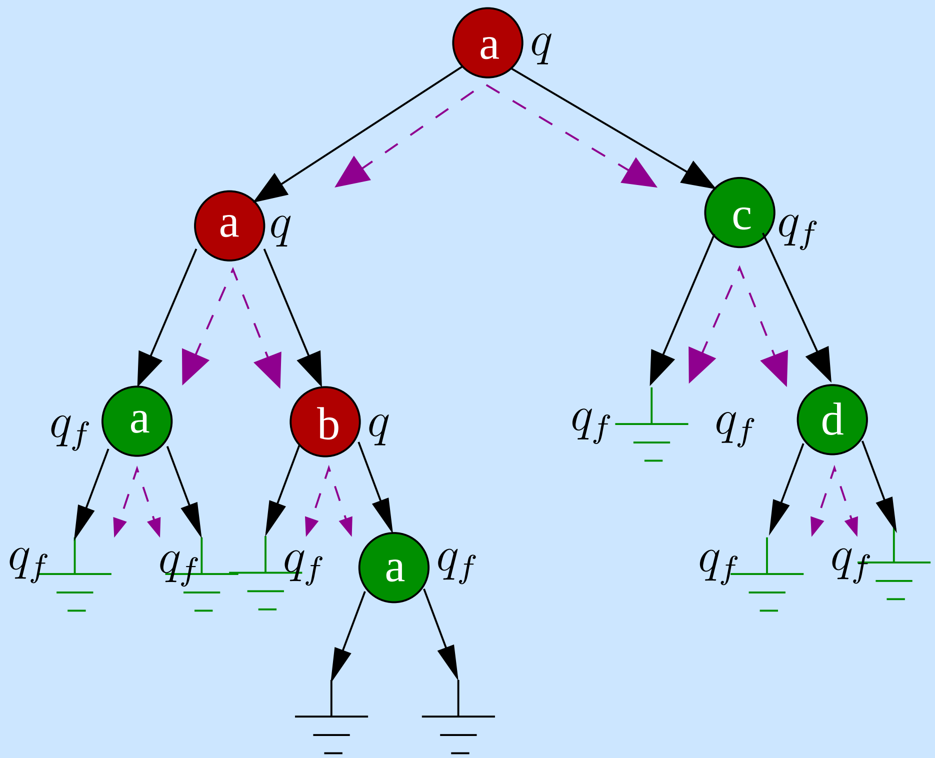
Example: accepting run of $\mathcal{A}_{reach}$ (EF b)



Example: accepting run of $\mathcal{A}_{reach}$ (EF b)



Example: accepting run of $\mathcal{A}_{reach}$ (EF b)



Definition: Destructive pass

- **Pass**: Traversal of graph visiting each node at most once.
- **Destructive update**: Modification of the input graph.

  e.g. Adding a node, Deleting a node, Changing a link, Changing a value, etc.
- **Destructive pass**: pass that performs at least one destructive update.

Stipulations

- Methods:
  - must terminate.
  - **should perform only a bounded number of destructive passes over the graph.**
  - should be iterative (no recursion).
- Domain of data values should be finite.
- Input graphs have varying, but bounded branching.

Example methods

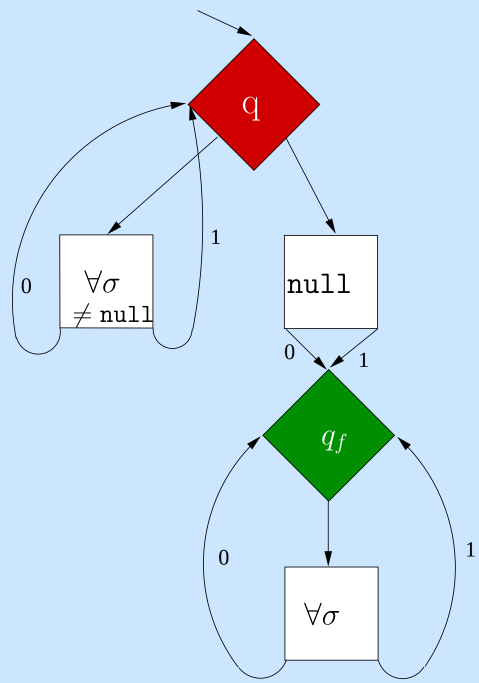
- Insertion/Deletion of nodes in linked lists (linear/circular),
- Insertion/Deletion of nodes in k-ary trees,
- Iterative modification of nodes in general graphs,
- Reversal of linked lists,
- Swapping nodes within a bounded distance.

