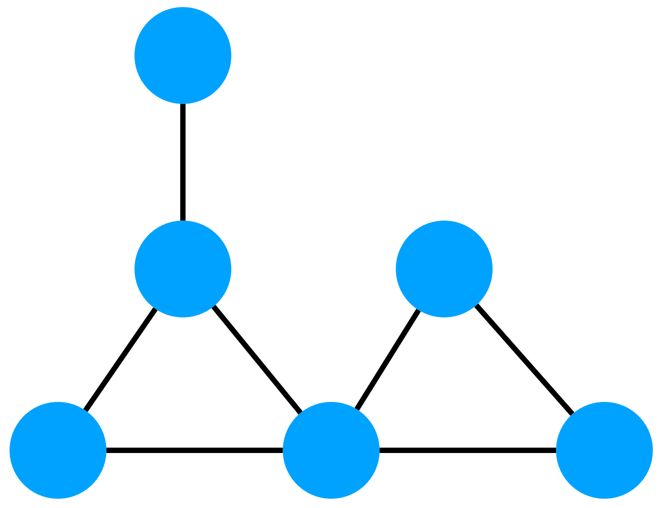
Using Tree-width for the verification of infinite state systems

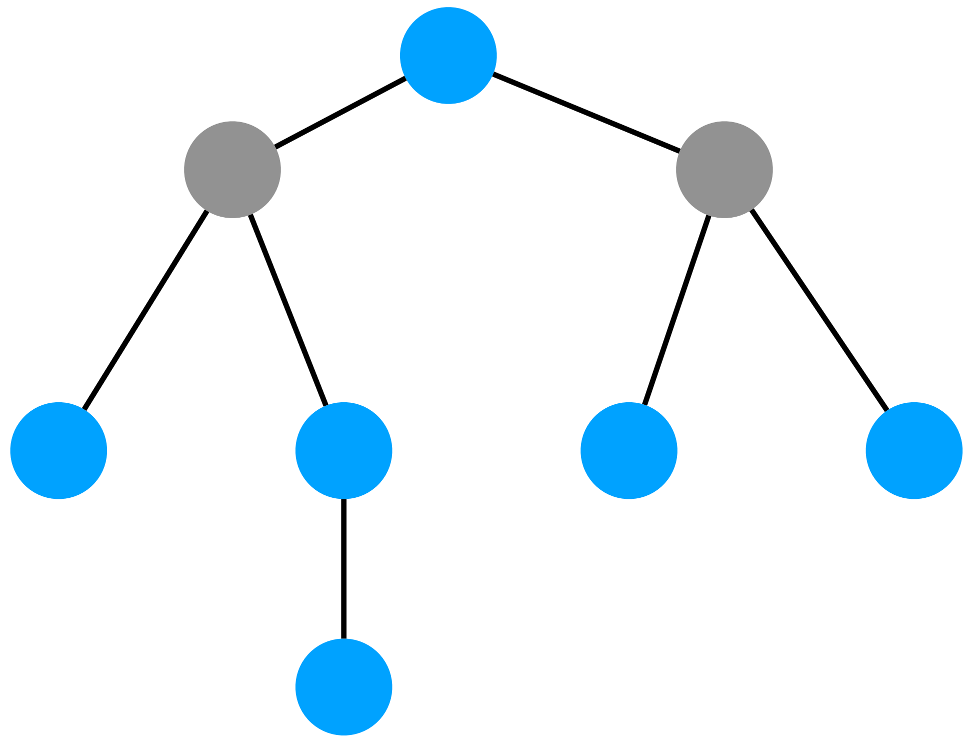
C. Aiswarya - CMI, UMI ReLaX

CAALM - 2019

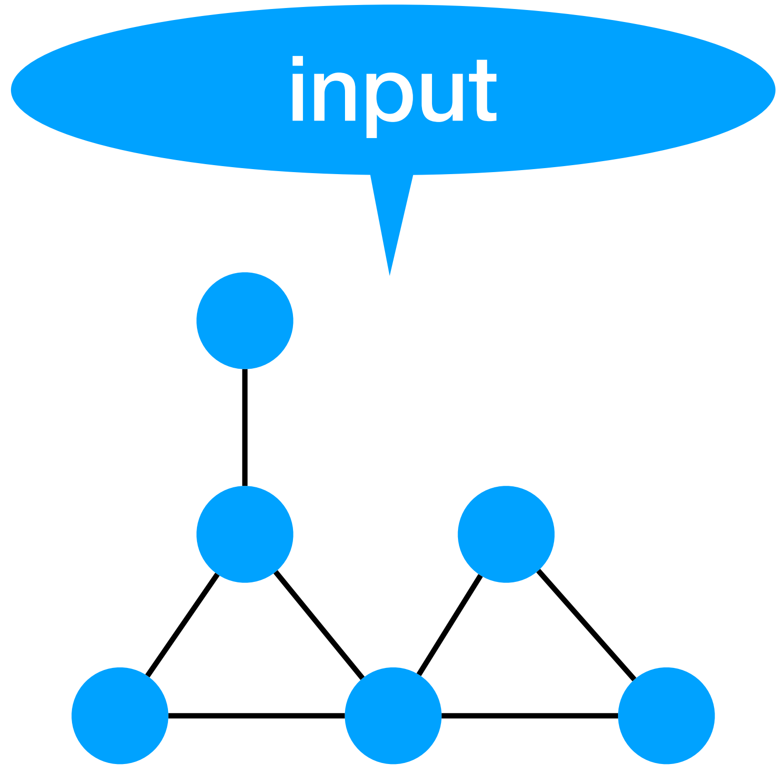
Tree-interpretation

Interpret a graph in a tree





Model checking



fixed

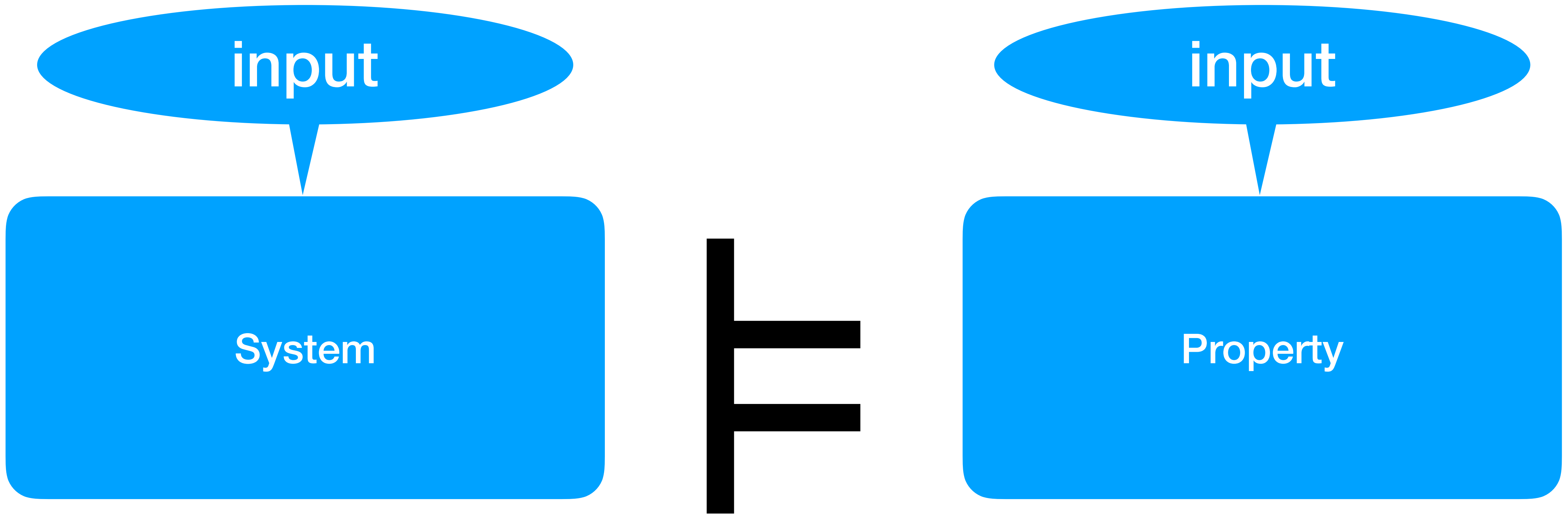
$$\models \quad \forall x\, \exists y\, \exists z\;\; E(x,y) \wedge E(x,z) \wedge x \neq y$$

Can be done efficiently if the tree-width is bounded.

Fixed-parameter tractable.

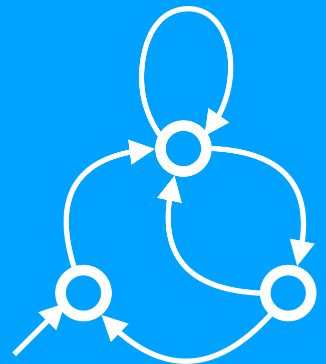
Model checking

Model checking



Model checking

input

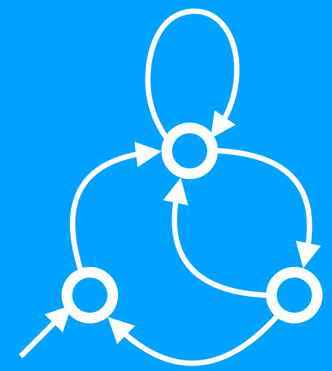


input

$\forall x\, \exists y\, \exists z\;\; E(x,y) \wedge E(x,z) \wedge x \neq y$

Model checking

input



input

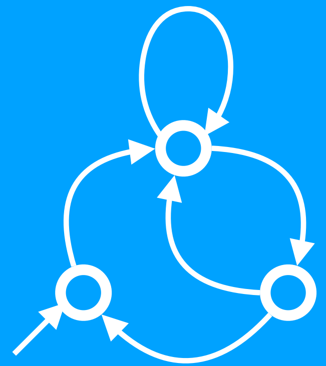
$\forall x\, \exists y\, \exists z\; E(x,y) \wedge E(x,z) \wedge x \neq y$

Every run of the system satisfies the property.

Model checking

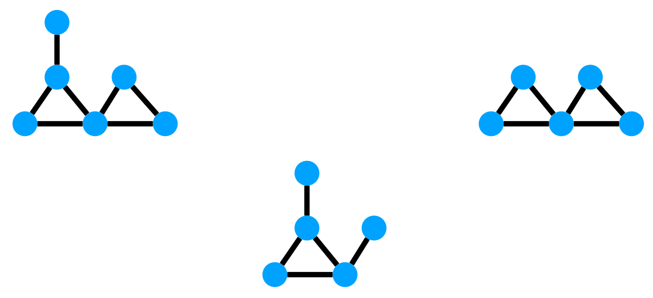
input

input



$$\models$$

$$\forall x\,\exists y\,\exists z\; E(x,y) \wedge E(x,z) \wedge x \neq y$$

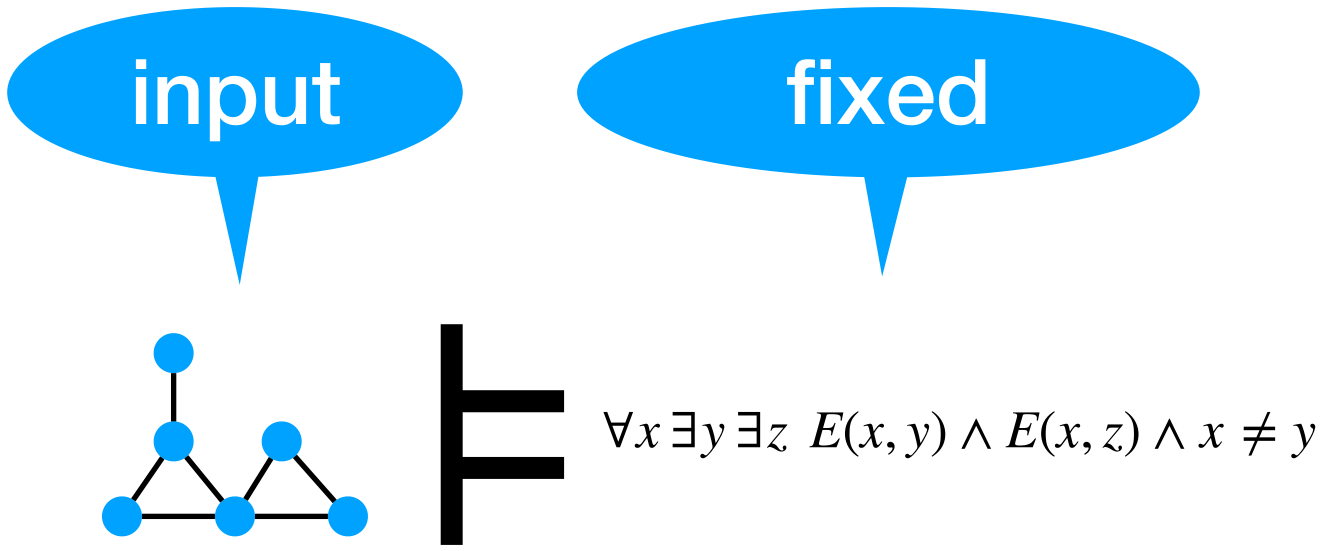


...

