Forward Analysis of Depth-Bounded Processes

Thomas Wies Damien Zufferey Thomas A. Henzinger

IST Austria (Institute of Science and Technology Austria)

March 30, 2010

From the LIFT web framework (using SCALA actors)

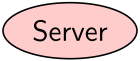
```
class DynamicBlogView extends CometActor {
  //...
  override def localSetup {
    //...
    (BlogCache.cache !? AddBlogWatcher(this, this.blogid)) match {
      case BlogUpdate(entries) => this.blog = entries
    }
  }

  override def lowPriority : PartialFunction[Any, Unit] = {
    case BlogUpdate(entries : List[Entry]) => this.blog = entries; reRender(false)
  }
}

class BlogCache extends LiftActor {
  //...
  protected def messageHandler =
    {
      case AddBlogWatcher(me, id) =>
        val blog = cache.getOrElse(id, getEntries(id)).take(20)
        reply(BlogUpdate(blog))
        //...

      case AddEntry(e, id) =>
        cache += (id -> (e :: cache.getOrElse(id, getEntries(id))))
        sessions.getOrElse(id, Nil).foreach(_ ! BlogUpdate(cache.getOrElse(id, Nil)))

      case DeleteEntry(e, id) => //...
      case EditEntry(e, id) => //..
      case _ =>
    }
}
```

```
class DynamicBlogView extends CometActor {
  //...
  override def localSetup {
    //...
    (BlogCache.cache !? AddBlogWatcher(this, this.blogid)) match {
      case BlogUpdate(entries) => this.blog = entries
    }
  }

  override def lowPriority : PartialFunction[Any, Unit] = {
    case BlogUpdate(entries : List[Entry]) => this.blog = entries; reRender(false)
```

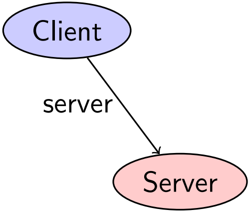
```
(BlogCache.cache !? AddBlogWatcher(this, this.blogid)) match
{ case BlogUpdate(entries) => this.blog = entries}
```

```
        val blog = cache.getOrElse(id, getEntries(id)).take(20)
        reply(BlogUpdate(blog))
        //...

      case AddEntry(e, id) =>
        cache += (id -> (e :: cache.getOrElse(id, getEntries(id))))
        sessions.getOrElse(id, Nil).foreach(_ ! BlogUpdate(cache.getOrElse(id, Nil)))

      case DeleteEntry(e, id) => //...
      case EditEntry(e, id) => //..
      case _ =>
    }
}
```

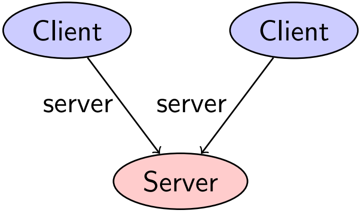
Motivation: Client-Server system (# of clients)



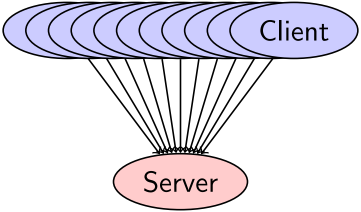
Motivation: Client-Server system (# of clients)



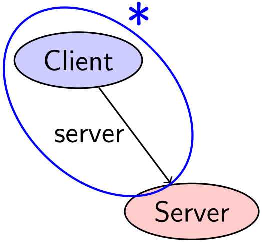
Motivation: Client-Server system (# of clients)



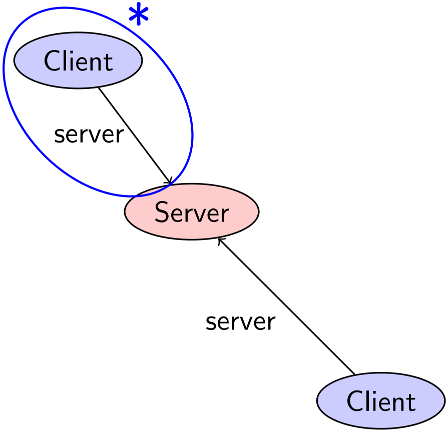
Motivation: Client-Server system (# of clients)

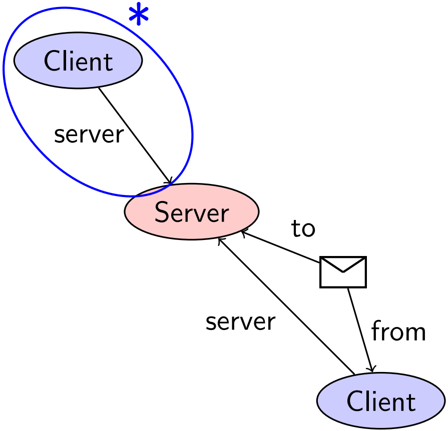


Motivation: Client-Server system (# of clients)