Property specification

- Properties specified as non-deterministic tree automata.
- $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ called property automata.
- Examples include: Acyclicity, Sortedness, Reachability, Treeness, Listness, etc.

Example: checking acyclicity in a binary graph

$$\mathcal{A}_{cy} = \{\Sigma, \{q, q_f\}, q, \delta, \{c(q) = red(1), c(q_f) = green(2)\}\}$$



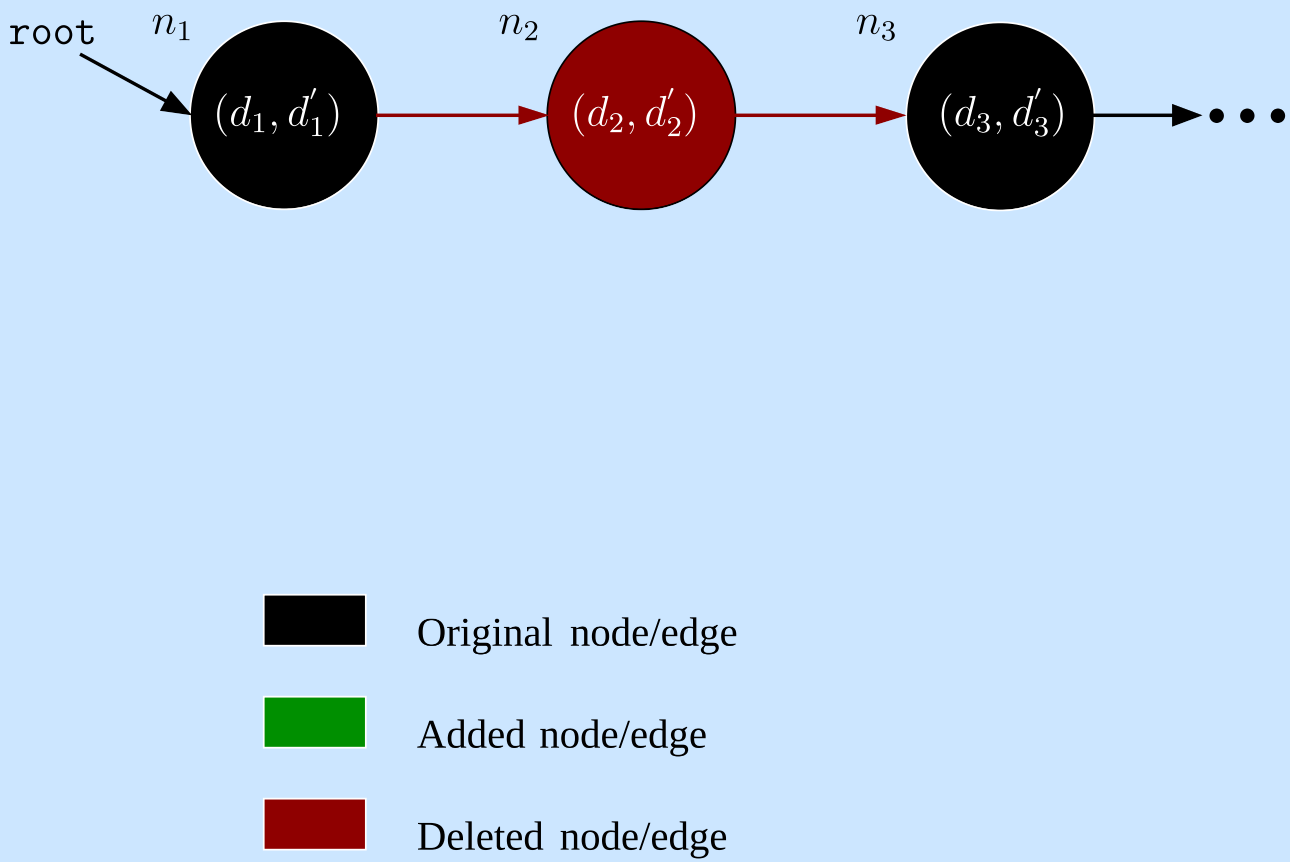
Transition diagram for $\mathcal{A}_{cy}$