$$\models$$

$$\forall x\,\exists y\,\exists z\; E(x,y) \wedge E(x,z) \wedge x \neq y$$

Model checking

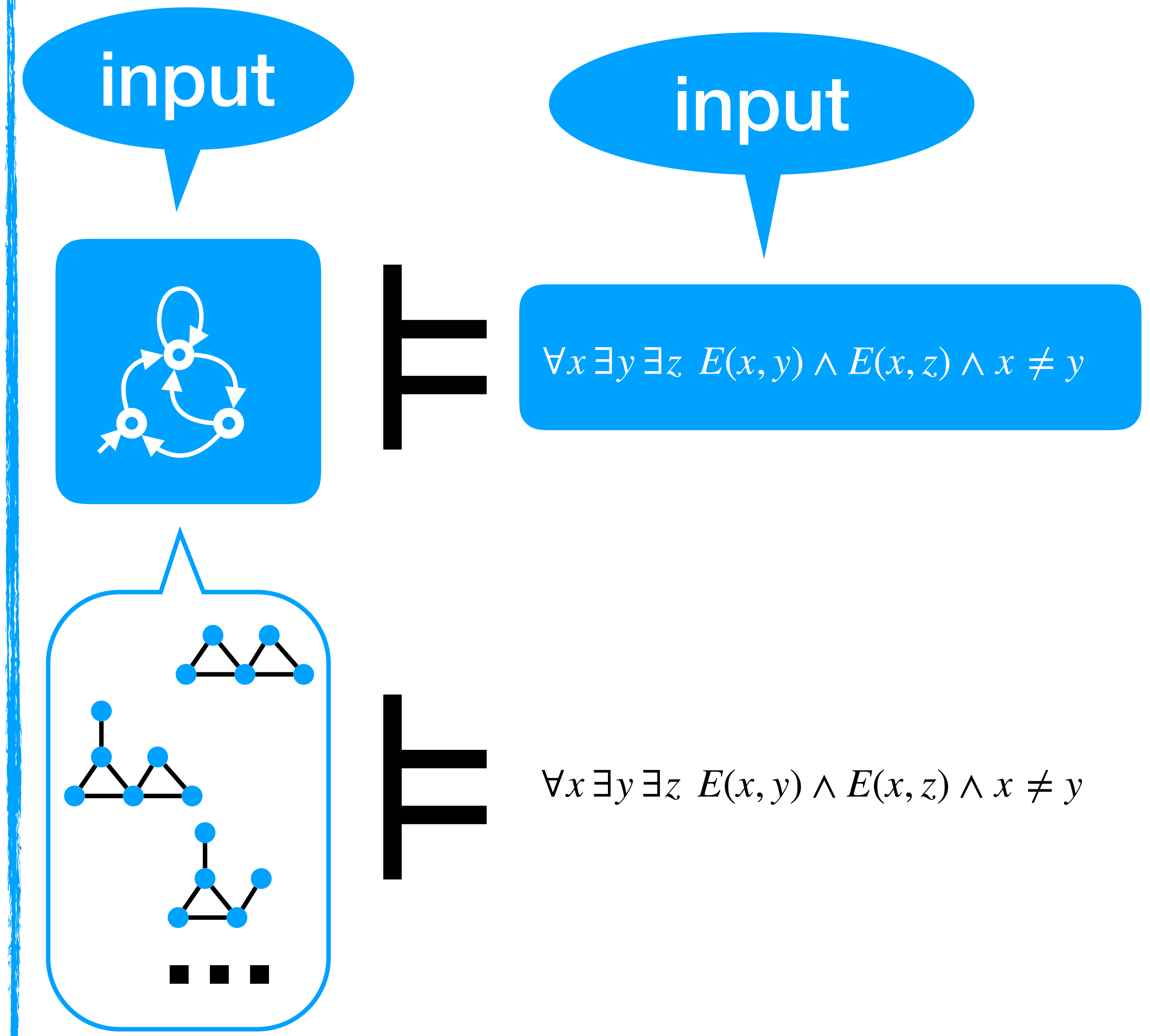


$$\models \quad \forall x\, \exists y\, \exists z\ \ E(x,y) \wedge E(x,z) \wedge x \neq y$$

Can be done efficiently if the tree-width is bounded.

Fixed-parameter tractable.

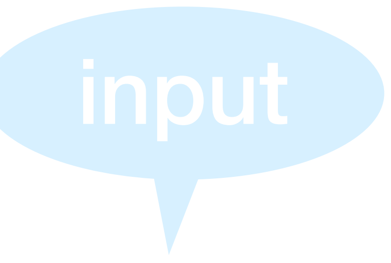
Model checking



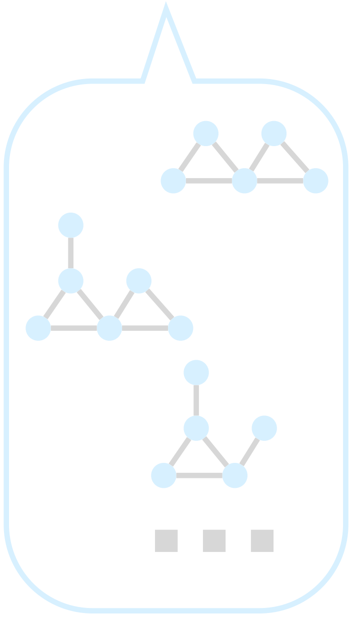
$$\models \quad \forall x\, \exists y\, \exists z\ \ E(x,y) \wedge E(x,z) \wedge x \neq y$$

$$\models \quad \forall x\, \exists y\, \exists z\ \ E(x,y) \wedge E(x,z) \wedge x \neq y$$

Verification



Understanding behaviours as graphs…

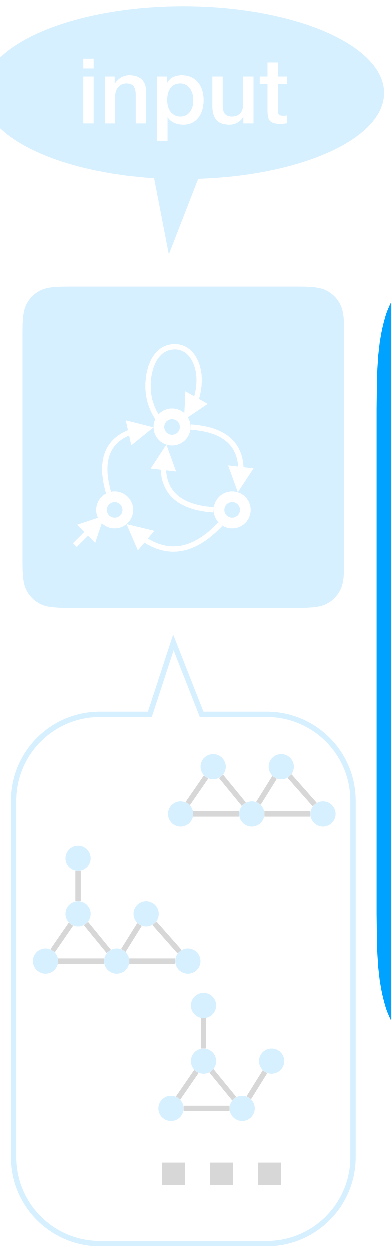


Pushdown systems

Multi-pushdown systems

Message passing systems

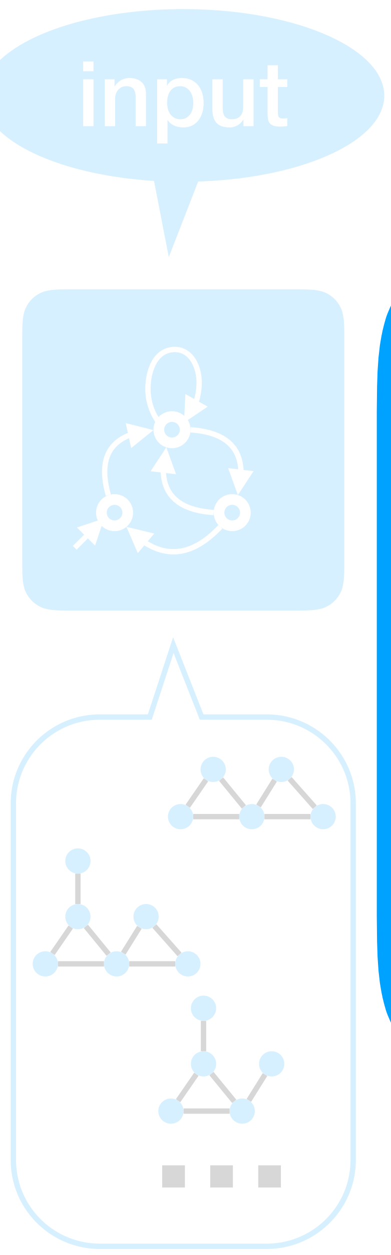
Message passing pushdowns



Understanding behaviours as graphs...

Pushdown systems
(recursive programs)

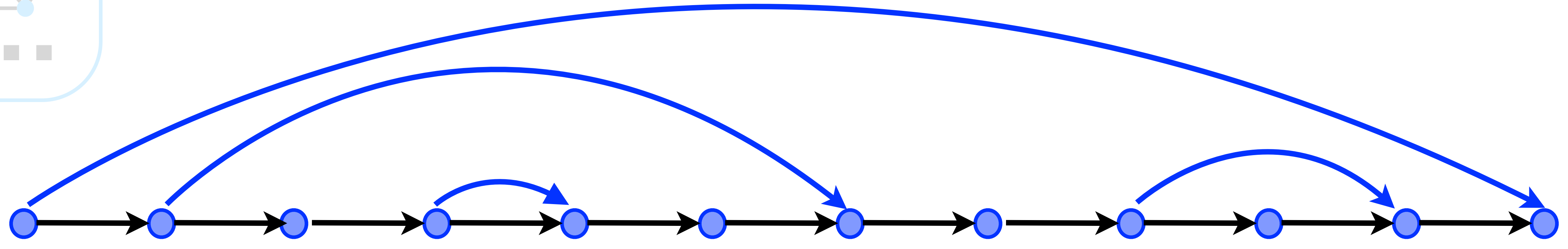
```
func f1
{while <true>
 {call f1 OR
  a OR
  exit;}
 return;}
```

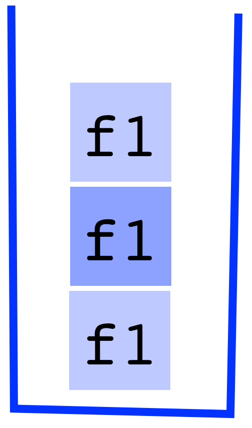


Understanding behaviours as graphs...

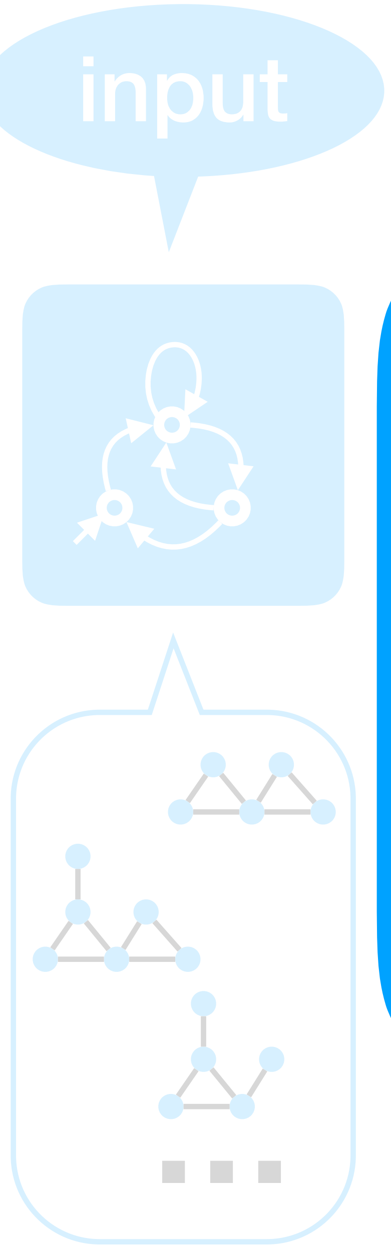
Pushdown systems
(recursive programs)

```
func f1
{while <true>
 {call f1 OR
   a OR
  exit;}
 return;}
```





Nested word
= word + binary nesting relation

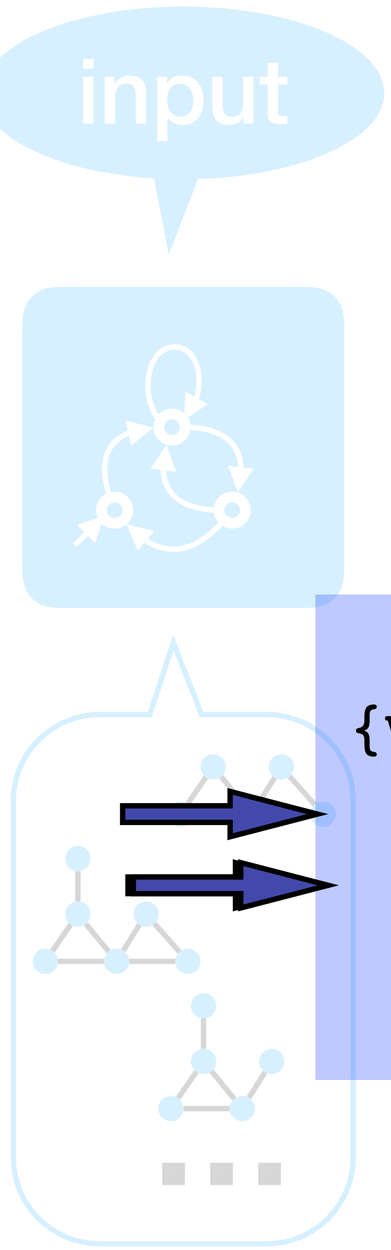


Understanding behaviours as graphs…

Pushdown systems

Nested Words
(AlurMadhusudan2006)

Multi-pushdown systems



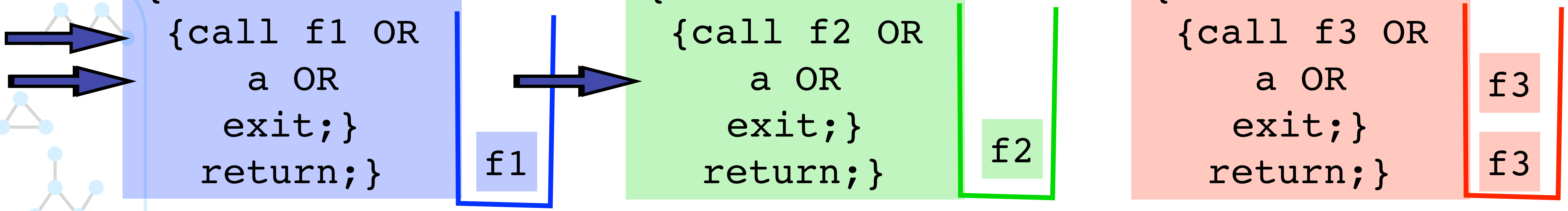
Understanding behaviours as graphs...

