Symbolic Verification of Programs with Pointers using Tree Automata

Jiří Šimáček

Université Joseph Fourrier (France)
Brno University of Technology (Czech Republic)

Ph.D.

- 1st year of doctoral degree programme at Brno University of Technology
  - supervised by Tomáš Vojnar
- joint supervision under the cotutelle agreement with Université Joseph Fourrier
  - supervised by Yassine Lakhnech
  - co-supervised by Radu Iosif
- the topic of the research:
  Advanced Symbolic Verification Methods Using Finite-State Automata and Related Formalisms

General Program Structure

- a computer program can combine various constructions such as:
  - arithmetic,
  - array manipulation,
  - pointer manipulation,
  - recursion,
  - parallel execution, etc.
- verification of each of the above requires different approaches (which can be combined in the ideal case)
- we focus on programs with pointers
  - bugs in pointer manipulation can be very tricky when using low level programming languages (C/C++)
  - yet the pointers allow construction of useful data structures (list, trees, etc.)

Programs with Pointers

- we restrict to the following statements (`x`, `y` are pointer variables, `next(i)` denotes i-th selector):
  - `new(x)` (heap allocation)
  - `x := null` (nil assignement)
  - `x := y` (simple assignement)
  - `x := y.next(i)` (assignement with dereference of source)
  - `x.next(i) := y` (assignement with dereference of destination)
  - `if/while (x = y)` (conditional branching)
  - `delete(x)` (heap deallocation – optional)
- no C-style pointer arithmetic (`p++`, `*(p+3)`)

Programs with Pointers – Verification

- safety
  - a pointer variable has to point to some memory cell when dereferenced, i.e. it has to be assigned a valid address before
  - a memory cell released by calling `delete` is never used in the future (and also never released again)
  - user specified assertions
- termination (liveness)
  - a program terminates for any input

Related Work

- 3-valued predicate logic with transitive closure
  - [Sagiv, Reps, Wilhelm '96]
- separation logic
  - [Reynolds '02]
- regular model checking
  - [Kesten, Maler, Marcus, Pnueli, Shahar '97]
- many other approaches exist

3-valued Predicate Logic with Transitive Closure

- at a given program point, a single pointer variable can point to a (possibly infinite) set of structures (in all possible executions of a program)
- the aim of the analysis is to create a finite representation of the heap
- it does so by using *shape graphs*, which consist of an *abstract state*, an *abstract heap*, and a *sharing information* for *abstract locations*

Separation Logic

- the heap often consists of indipendent parts which are not interconnected or which are interconnected in a bounded way
- separation logic extends Hoare logic in order to reason about different parts of the heap locally
  - heap configurations are represented by formulae in separation logic (data structures are described using recursive predicates)
  - an execution of the program statements is replaced by a Hoare-style reasoning and a generating of invariants

Seperation Logic – Example

- list segment predicate:
  $ls(E, F) \iff E \neq F \wedge (E \mapsto F \vee (\exists x'.E \mapsto x' * ls(x', F)))$
- list reversal (u points to a singly-linked list at the beginning):

```
while (u ≠ null) do        {ls(u, ⊥)}
      w := u.next;
      u.next := v;
      v := u; u := w;
od                         {ls(u, ⊥) * ls(v, ⊥)} (inv.)
                           {ls(v, ⊥)}
```

- things to verify:
  - no null pointer dereference occurs,
  - the program eventually terminates,
  - v contains the reversal of u at the end