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

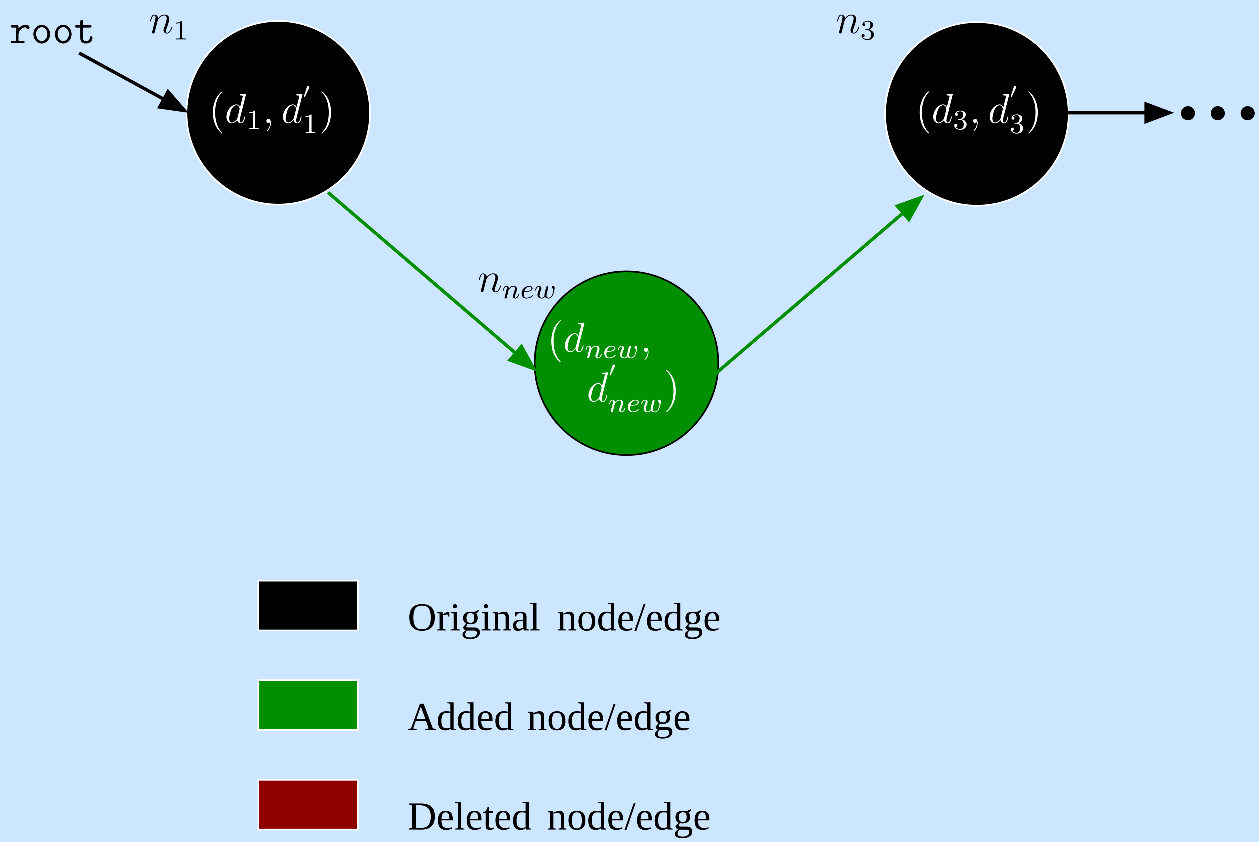
Modeling the method

- Method $\mathcal{M}$ modeled using tree automaton $\mathcal{A}_{\mathcal{M}}$.
- (G_i, G_o) represented as composite graph G_c.
- $\mathcal{A}_{\mathcal{M}}$ accepts all graphs G_c that represent valid I/O behavior of $\mathcal{M}$.