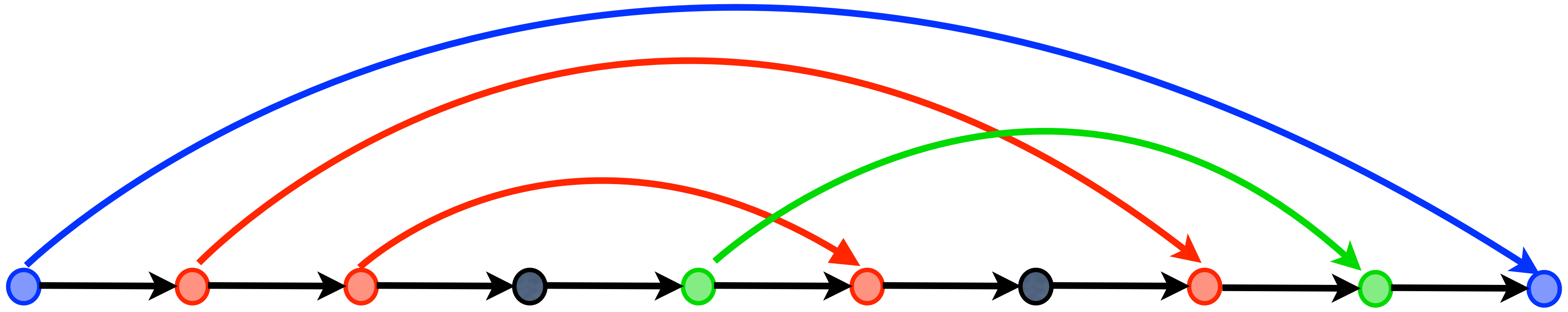
Multi-pushdown systems
(Concurrent Recursive Programs)

```
func f1
{while <true>
 {call f1 OR
   a OR
  exit;}
 return;}
```

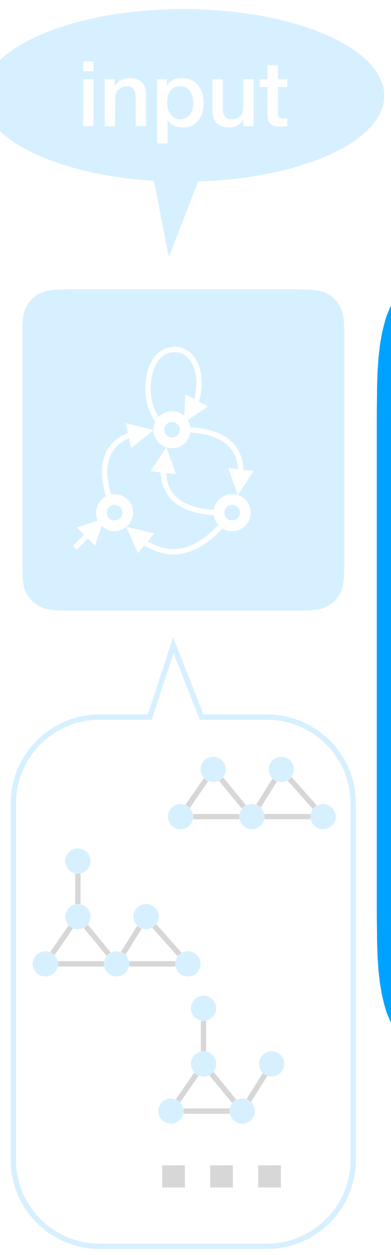
```
func f2
{while <true>
 {call f2 OR
   a OR
  exit;}
 return;}
```

```
func f3
{while <true>
 {call f3 OR
   a OR
  exit;}
 return;}
```





Multiply Nested word (MNW)
= word + multiple nesting relation



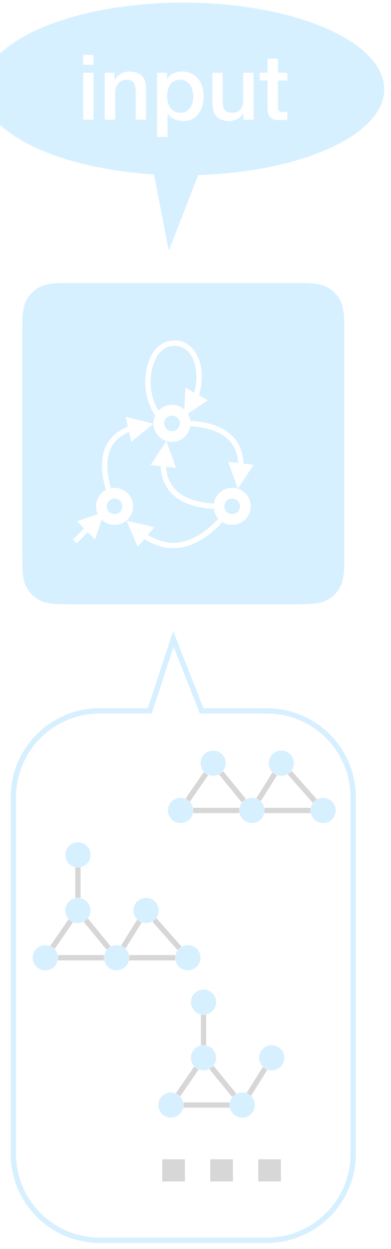
Understanding behaviours as graphs…

Pushdown systems

Nested Words
(AlurMadhusudan2006)