Regular Model Checking

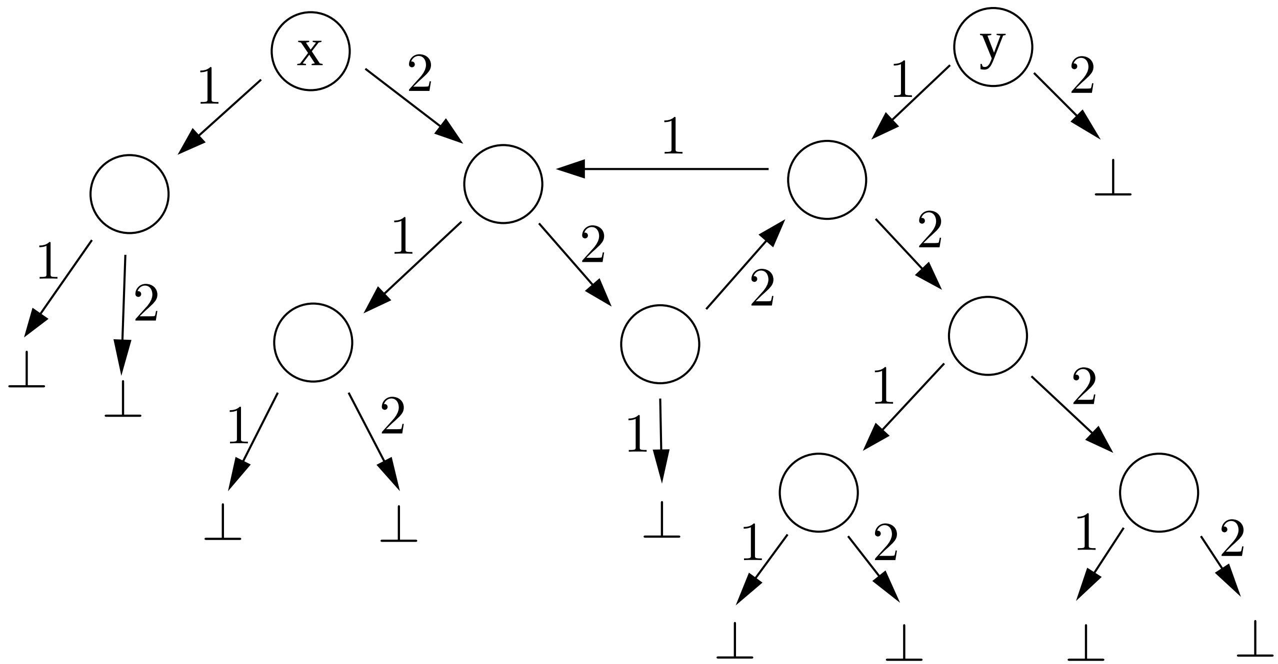
- heap configurations are represented by finite automata (over words or trees)
- program statements are interpreted over these automata (usually using transducers)
- it is possible to use CEGAR approach
- some modifications (ARTMC) allow verification of more complex structures than trees by using tree automata only
  - [Bouajjani, Habermehl, Rogalewicz, Vojnar '06]
- it is possible to verify:
  - operations on doubly linked lists,
  - operations on different kind of trees,
  - Deutsch-Schorr-Waite algorithm, etc.

A New Method of Verification based on Tree Automata

- why?
  - separation logic: often requires the specification of recursive predicates (e.g. for a singly-linked list) and invariant generation rules over these predicates; only a limited ability to handle something more complex than lists
  - regular model checking: the invariant generation is automated, but the heap is represented by a single automaton; doesn't scale well on very complex structures
- we want to combine advantages of both methods
- we want to handle more general structures than lists or trees
- we want to avoid using transducers for symbolic execution of statements (overhead)