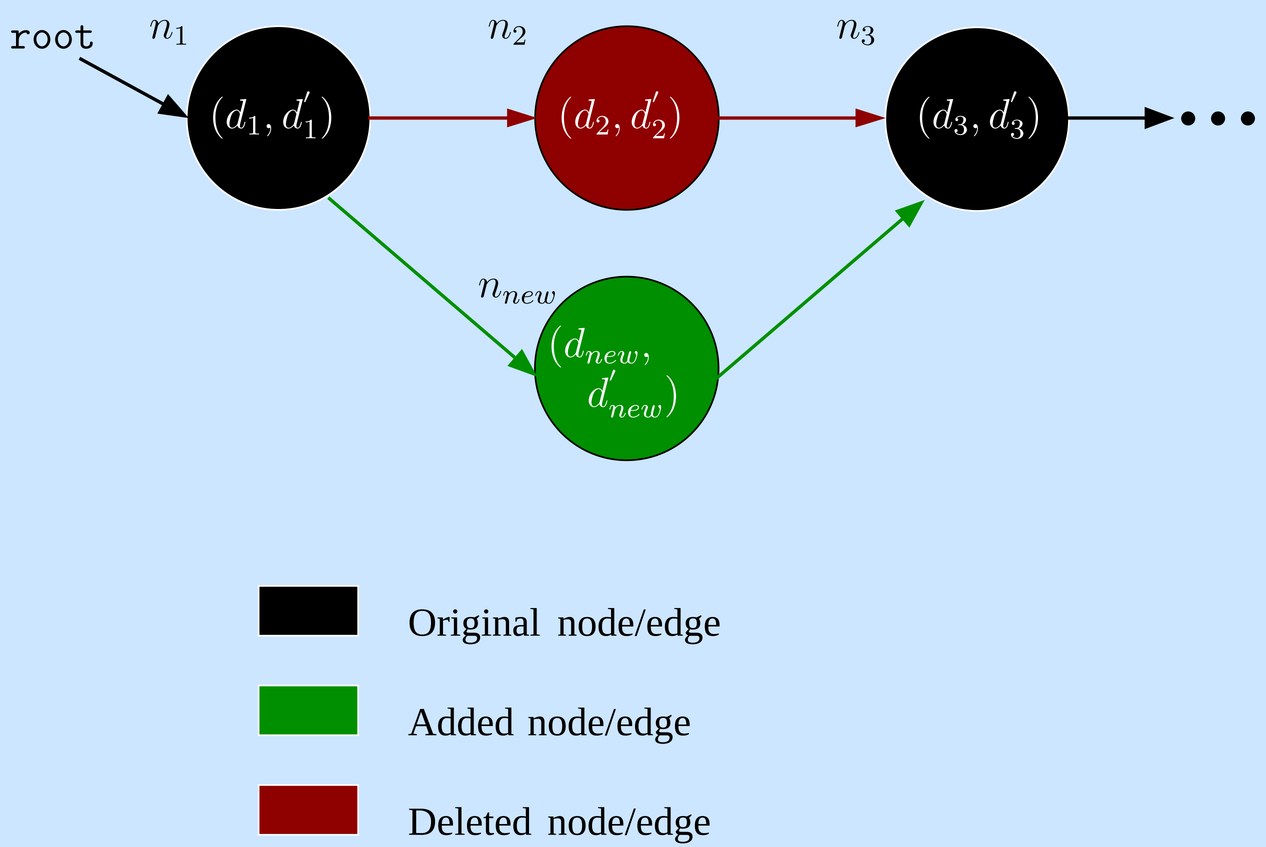
Input Graph: G_i



Output Graph: G_o



Composite Graph



Composite automaton

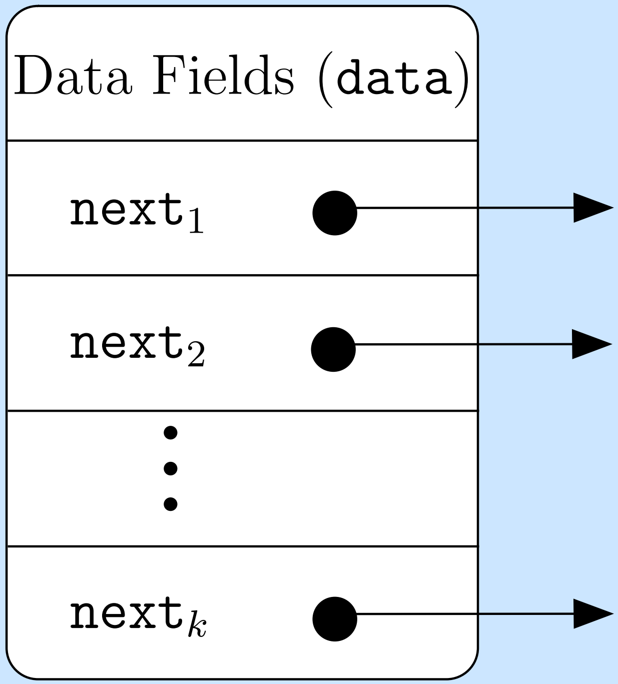
- **Given**: $\mathcal{A}_\varphi$, $\mathcal{A}_{\neg\varphi}$ and $\mathcal{A}_\mathcal{M}$.
- **Construct**: Composite automaton $\mathcal{A}_c$.
- $\mathcal{A}_c$: (synchronous) product of $\mathcal{A}_\varphi$, $\mathcal{A}_\mathcal{M}$ and $\mathcal{A}_{\neg\varphi}$.
- $\mathcal{A}_c$ accepts G_c, iff:
  - $\mathcal{A}_\mathcal{M}$ accepts G_c,
  - $\mathcal{A}_\varphi$ accepts G_i (input part), and
  - $\mathcal{A}_{\neg\varphi}$ accepts G_o (output part).