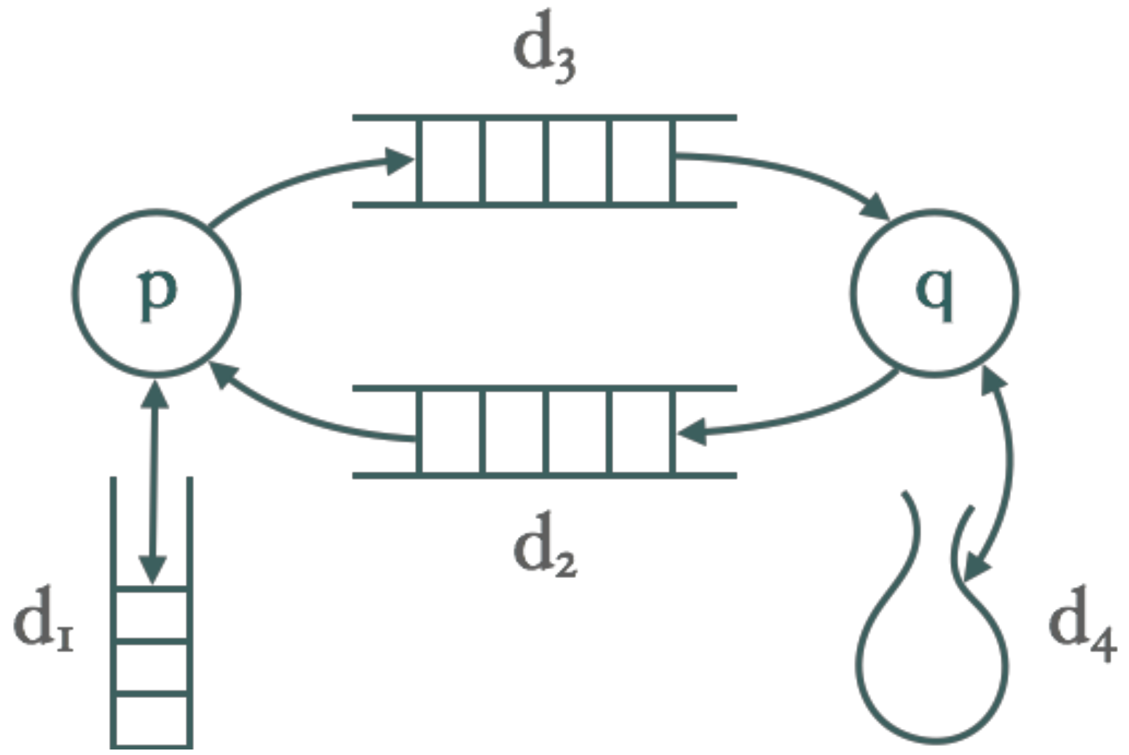
Multi-pushdown systems

Multiply Nested Words



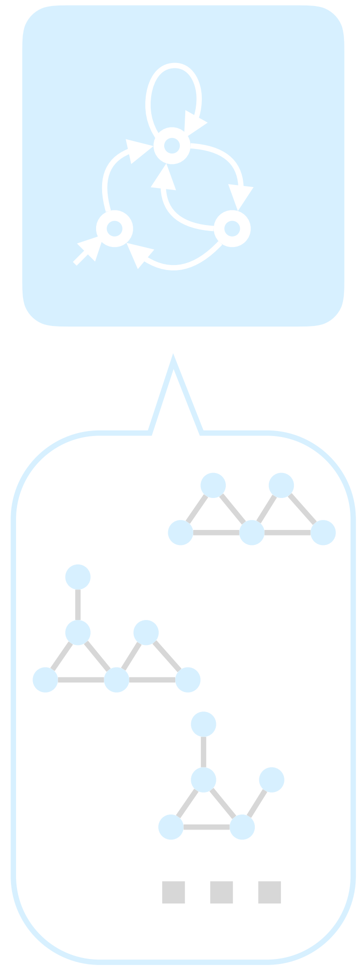
Understanding behaviours as graphs…

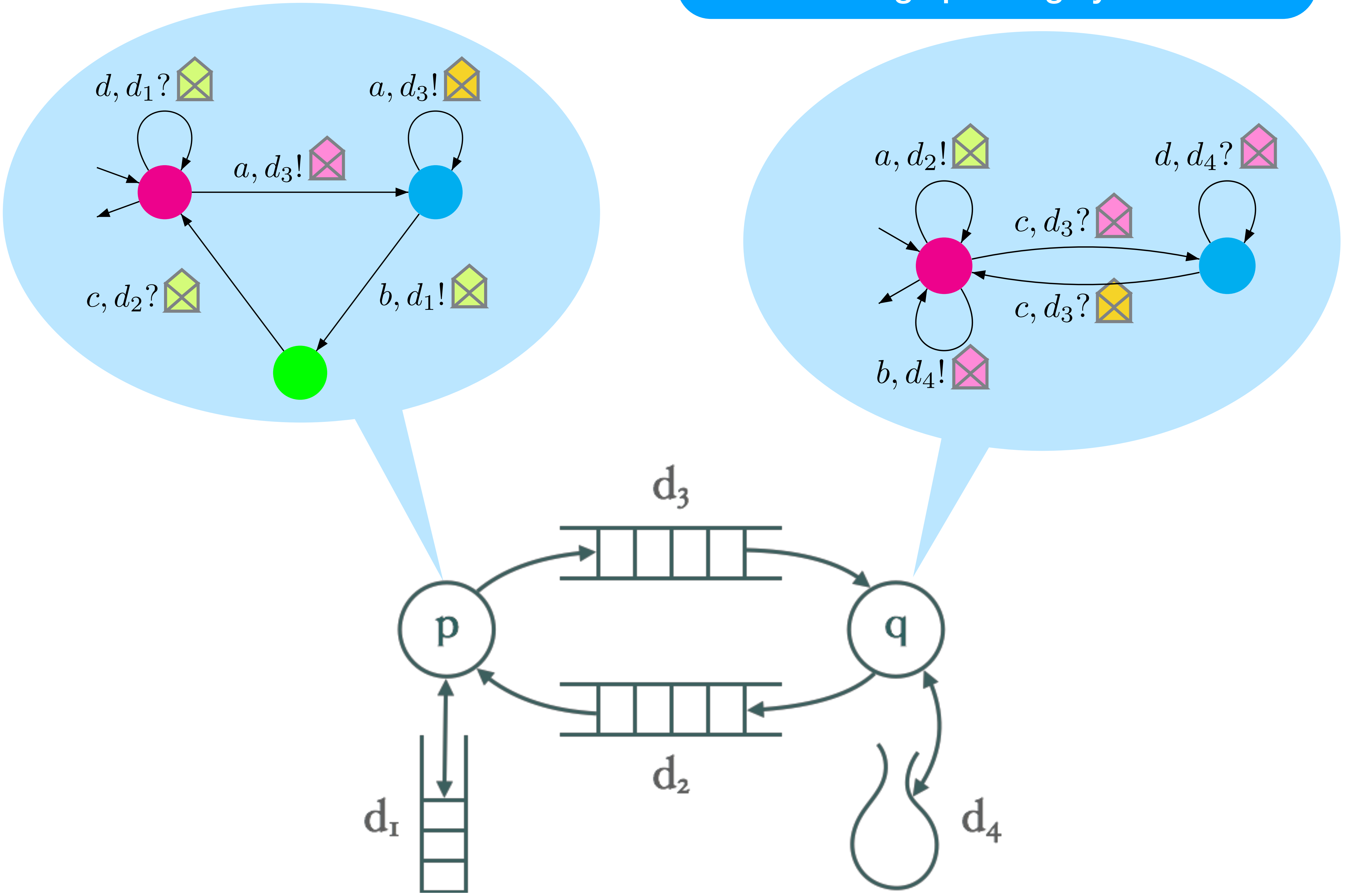
Message passing systems

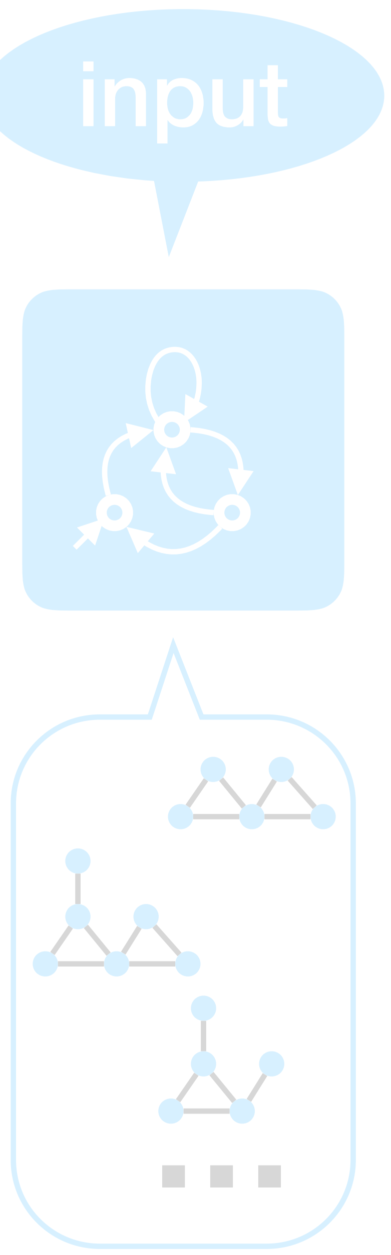


Understanding behaviours as graphs…

Message passing systems



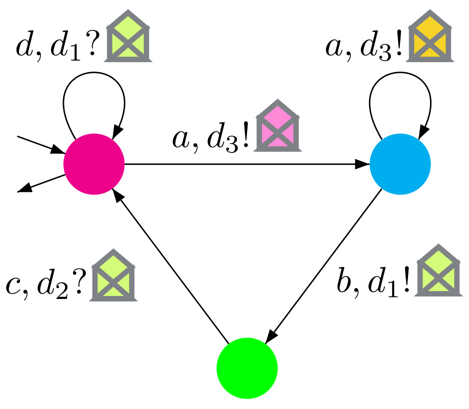


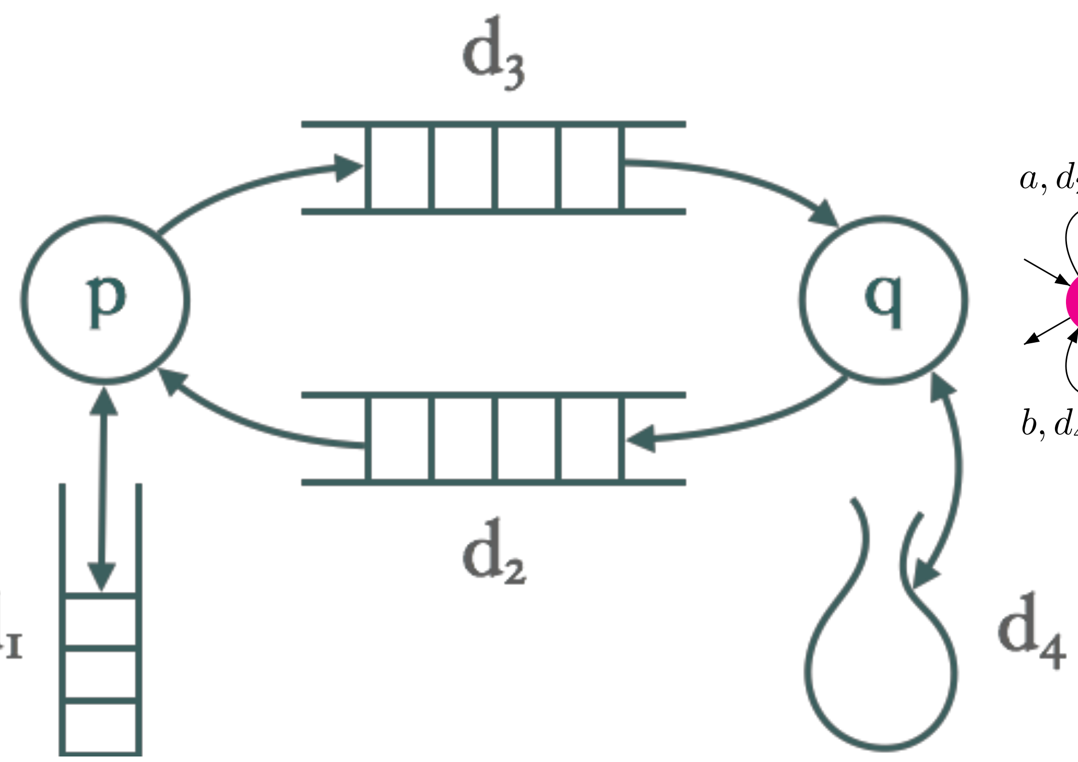


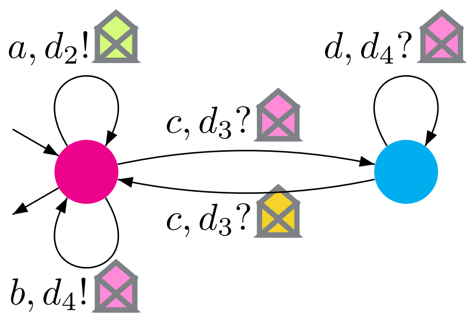
Understanding behaviours as graphs...

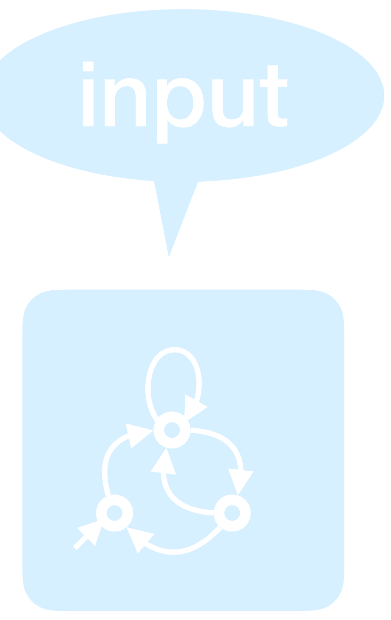
Message passing systems

$$\begin{tikzcd}
a \arrow[r] \arrow[drr] & a \arrow[r] \arrow[drrr] & b \arrow[r] \arrow[rr, bend left] & c \arrow[r] & d \\
b \arrow[r] \arrow[rrr, bend right] & a \arrow[r] \arrow[urr] & c \arrow[r] & d \arrow[r] & c
\end{tikzcd}$$









Understanding behaviours as graphs…

Pushdown systems

Nested Words
(AlurMadhusudan2006)

Multi-pushdown systems

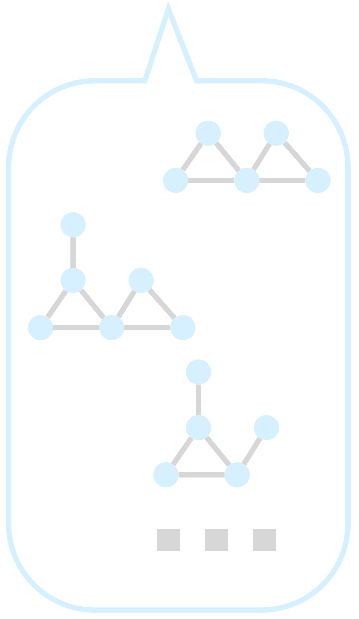
Multiply Nested Words

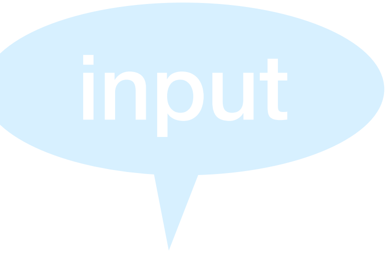
Communicating Finite State Machines

Message Sequence Charts

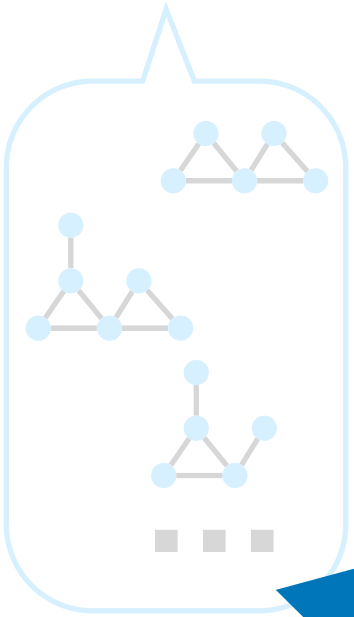
Message passing pushdowns

…





Understanding behaviours as graphs…



Pushdown systems

Nested Words
(AlurMadhusudan2006)

Multi-pushdown systems

rd

C ic ng Finite State Machines

Message Sequence Charts

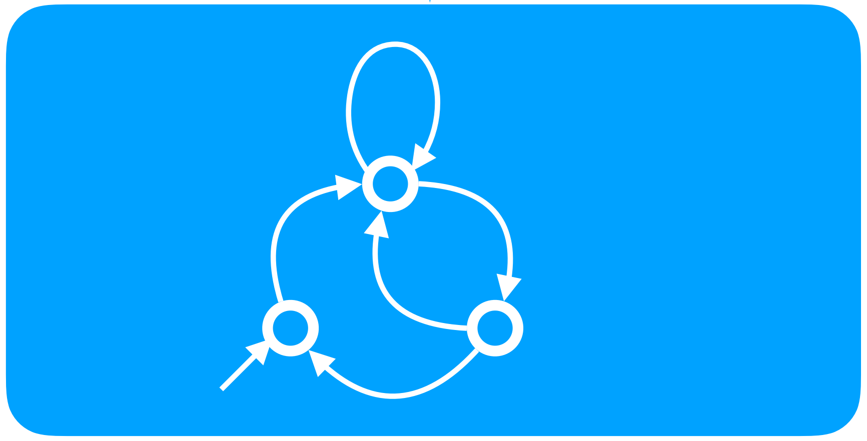
Message passing pushdowns

…

Finite state machines with auxiliary storage

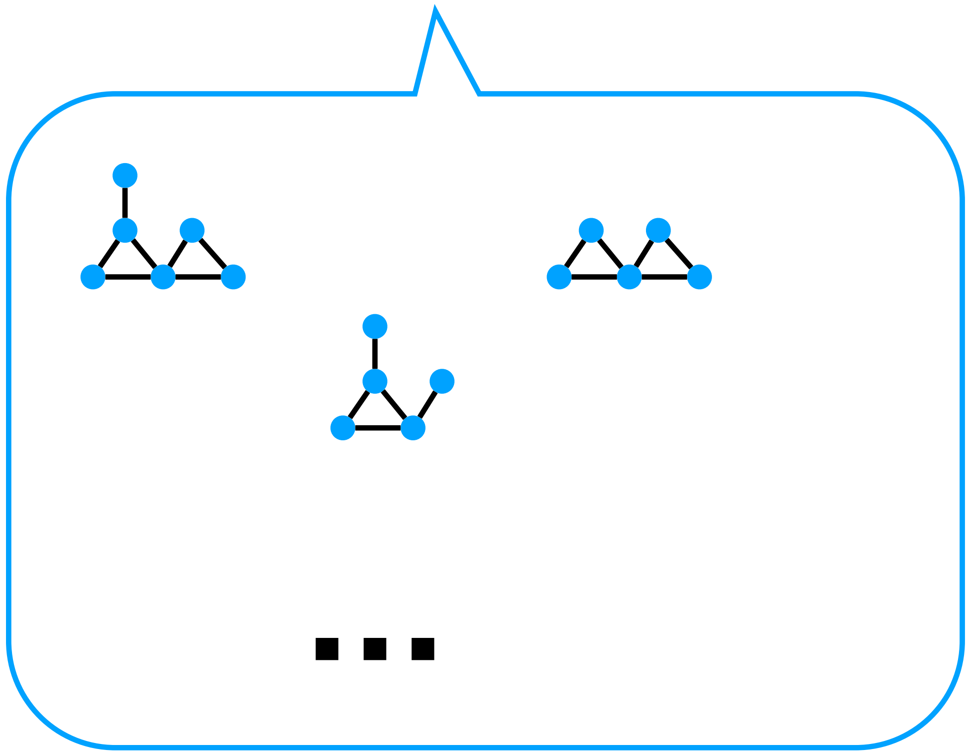
constructive writes and destructive reads

Model checking



$\models$

$\forall x\, \exists y\, \exists z\; E(x,y) \wedge E(x,z) \wedge x \neq y$

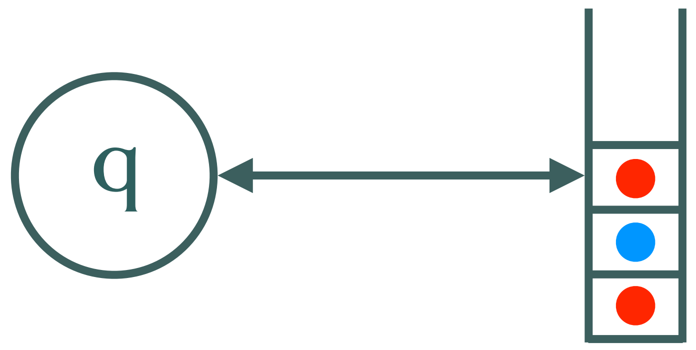


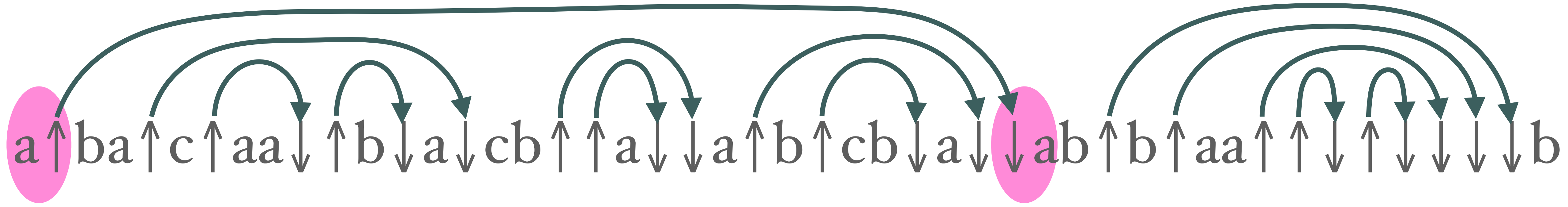
$\models$

$\forall x\, \exists y\, \exists z\; E(x,y) \wedge E(x,z) \wedge x \neq y$

Understanding specification over graphs...

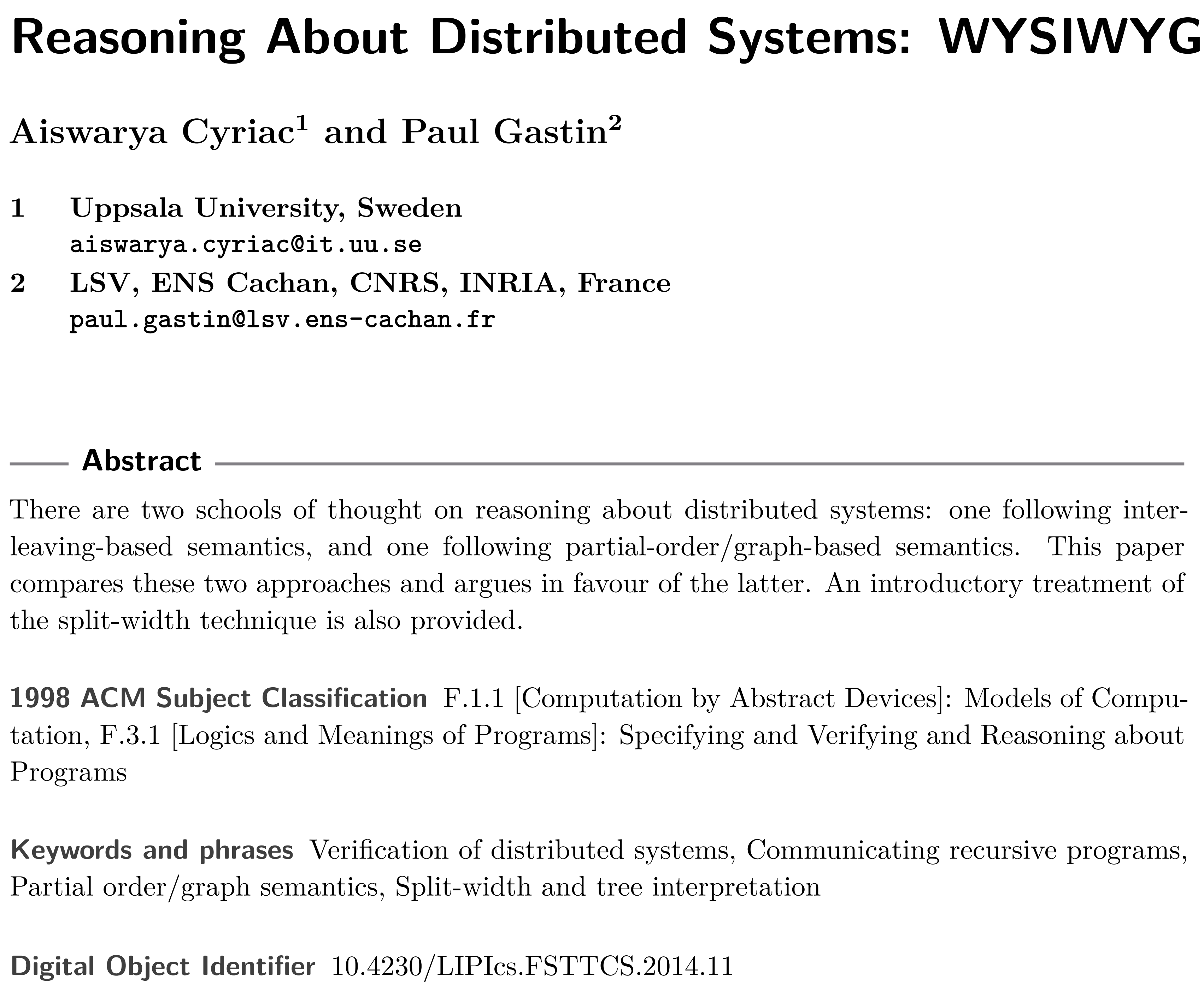
Answer the correct client for topmost requests





$$\forall x, y \left(\begin{array}{c} a(x-1) \wedge x \rhd y \wedge \\ \neg\exists z, z' \, (z \rhd z' \wedge z < x < z') \end{array} \right) \Rightarrow a(y+1)$$

Behaviours must be graphs and specifications should be over graphs…



Reasoning About Distributed Systems: WYSIWYG*

Aiswarya Cyriac[1] and Paul Gastin[2]

1 **Uppsala University, Sweden**
`aiswarya.cyriac@it.uu.se`

2 **LSV, ENS Cachan, CNRS, INRIA, France**
`paul.gastin@lsv.ens-cachan.fr`

Abstract

There are two schools of thought on reasoning about distributed systems: one following interleaving-based semantics, and one following partial-order/graph-based semantics. This paper compares these two approaches and argues in favour of the latter. An introductory treatment of the split-width technique is also provided.