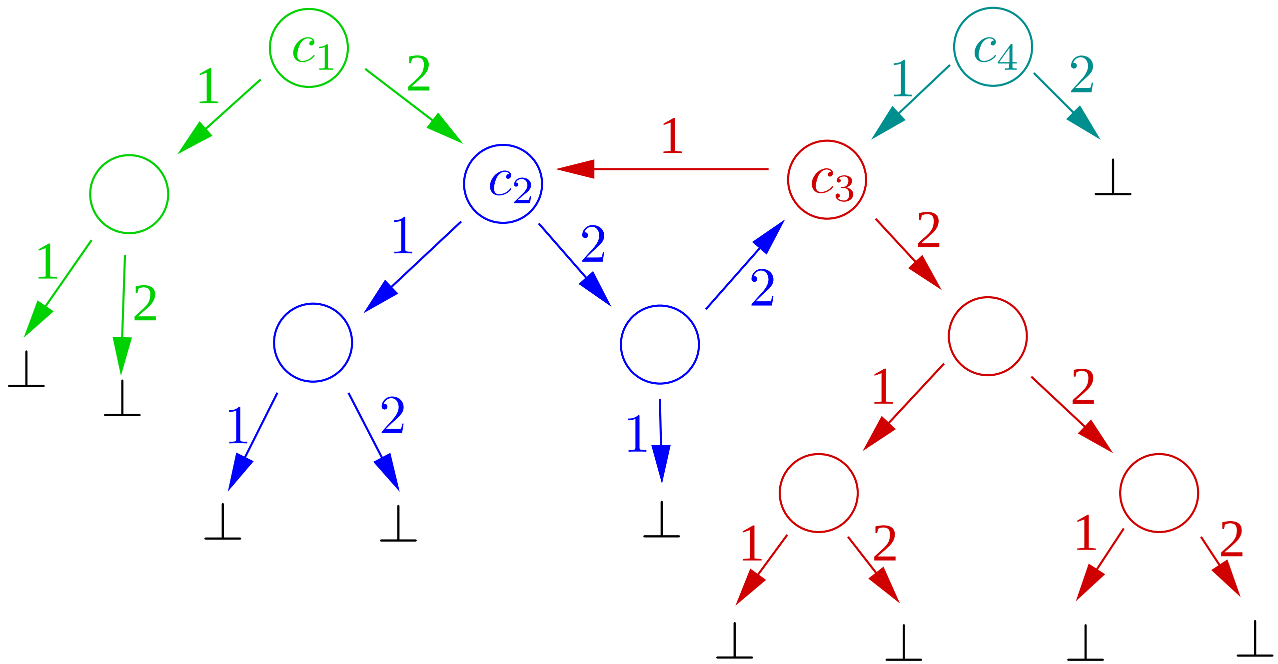
Heap Representation

- the heap can be viewed as a directed graph, where nodes represent memory cells and edges represent the selectors
- an example (⊥ denotes `null` value, `x`, `y` are pointer variables, memory cells contain selectors 1, 2)



Tree-based Heap Decomposition and Cut-points

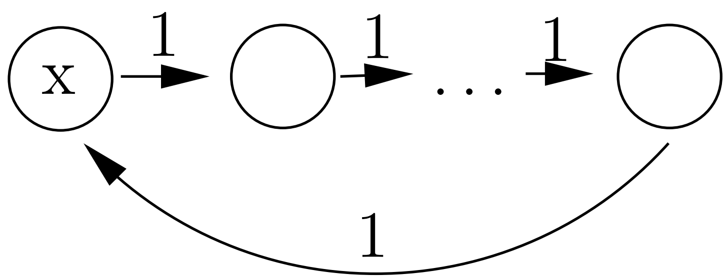
- the heap is a general directed graph, but we have tree automata only
  - graph automata exist, but operations are too hard
- the heap can be decomposed into trees by using *cut-points*, which are nodes pointed to by a variable or nodes that contain more than one incoming edge (are pointed to by more than one selector)
- example (x, y point to c_1 and c_2 respectively):



Representing Memory Configurations by Tree Automata

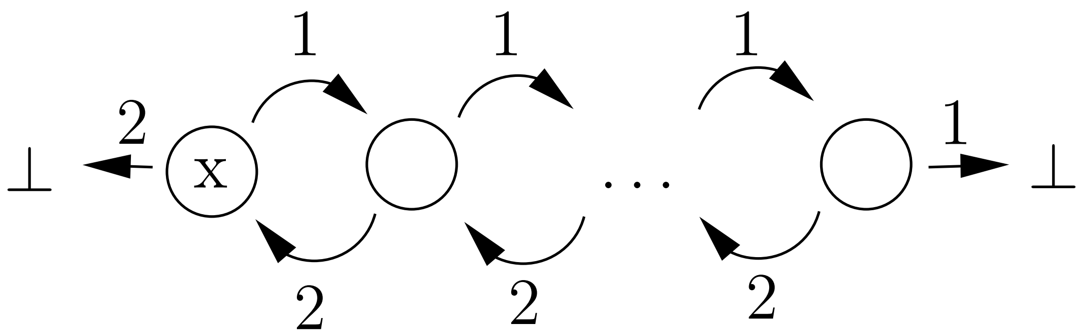
- an accepting run (bottom-up) of the automaton describes a part of one heap configuration (memory cells and content of their selectors); the complete configuration is obtained by combining runs of several such automata
- each cut-point can appear at most once (as an accepting state) in a run (it represents only a single memory cell)
- the automaton contains leaf rules for $\perp$ and for each cut-point
- an example (a singly-linked list):

$$\begin{array}{rcl} 1(q_1) & \rightarrow & c_1' \\ 1(q_1) & \rightarrow & q_1 \\ 1(c_1) & \rightarrow & q_1 \\ (\text{leaf rule: } a & \rightarrow & c_1) \end{array}$$