Reduction to language emptiness

- $\mathcal{A}_c$ accepts exactly those graphs that witness a *failure* of $\mathcal{M}$.
- $\mathcal{M}$ is correct iff language accepted by $\mathcal{A}_c$ is empty.
- $\mathcal{A}_c$ is empty implies parameterized correctness of $\mathcal{M}$.

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

Node of a data structure



Programming language

- Methods equipped with an iterator called "`cursor`".
- *Bounded window* (w): set of nodes within fixed distance from `cursor`.
- Auxiliary pointers: denote positions within w, relative to `cursor`.
- Types of statements: Assignment, Conditional and Loop statements.

Example method: Insertion in a singly linked list

```
method InsertNode (value, newValue){
    1:  cursor := head;
    2:  while (cursor != null) {
            [ncursor := cursor->next]
    3:      if (cursor->data == value) {
    4:          cursor->next := new node {
                        data := newValue;
                        next := ncursor;};
    5:          break; }
    6:      cursor := ncursor when true; } }
```

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

How does $\mathcal{A}_{\mathcal{M}}$ emulate $\mathcal{M}$?

While operating on composite graph $G_c = (G_i, G_o)$, $\mathcal{A}_{\mathcal{M}}$:

- reads a new node $n = (n_i, n_o)$,
- changes state to mimic atomic updates to n_i,
- checks if updated node matches n_o, and
- if yes, moves to next node.

From $\mathcal{M}$ to $\mathcal{A}_{\mathcal{M}}$: I

- $\mathcal{A}_{\mathcal{M}}$ starts in state q_0 and reads node (n_i, n_o).
- State of $\mathcal{A}_{\mathcal{M}}$ encodes updated value of n_i.
- Statements that do not alter `cursor` position map to ε-moves.

  e.g. conditionals, loop body, assignments (except to `cursor`)

From $\mathcal{M}$ to $\mathcal{A}_{\mathcal{M}}$: $\mathtt{II}$

- For assignments that alter `cursor` position:
  - check if current state matches n_o,
  - if yes, read new node,
  - if no, transition to *reject* state.
- Transition to accept state after last statement in $\mathcal{M}$.
- Add self-loops to reject and accept states.

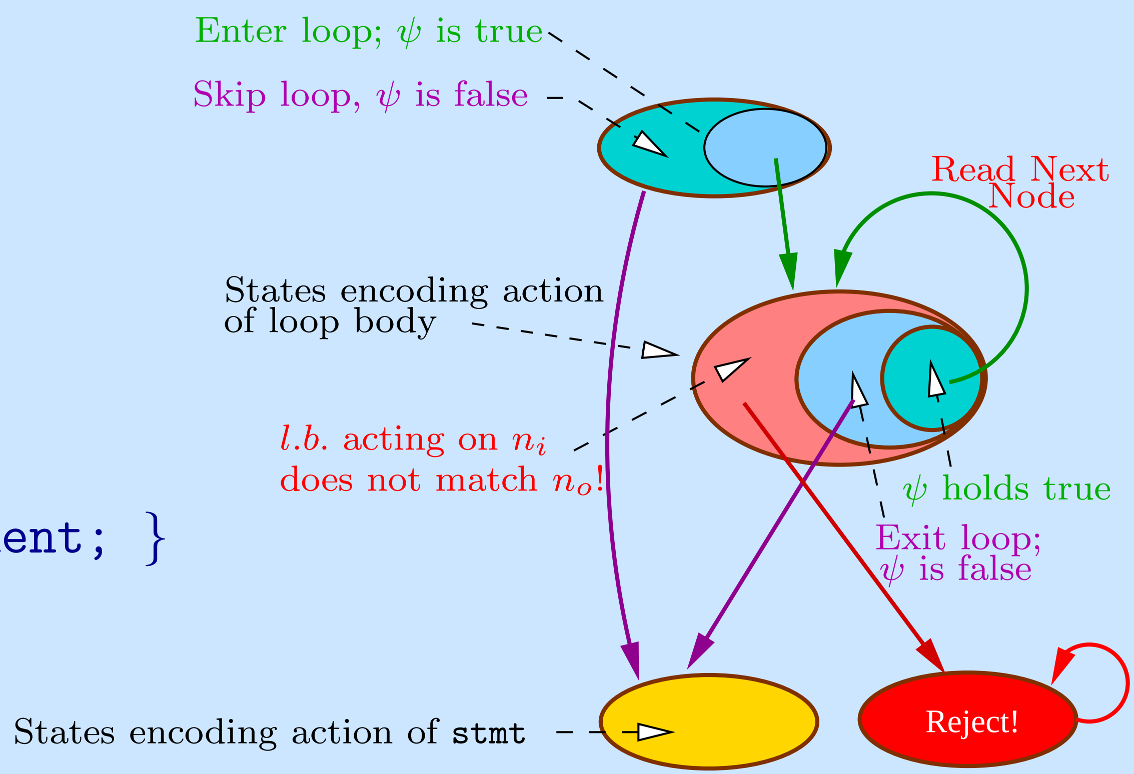
Example: Compilation of a `while` statement