1998 ACM Subject Classification F.1.1 [Computation by Abstract Devices]: Models of Computation, F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

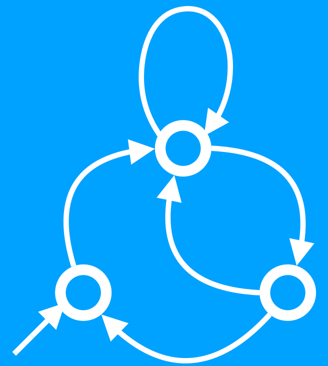
Keywords and phrases Verification of distributed systems, Communicating recursive programs, Partial order/graph semantics, Split-width and tree interpretation

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2014.11

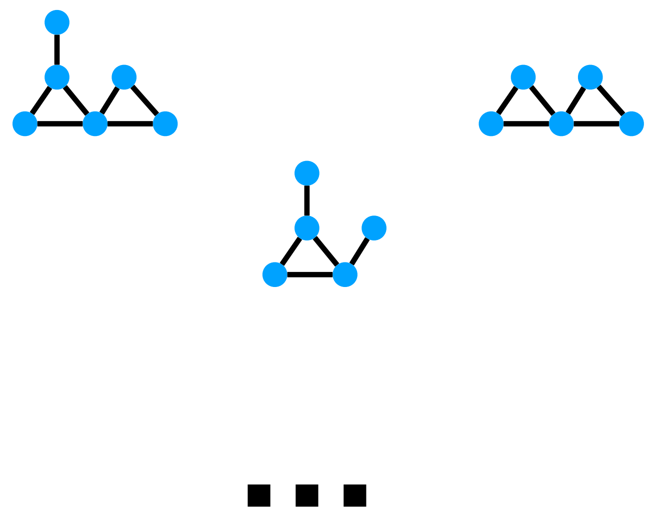
Model checking

input

input

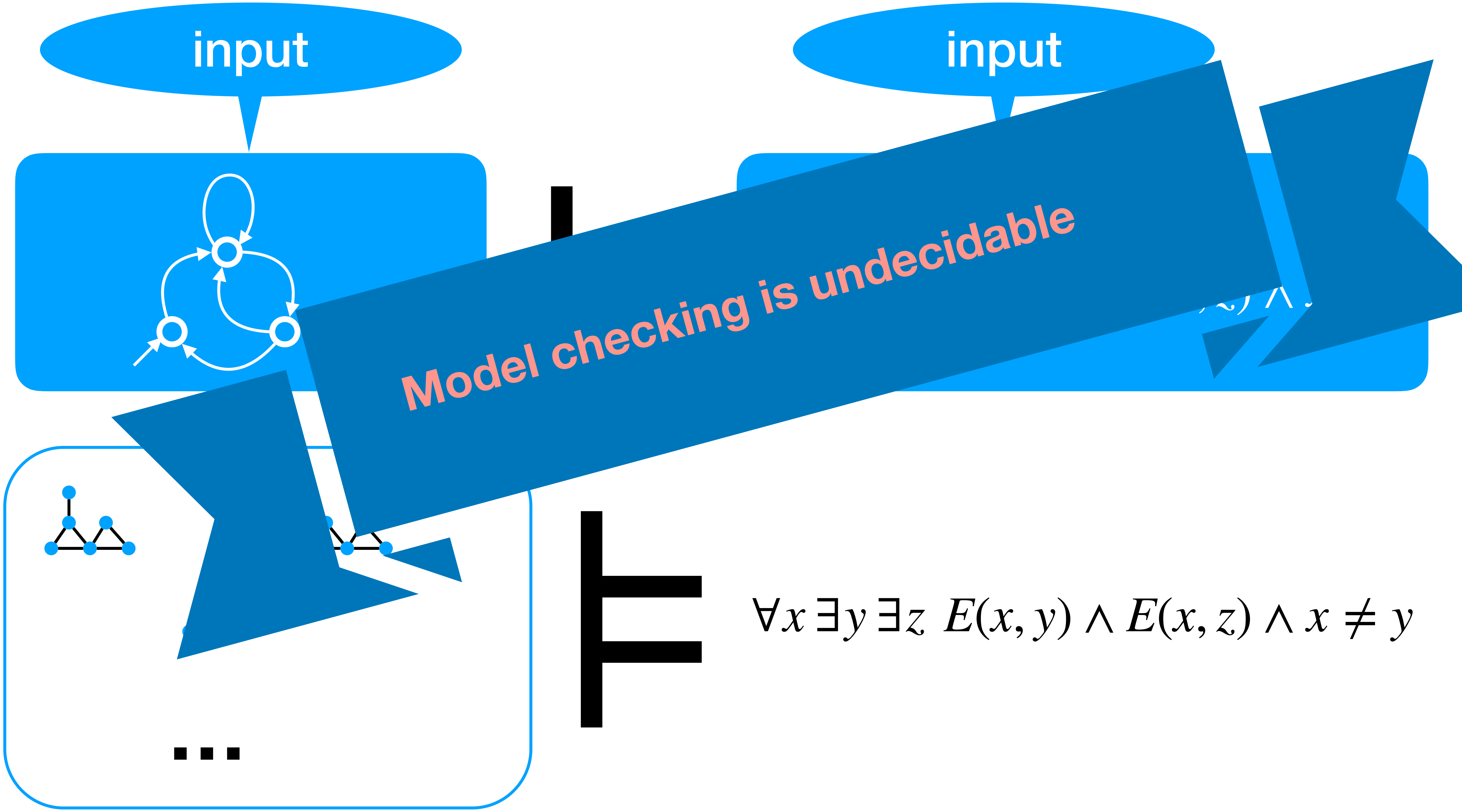


$$\models \quad \forall x\, \exists y\, \exists z\; E(x,y) \land E(x,z) \land x \neq y$$



$$\models \quad \forall x\, \exists y\, \exists z\; E(x,y) \land E(x,z) \land x \neq y$$

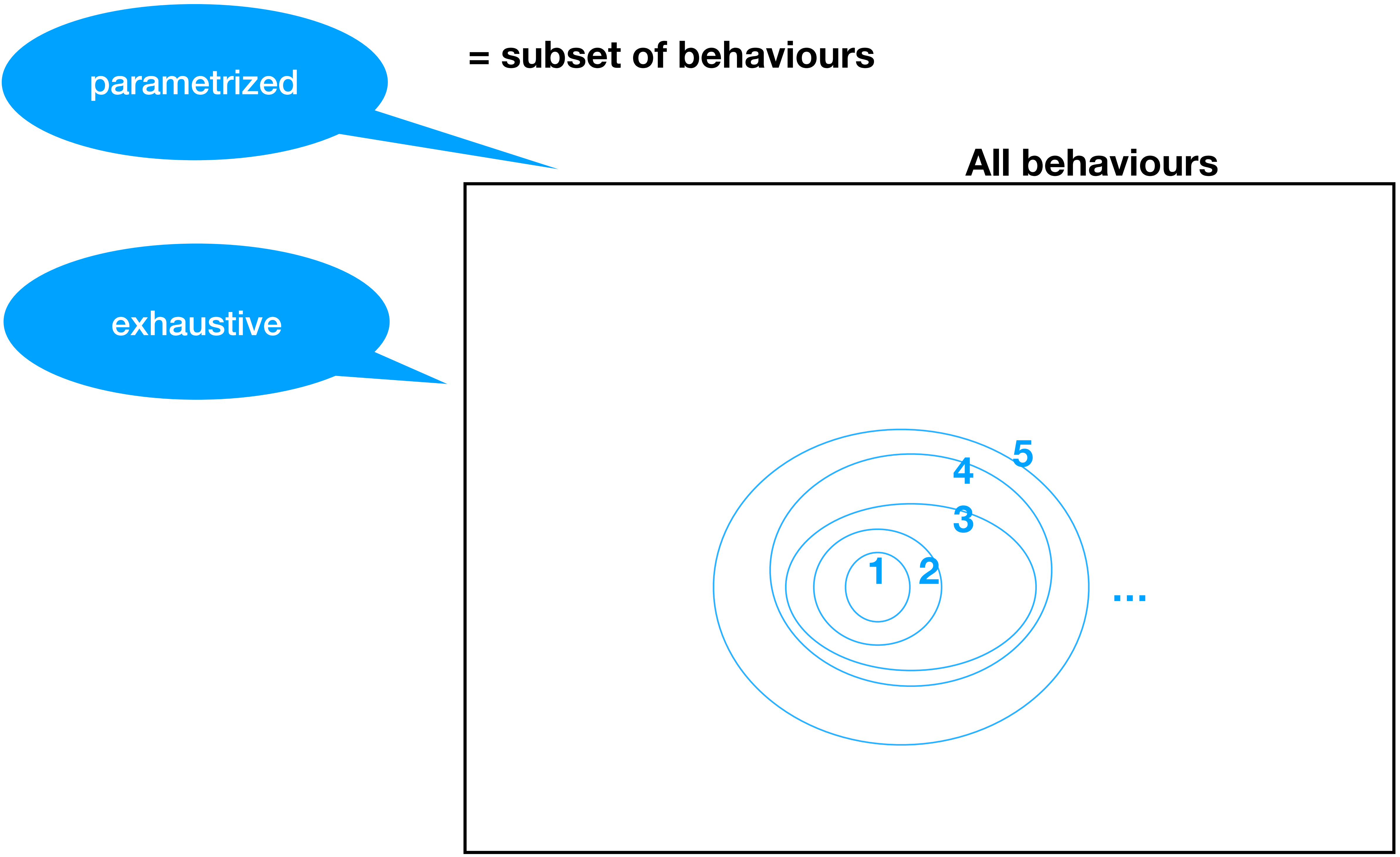
Model checking



$$\models \quad \forall x\, \exists y\, \exists z\ E(x,y) \wedge E(x,z) \wedge x \neq y$$