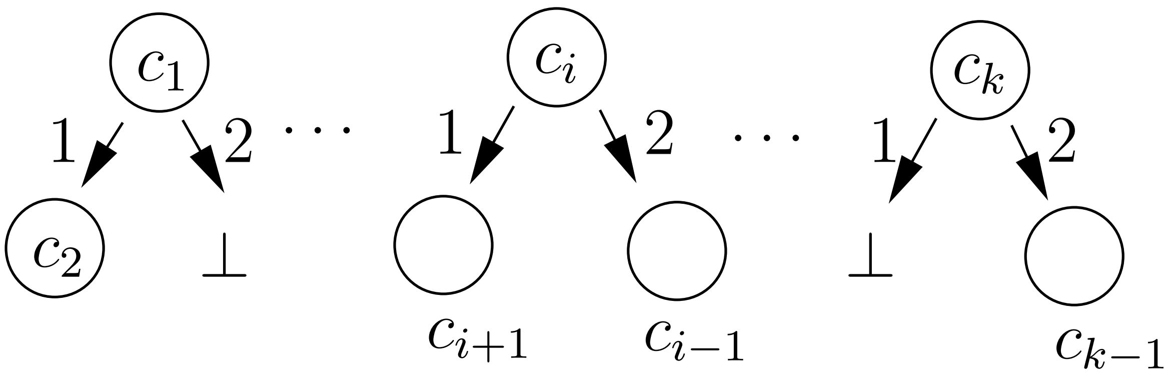


Introducing hierarchy

- what about a doubly-linked list?



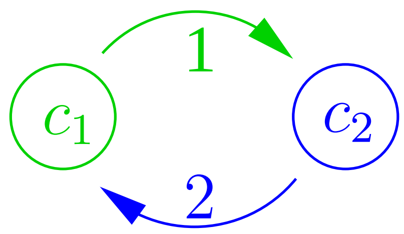
- we get an unbounded number of cut-points in the tree decomposition!



Introducing Hierarchy

- try to hide some of the cut-points in the hierarchically structured automata
- in the case of doubly-linked lists, create a box consisting of 2 automata – $DLL(out : c_1, in : c_2)$:

$$A_1\colon\ 1(c_2) \quad \rightarrow \quad c_1'$$
$$A_2\colon\ 2(c_1) \quad \rightarrow \quad c_2'$$

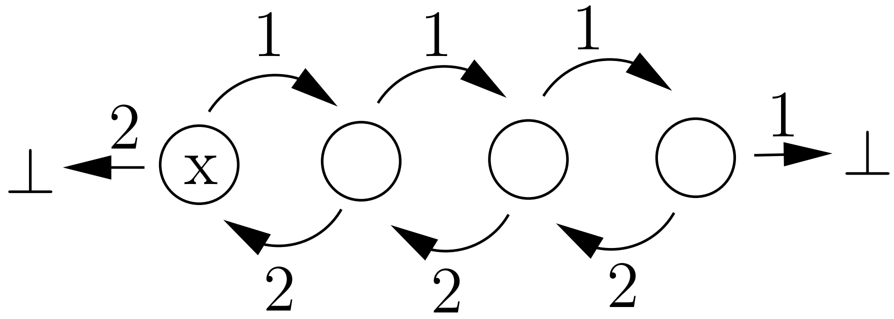


- use this box as a symbol on a higher level:

$$\begin{aligned} \langle DLL, 2\rangle(q_1, \perp) &\rightarrow c_1' \\ DLL(q_1) &\rightarrow q_1 \\ 1(\perp) &\rightarrow q_1 \end{aligned}$$

Introducing Hierarchy – Example

- consider the doubly-linked list:



- the run of the corresponding automaton looks as follows (without leaf rules):

$$
\begin{array}{ccccccccc}
\perp & \xrightarrow{1} & q_1 & \xrightarrow{DLL} & q_1 & \xrightarrow{DLL} & q_1 & \xrightarrow{DLL} & c'_1 \\
 & & & & & & \perp & \xrightarrow{2} &
\end{array}
$$

Main Challenges

- language inclusion ($\subseteq$)
  - we don't know how to complement hierarchical tree automata but we know how to test inclusion on tree automata without complementing [Bouajjani, Habermehl, Holik, Touili, Vojnar '08]
  - we don't know how to do the inclusion in general (yet)
  - there are some safe approximations though (top-level inclusion checking)
- the other automata operations ($\cup$, $\cap$)
- invariant generation

Low Level Symbolic Representation

- automata tend to grow too much to fit in a memory
- there are ways how to store them efficiently using symbolic representation
  - BDDs,
  - sparse matrices, ...
- already used in ARTMC (MONA)
- current implementations usually targets deterministic automata only

Future Directions

- an ability to handle dynamic structures containing data
- an automated learning of the hierarchy
- function calls
  - heap summaries
  - the recursion
- multi-threaded programs
  - an ability to lock each node separately
- a tool that scales

Thank You