while (ψ) {

loop body;
($l.b.$)

update statement; }

stmt;

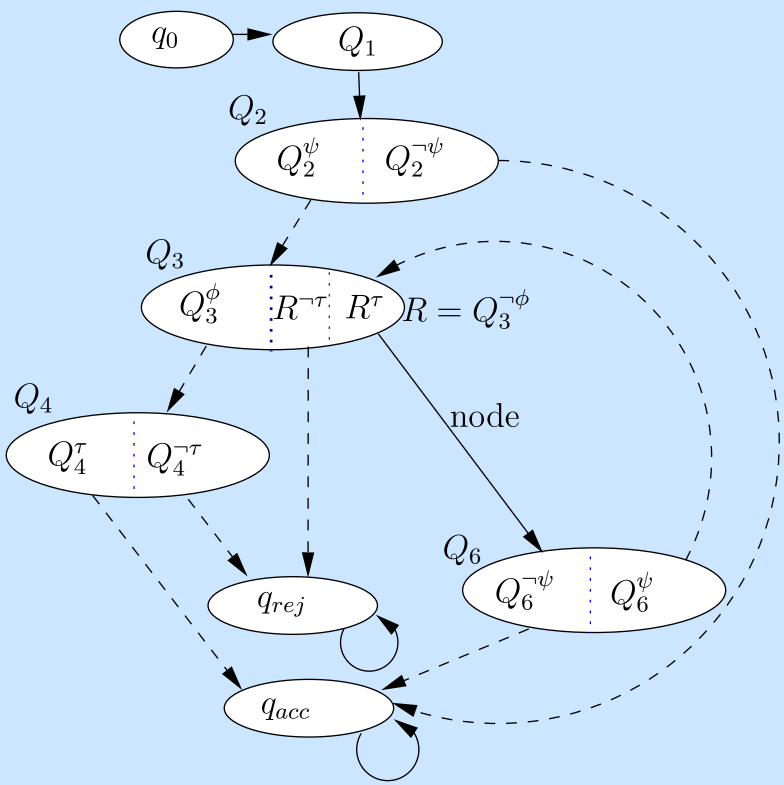


- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
  - Examples
- Efficiency
- Related work and conclusions

Example method: Insertion in a singly linked list

```
method InsertNode (value, newValue){
     1:  cursor := head;
     2:  while (cursor != null) {
              [ncursor := cursor->next]
     3:       if (cursor->data == value) {
     4:            cursor->next := new node {
                        data := newValue;
                        next := ncursor;};
     5:            break; }
     6:       cursor := ncursor when true; } }
```