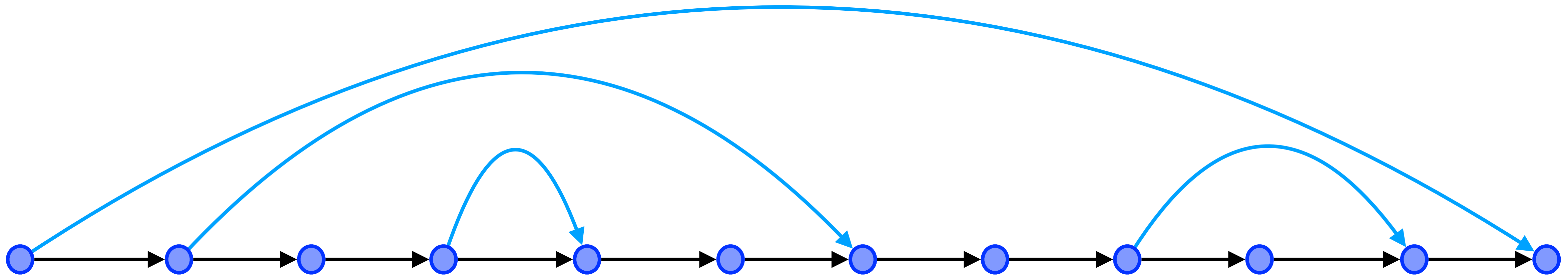
under-approximation

= subset of behaviours



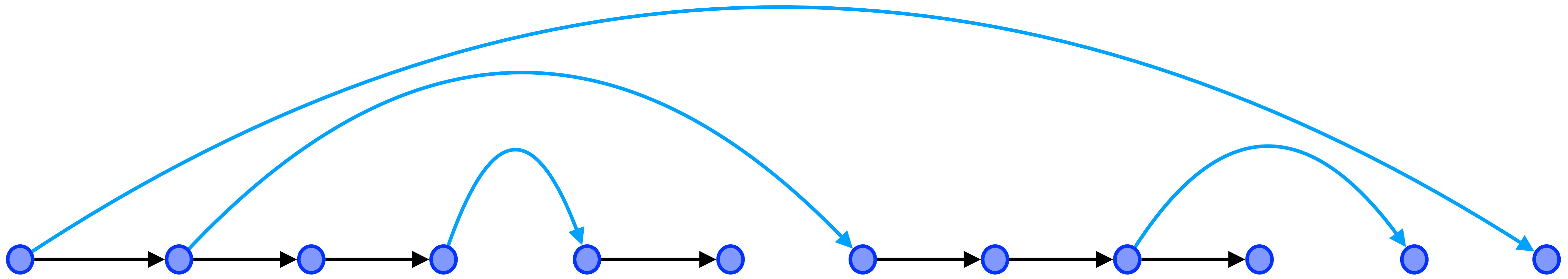
Tree-width as under-approximation

allows interpretation in trees…



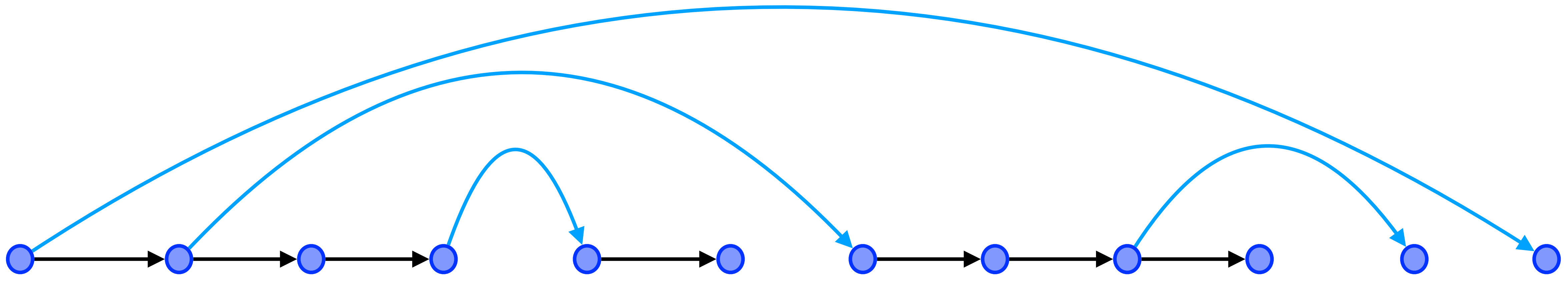
Tree-width as under-approximation

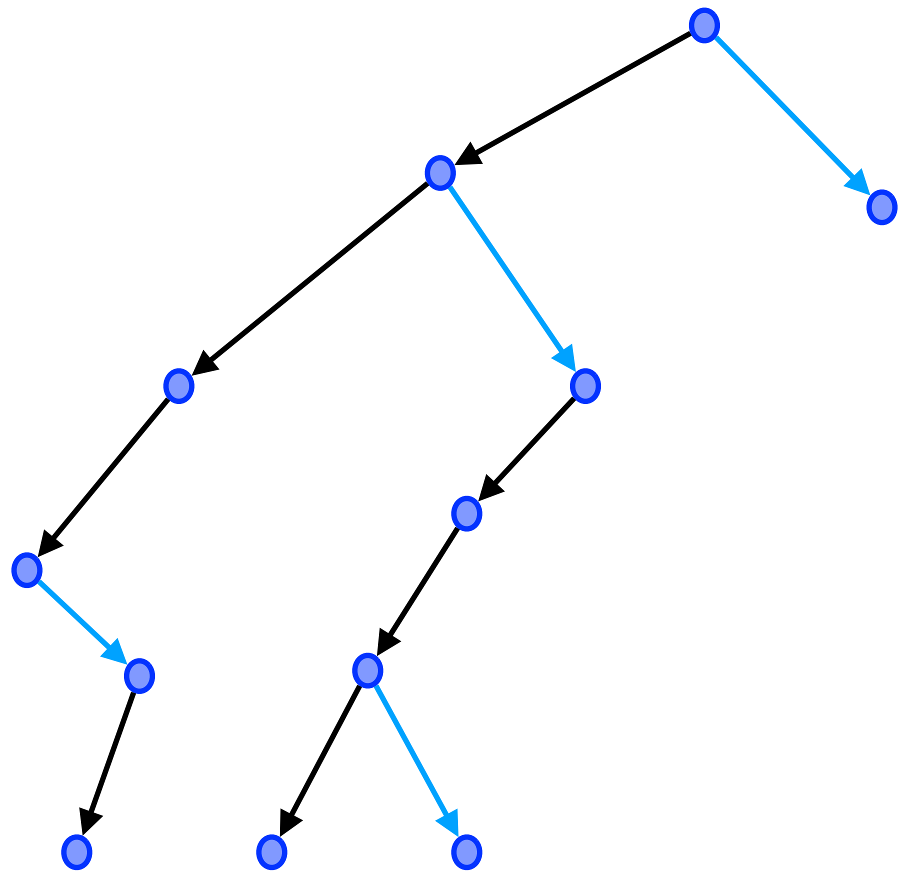
allows interpretation in trees…



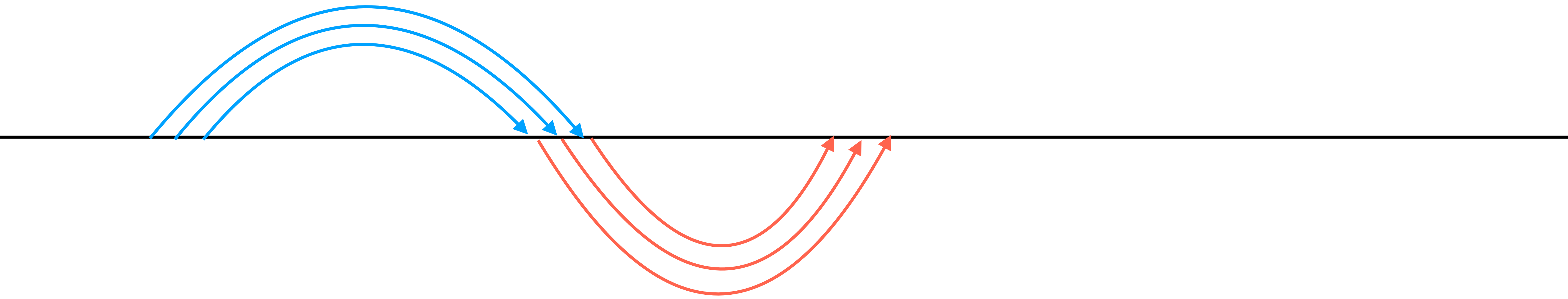
Tree-width as under-approximation

allows interpretation in trees…

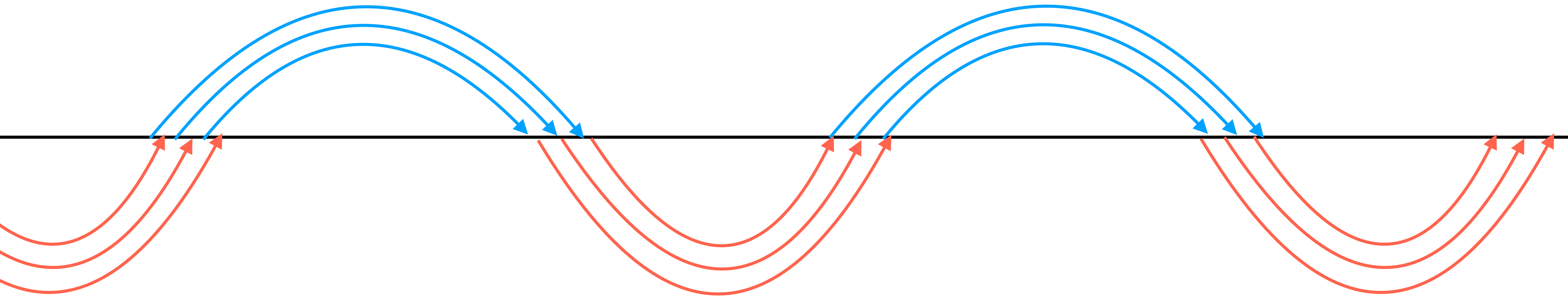




turing powerful models - unbounded tree-width

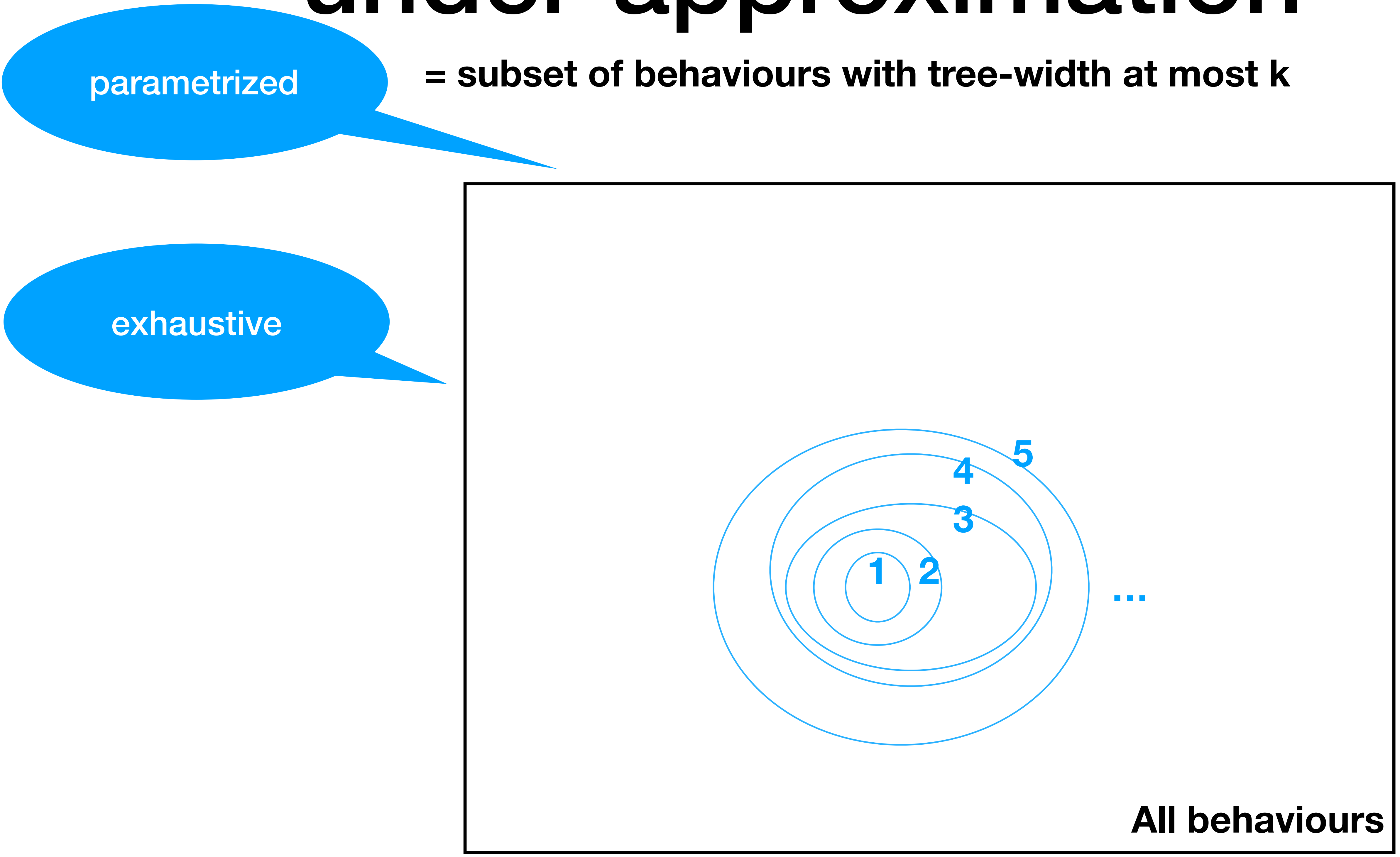


turing powerful models - unbounded tree-width



bounded tree-width under-approximation

= subset of behaviours with tree-width at most k

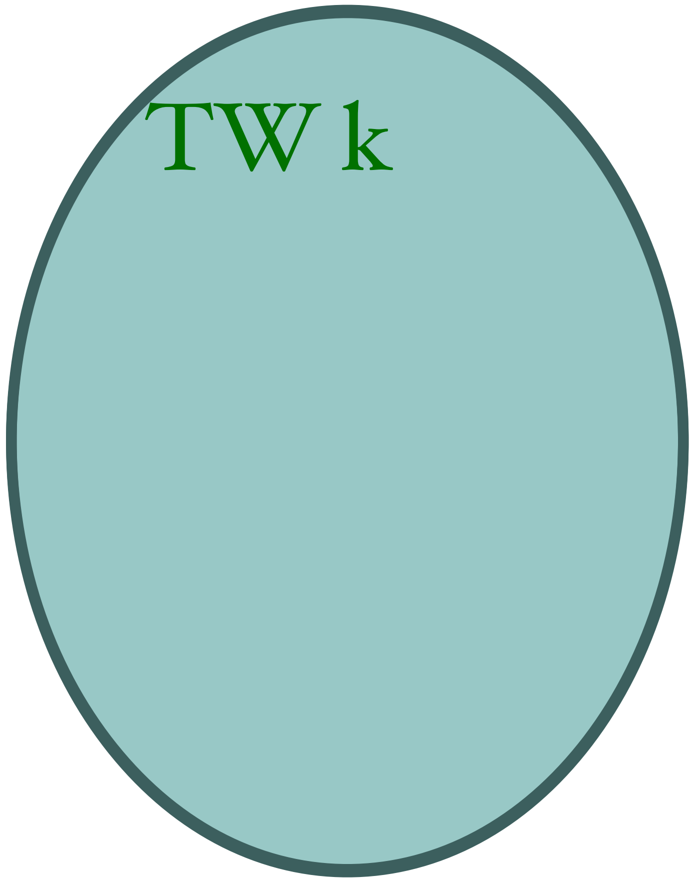


verification results with bounded tree-width

Non-emptiness checking :	ExpTime Complete
Temporal logic model checking :	ExpTime Complete
MSO model checking :	non-elementary

proof idea = move to the world of trees

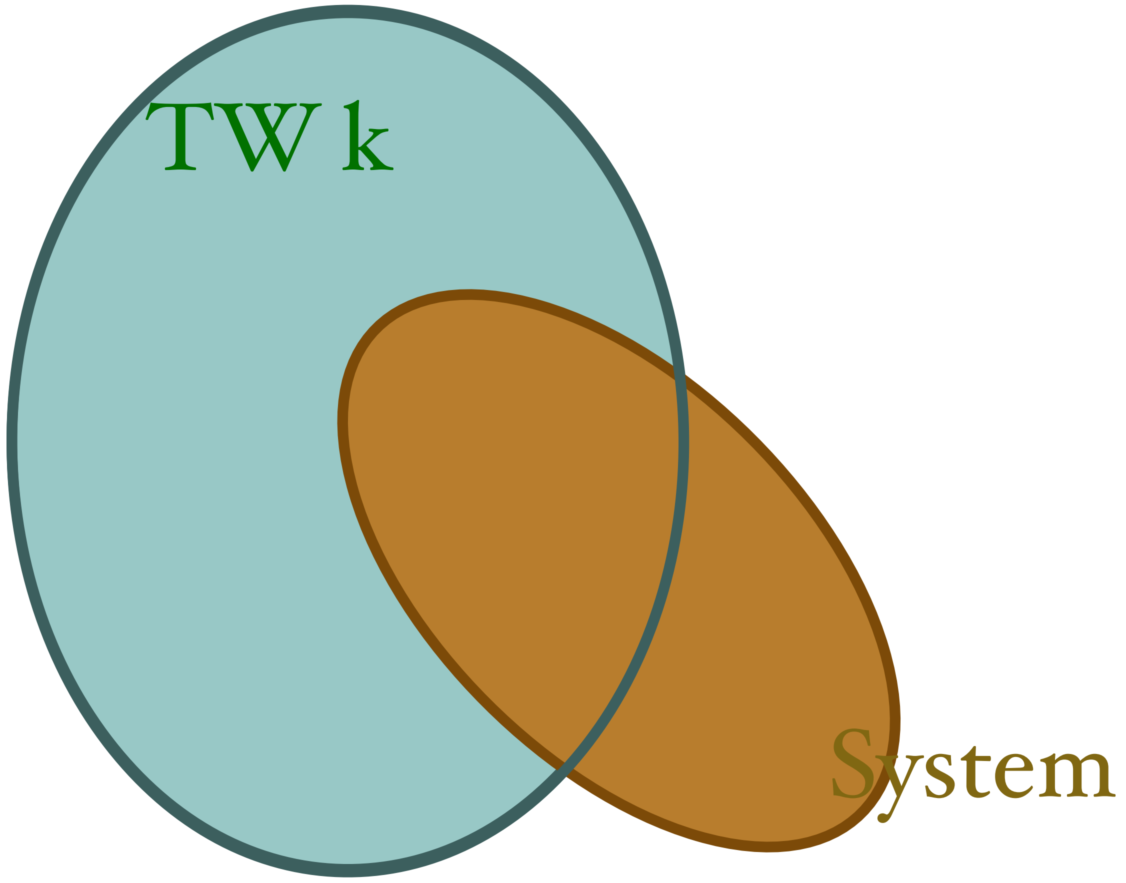
get tree automata to recognize all
tree interpretations



proof idea = move to the world of trees

get tree automata to recognize all tree interpretations

get tree automata to recognize all trees representing valid runs in the system

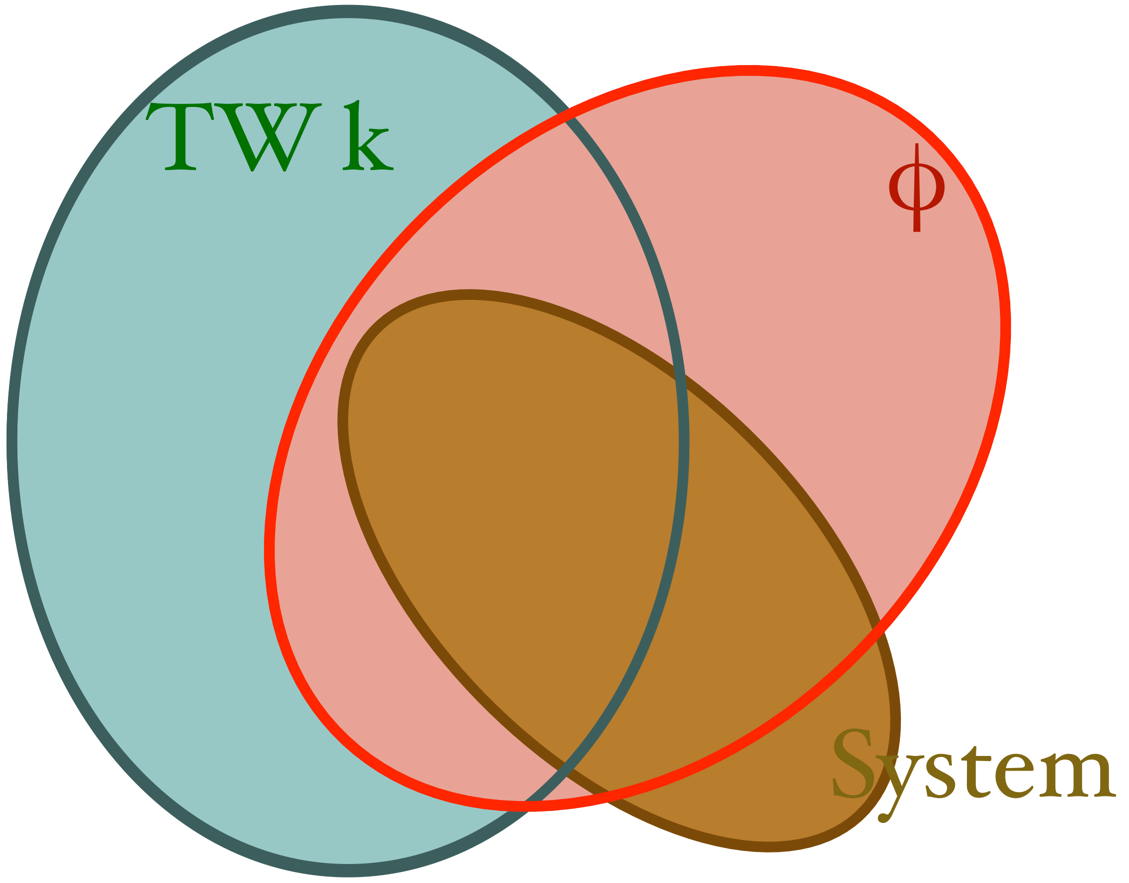


proof idea = move to the world of trees

get tree automata to recognize all tree interpretations

get tree automata to recognize all trees representing valid runs in the system

get tree automata for trees representing runs violating the property



The Tree Width of Auxiliary Storage

P. Madhusudan
University of Illinois at Urbana-Champaign, USA
madhu@illinois.edu

Gennaro Parlato
LIAFA, CNRS and University of Paris Diderot, France.
gennaro@liafa.jussieu.fr

Abstract

We propose a generalization of results on the decidability of emptiness for several restricted classes of sequential and distributed automata with auxiliary storage (stacks, queues) that have recently been proved. Our generalization relies on reducing emptiness of these automata to finite-state *graph automata* (without storage) restricted to monadic second-order (MSO) definable graphs of bounded tree-width, where the graph structure encodes the mechanism provided by the auxiliary storage. Our results outline a uniform mechanism to derive emptiness algorithms for automata, explaining and simplifying several existing results, as well as proving new decidability results.

Categories and Subject Descriptors F.1.1 [*Theory of Computation*]: Models of Computation: Automata; D.2.4 [*Software Engineering*]: Software/Program Verification: Model checking; F.4.3 [*Theory of Computation*]: Formal Languages: Decision problems

General Terms Algorithms, Reliability, Theory, Verification

Keywords model checking, automata, decision procedures, bounded tree-width

1. Introduction

Several classes of automata with auxiliary storage have been defined over the years that have a decidable emptiness problem. Classic models like pushdown automata utilizing a *stack* have a decidable emptiness problem [14], and several new models like restricted classes of multi-stack pushdown automata, automata with queues, and automata with both stacks and queues, have been proved decidable recently [8, 15, 17, 22].

The decidability of emptiness of these automata has often been motivated for *model-checking* systems. Software models can be captured using automata with auxiliary storage, as stacks can model the *control recursion* in programs while queues model *FIFO communication between processes*. In abstraction-based model-checking, data domains get abstracted from programs, resulting in automata models (e.g., the SLAM tool builds pushdown automata models using *predicate abstraction* [7], and the GETAFIX tool model-checks both single-stack and multi-stack automata models [18, 19]). The emptiness problem for these automata is the most relevant problem as it directly corresponds to checking reachability of an error state.

However, the various identified decidable restrictions on these automata are, for the most part, *awkward* in their definitions—e.g. emptiness of multi-stack pushdown automata where pushes to any stack is allowed at any time, but popping is restricted to the first non-empty stack is decidable! [8]. Yet, relaxing these definitions to more natural ones seems to either destroy decidability or their power. It is hence natural to ask: why do these automata have decidable emptiness problems? Is there a common underlying principle that explains their decidability?

We propose, in this paper, a general criterion that uniformly explains many such results— several restricted uses of auxiliary storage are decidable because they can be simulated by *graph automata* working on graphs that capture the storage as well as their sequential or distributed nature, and are also of bounded tree-width.

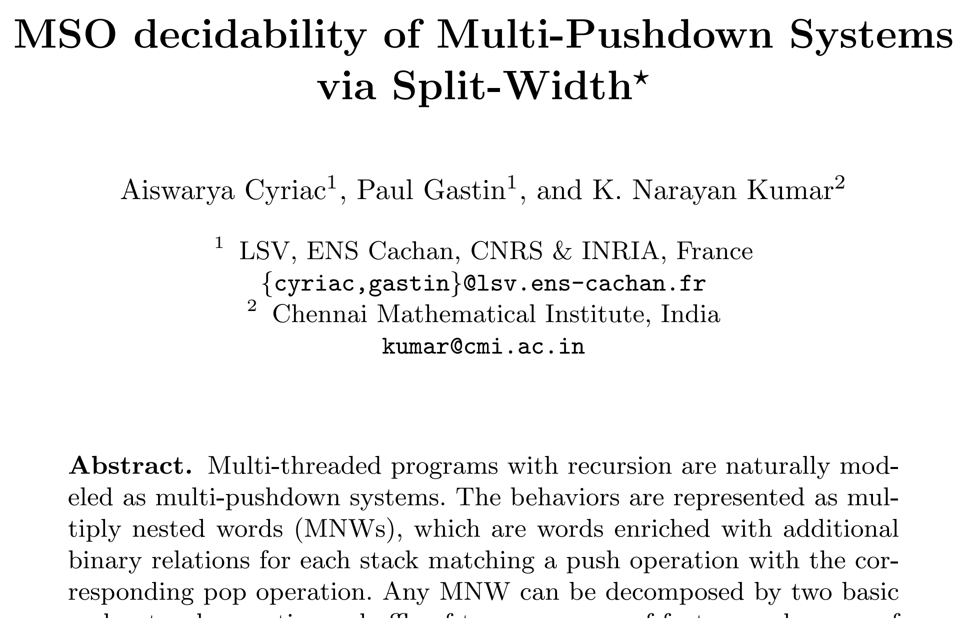
More precisely, we can show, using generalizations of known results on the decidability of *satisfiability* of monadic second-order logic (MSO) on bounded tree-width graphs [9, 23], that graph automata on *MSO-definable* graphs of bounded tree-width are decidable. Graph automata [24] are finite-state automata (without auxiliary storage) that accept or reject graphs using *tilings of the graph using states*, where the restrictions on tiling determine the graphs that get accepted. The general decidability of emptiness of graph automata on MSO-definable graphs follows since the existence of acceptable tilings is MSO-definable.

We proceed to show that several sequential/distributed automata with an auxiliary storage (we consider stacks and queues only in this paper), can be realized as *graph automata* working on single or multiple directed paths augmented with special edges to capture the mechanism of the storage. Intuitively, a symbol that gets stored in a stack/queue and later gets retrieved can be simulated by a graph automaton working on a graph where there is a special edge between the point where the symbol gets stored to the point where it gets retrieved. A graph automaton can retrieve the symbol at the retrieval point by using an appropriate tiling of this special edge.

The idea of converting automata with storage to graph automata without storage but working on specialized graphs is that it allows us to examine the *complexity* of storage using the *structure* of the graph that simulates it. We show that many automata with a tractable emptiness problem can be converted to graph automata working on MSO definable graphs of bounded tree-width, from which decidability of their emptiness follows.

We prove the simulation of the following classes of automata with auxiliary storage by graph automata working on MSO-definable bounded tree-width graphs:

- Multi-stack pushdown automata with bounded context-switching: This is the class of multi-stack automata where each computation of the automaton can be divided into k stages, where in each stage the automaton touches only one stack (proved decidable first in [22]). We show that they can be simulated by graph automata on graphs of tree-width $O(k)$.

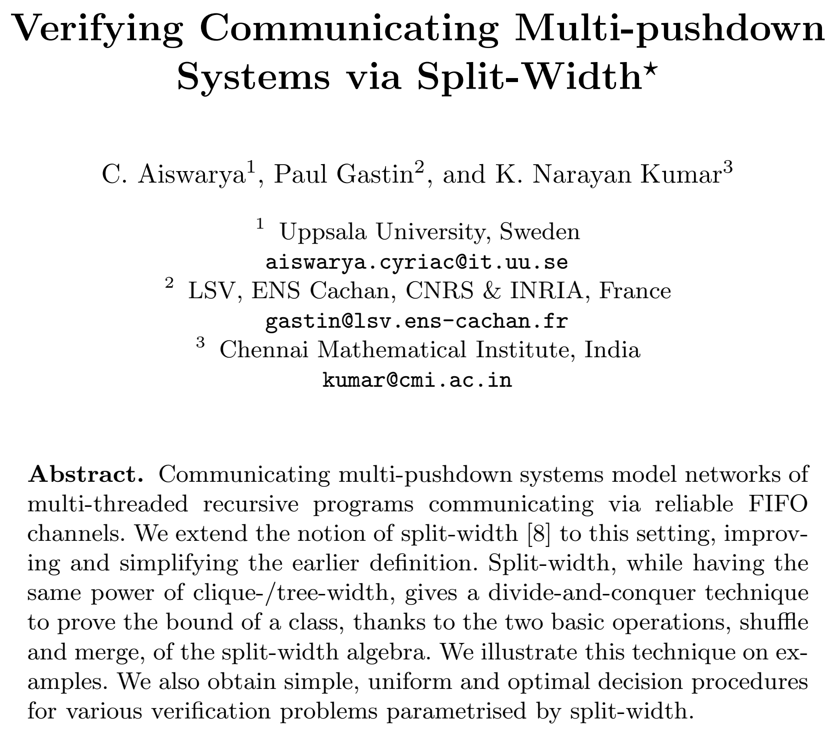


MSO decidability of Multi-Pushdown Systems via Split-Width⋆

Aiswarya Cyriac[1], Paul Gastin[1], and K. Narayan Kumar[2]

[1] LSV, ENS Cachan, CNRS & INRIA, France
{cyriac,gastin}@lsv.ens-cachan.fr
[2] Chennai Mathematical Institute, India
kumar@cmi.ac.in

Abstract. Multi-threaded programs with recursion are naturally modeled as multi-pushdown systems. The behaviors are represented as multiply nested words (MNWs), which are words enriched with additional binary relations for each stack matching a push operation with the corresponding pop operation. Any MNW can be decomposed by two basic



Verifying Communicating Multi-pushdown Systems via Split-Width⋆

C. Aiswarya[1], Paul Gastin[2], and K. Narayan Kumar[3]

[1] Uppsala University, Sweden
aiswarya.cyriac@it.uu.se
[2] LSV, ENS Cachan, CNRS & INRIA, France
gastin@lsv.ens-cachan.fr
[3] Chennai Mathematical Institute, India
kumar@cmi.ac.in

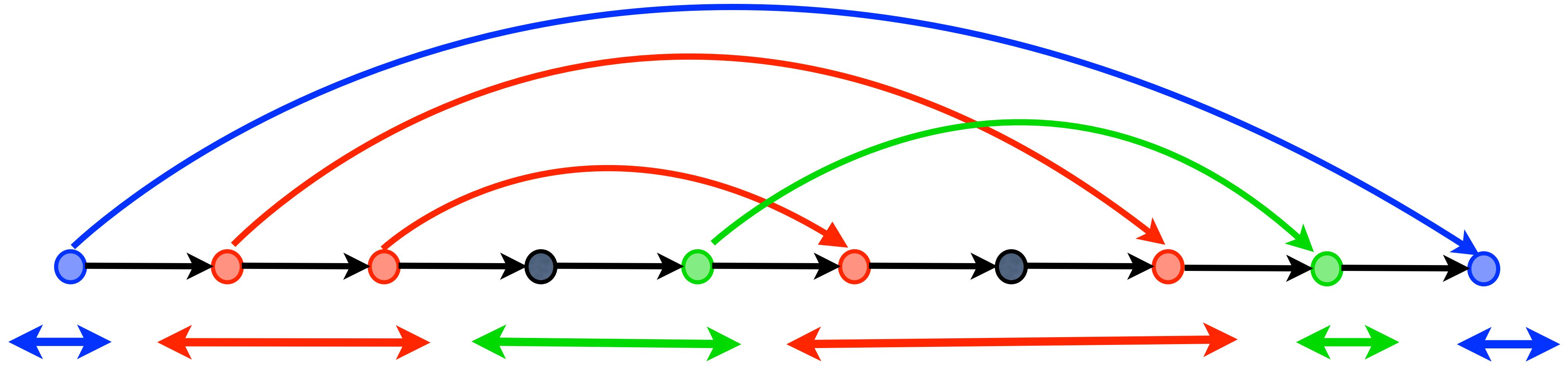
Abstract. Communicating multi-pushdown systems model networks of multi-threaded recursive programs communicating via reliable FIFO channels. We extend the notion of split-width [8] to this setting, improving and simplifying the earlier definition. Split-width, while having the same power of clique-/tree-width, gives a divide-and-conquer technique to prove the bound of a class, thanks to the two basic operations, shuffle and merge, of the split-width algebra. We illustrate this technique on examples. We also obtain simple, uniform and optimal decision procedures for various verification problems parametrised by split-width.

what do bounded tree-width look like?