Example method automaton for `insertNode`

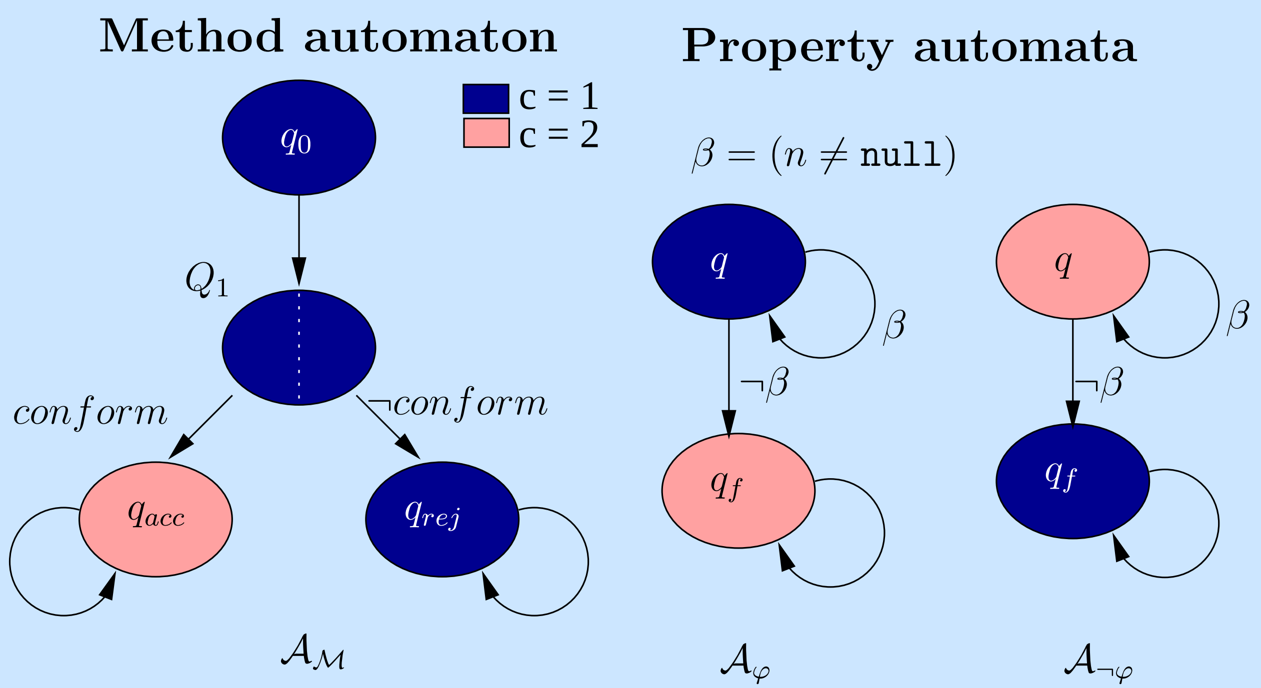


Example: An incorrect method

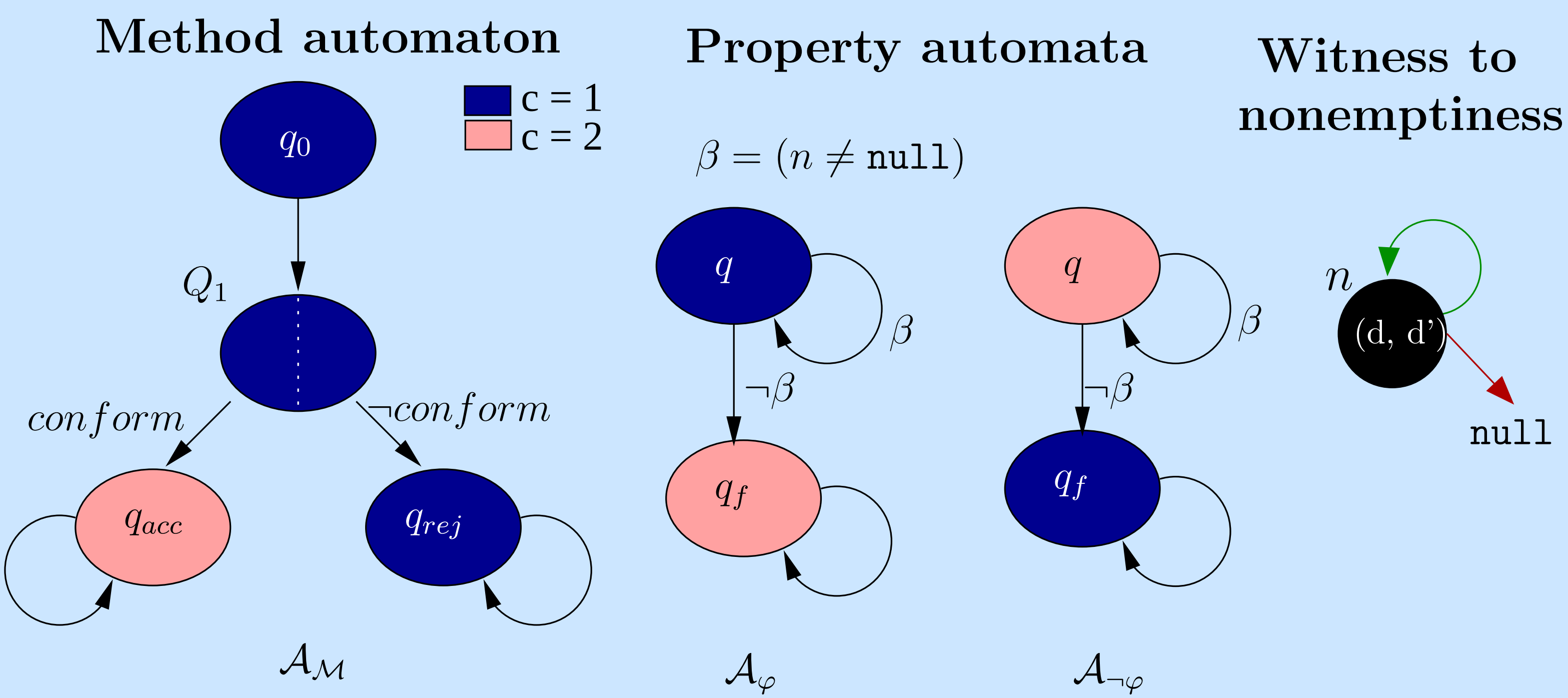
```
method sampleMethod {
    cursor->next := cursor;
    cursor->data := 10; }
```

Does this method preserve acyclicity?