- Bounded channel size
- Existentially bounded [Genest et al.,]
- Acyclic Architectures [Genest et al.,Clemente et al.]
- Bounded context switching [LaTorre et al.]
- Bounded context switching [Qadeer,Rehof]
- Bounded phase [LaTorre et al.]
- Bounded scope [LaTorre et al.]
- Priority ordering [Atig et al. Breveligiri et al, Saivasan et al.]

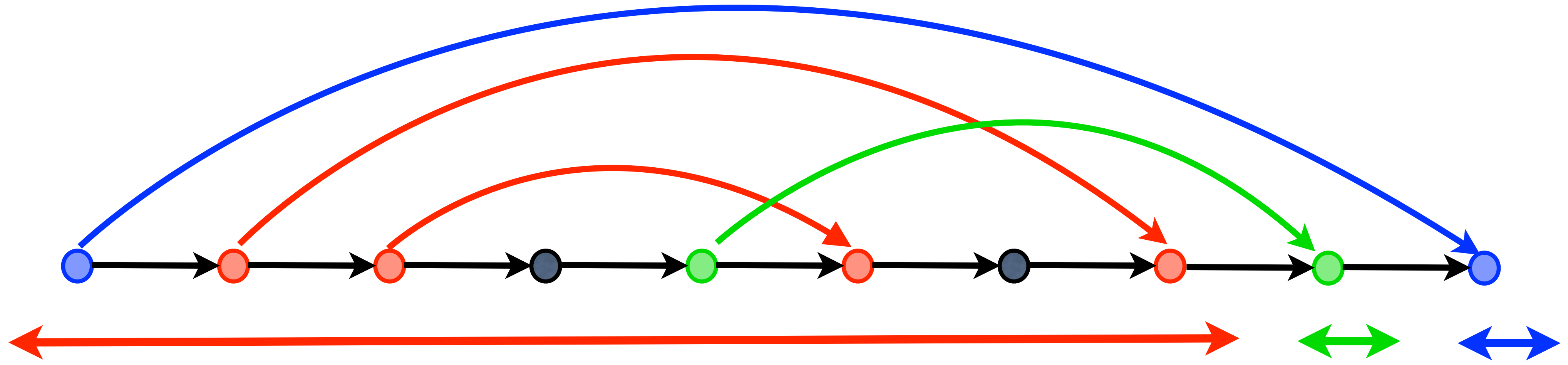
***list not exhaustive!!**

bounded context

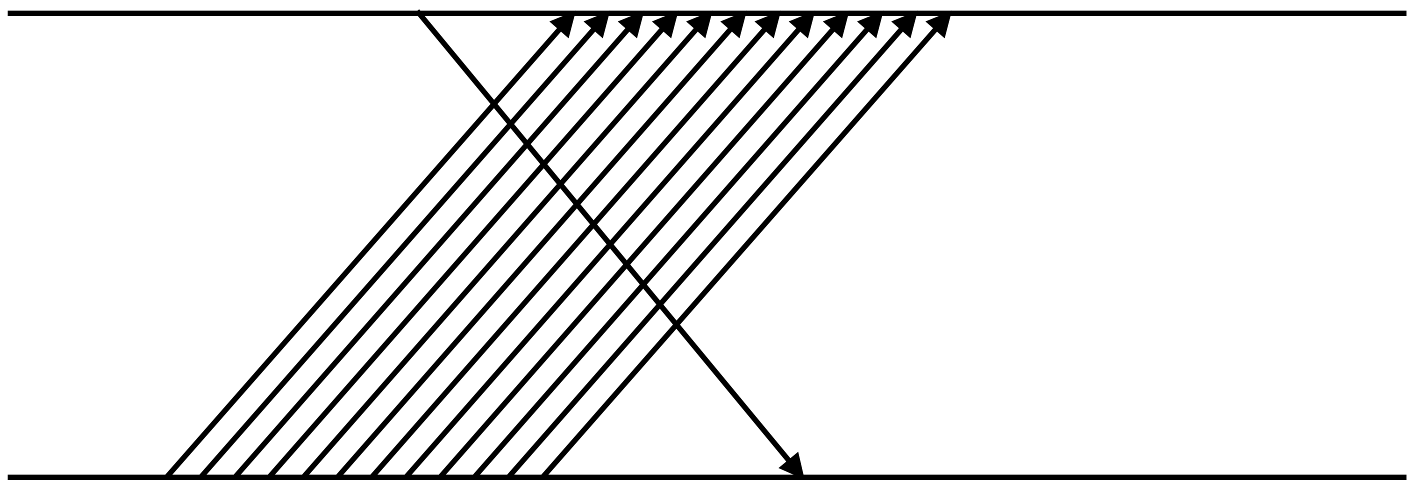


bounded scope : inside every curved edge bounded context

bounded phase



Existentially bounded:
at least one linearization where the channel size is bounded



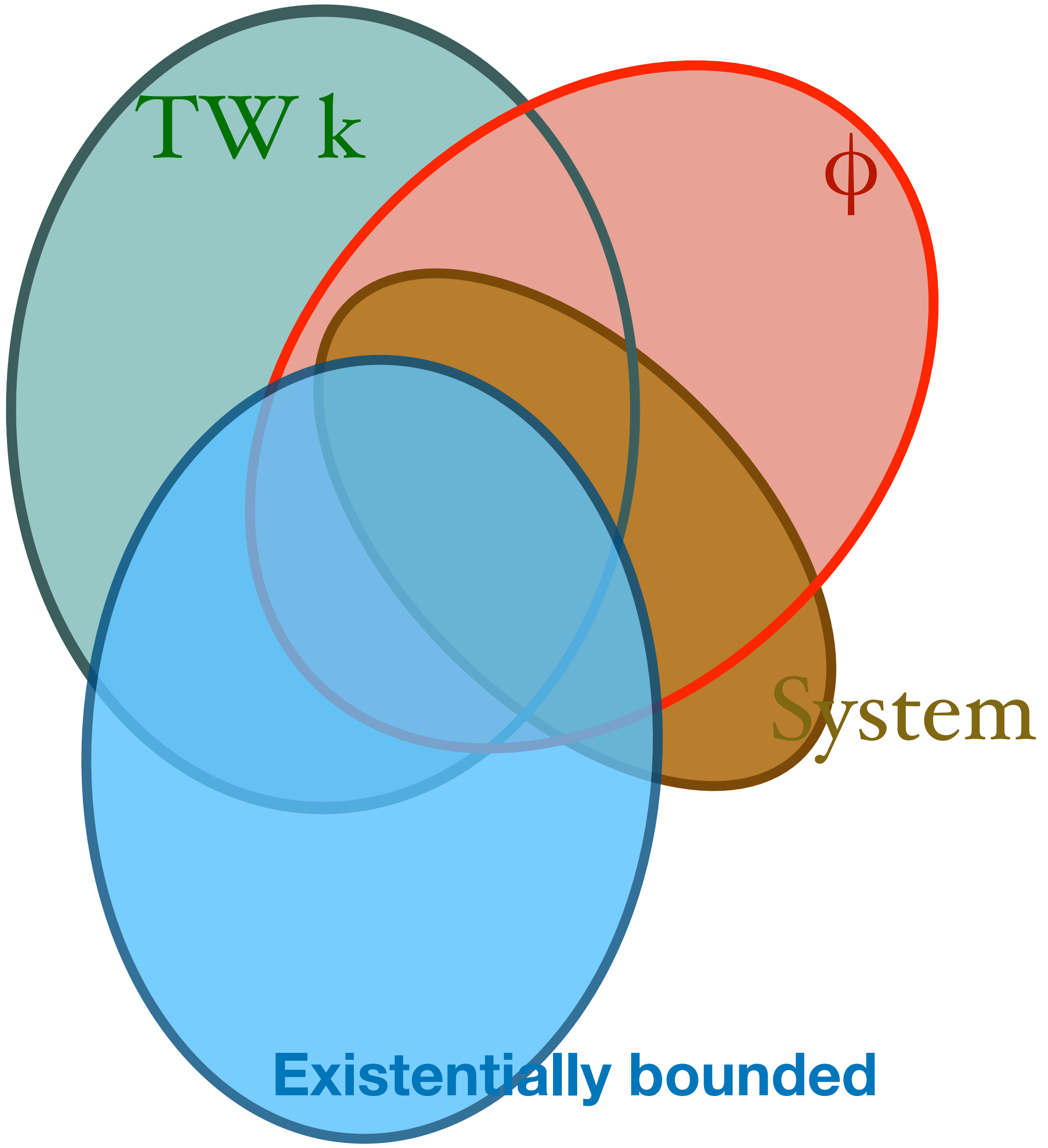
proof idea = move to the world of trees

get tree automata to recognize all tree interpretations

get tree automata to recognize all trees representing valid runs in the system

get tree automata for trees representing runs violating the property

get tree automata for the under-approximation



complete..

if an under-approximation gives decidability for MSO, then it has bounded tree-width

More results

Handle timing constraints

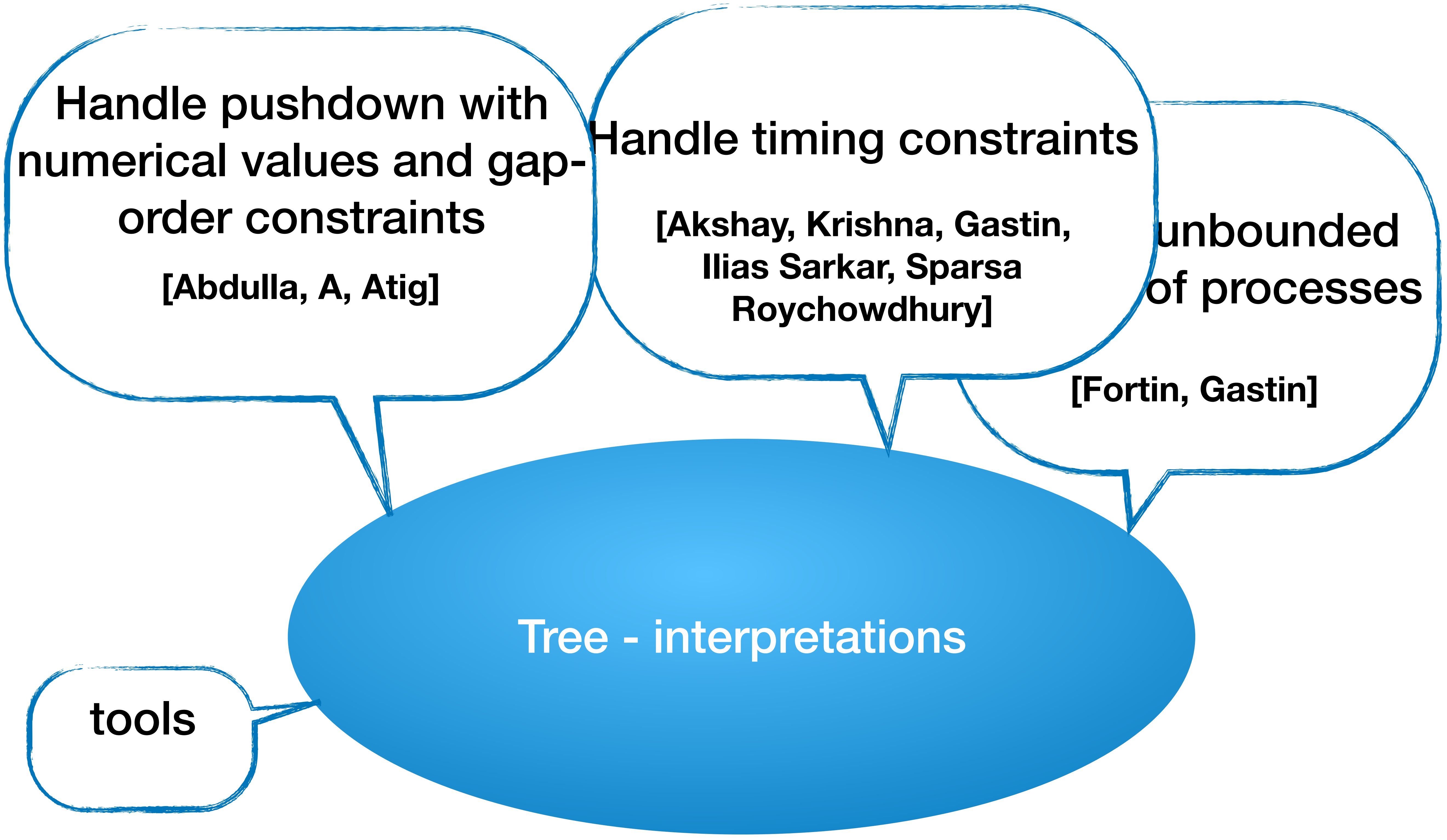
[Akshay, Krishna, Gastin, Ilias Sarkar, Sparsa Roychowdhury]

Handle unbounded number of processes

[Fortin, Gastin]

Tree - interpretations

More results


Handle pushdown with numerical values and gap-order constraints
[Abdulla, A, Atig]
Handle timing constraints
[Akshay, Krishna, Gastin, Ilias Sarkar, Sparsa Roychowdhury]
unbounded of processes
[Fortin, Gastin]
tools
Tree - interpretations

Open!

synthesis

From a tree-automata over interpretations is there a translation to system?

Tree - interpretations