Constructing the composite automaton



Composite automaton is non-empty!



- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

Efficiency

- $\mathcal{A}_{\mathcal{C}}$: linear in $|\mathcal{A}_{\mathcal{M}}|$, $|\mathcal{A}_{\varphi}|$ and $|\mathcal{A}_{\neg\varphi}|$.
  - Size of $\mathcal{A}_{\mathcal{M}}$: $O(|\mathcal{M}|)$.
  - $\mathcal{A}_{\mathcal{M}}$, $\mathcal{A}_{\varphi}$, $\mathcal{A}_{\neg\varphi}$ have small, fixed number of colors in parity condition.
- Non-emptiness: polynomial in $|\mathcal{A}_{\mathcal{C}}|$.
- Overall complexity: **polynomial** in size of $\mathcal{M}$ and property automata.

- Motivation
- Preliminaries
- Solution strategy
- Programming language
- Compiling into automata
- Efficiency
- Related work and conclusions

Related work: I

- **Pointer Assertion Logic Engine**: [Møller, Schwartzbach, 2001]
  - More general (uses *MSOL*), but complexity is non-elementary.
  - Requires human ingenuity in providing loop invariants.
- **Separation logic**: [O'Hearn, Reynolds, Yang, 2001]
  - Deductive system with proof rules.
  - Decidable fragment treats only linked lists.

Related Work: II

- **Shape analysis**: [Sagiv, Reps, Wilhelm, 1999]
  - *Shape* invariants represented using 3-valued logic.
  - Broad scope, but inexact solutions.
- **Transducer-based approach**:[Bouajjani *et al*, 2005]
  - Abstraction refinement based approach.
  - Limited to single successor data structures.

Conclusions

- Efficient algorithmic technique for verification of parameterized data structures.
- Reasoning about a large class of methods, examples include:

  Adding, deleting, inserting nodes in linked lists, binary search trees, swapping nodes within a bounded distance, reversing lists, etc.
- Properties such as: acyclicity, reachability, sortedness, treeness, listness, sharing etc.
- Complexity: *polynomial* in size of method and property specifications.

Thank You!

Tree automata

A (parity) tree automaton $\mathcal{A}$ has the form: $(\Sigma, Q, \delta, q_0, \Phi)$, where:

- Σ is the input alphabet (nodes of the graph),
- Q is the finite non-empty set of states,
- $\delta : Q \times \Sigma \rightarrow 2^{Q^k}$ is the non-deterministic transition relation,
- q_0 is the initial state, and
- Φ is the parity acceptance condition.

Run of a tree automaton $\mathcal{A}$

- *Run*: Annotation of input tree with states of $\mathcal{A}$.
- *Accepting run*: Run in which acceptance condition is true for all paths.
- $\mathcal{A}$ accepts tree T if there is some accepting run on T.
- Notion of run can be generalized to general graphs.

Parity acceptance condition

- States colored with colors $\{c_0, \ldots, c_m\}$.
- π is some finite/infinite sequence of states.
- π satisfies parity condition iff: maximal index of color appearing infinitely often is **even**.
- Remark: Our technique needs 2 colors in most cases.

Programming language: Syntax

- Assignment statement syntax:
  - `cursor->data :=` d`;` (Modify data value)
  - `cursor->next :=` ptr`;` (Redirect an edge)
  - `cursor :=` ptr`;` (Change cursor location)
  - `cursor := new node`$\{$`data:=`d`;next`$_1$`:=null;`$\ldots\}$`;` (Add new node)
  - `cursor->next := new node` $\{\ \ldots\ \}$`;` (Add new node after cursor)
- Conditional statements:
  - standard if-then-else construct
  - test condition: data comparison, pointer comparison (within the window)

Loop statements

```
while (ψ) {
    loop body;
    update statement; }
```

- Used for iterating through the data structure.
- Nesting of loops not permitted.
- `cursor` cannot be changed inside loop body.
- Update statement used to change `cursor` position.