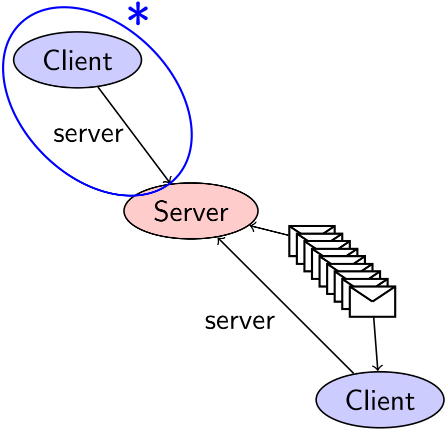


Motivation: Client-Server system (# of clients)

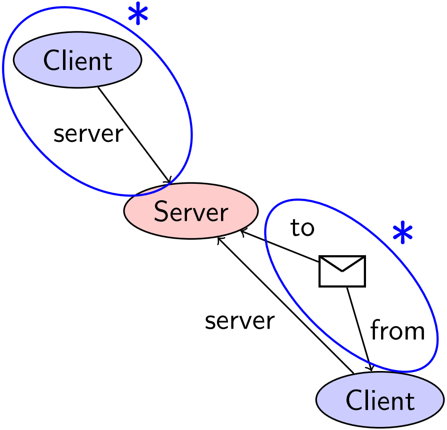



*
Client
server
Server
to
server
from
Client

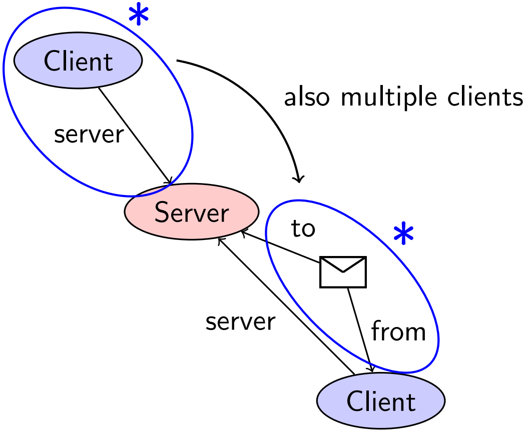
Motivation: Client-Server system (# of clients)



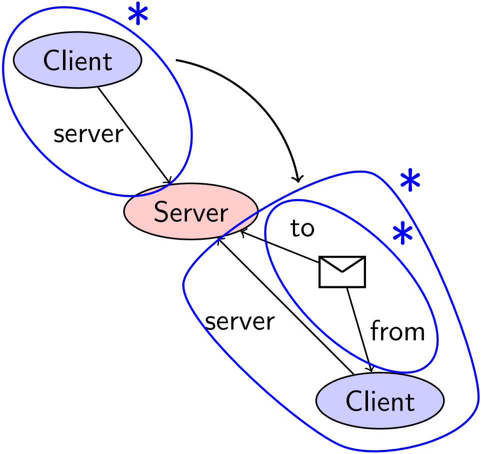
Motivation: Client-Server system (# of clients)



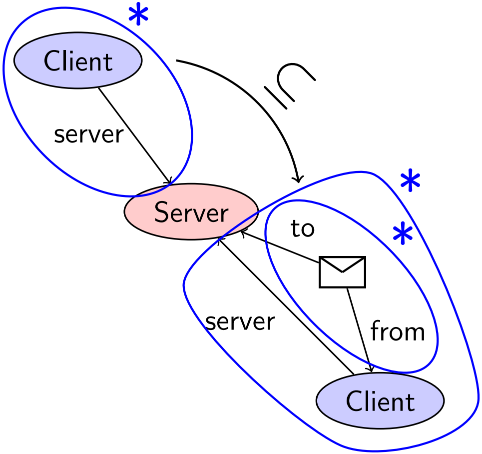
Motivation: Client-Server system (# of clients)

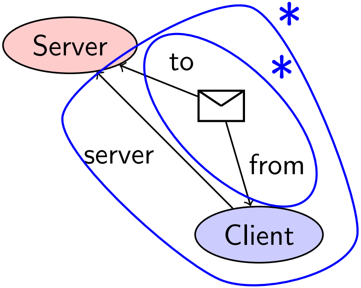


Motivation: Client-Server system (# of clients)

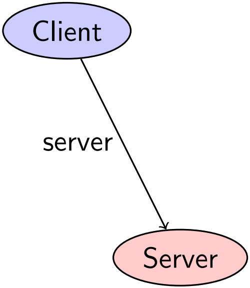


Motivation: Client-Server system (# of clients)

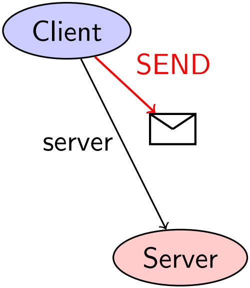



Server
to
*
*
server
from
Client

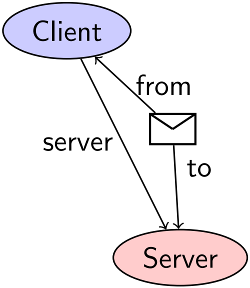
Motivation: Client-Server system (dynamic topology)



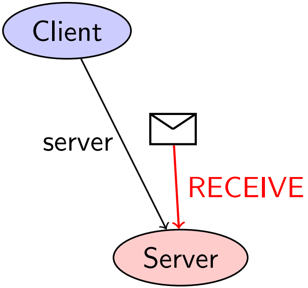
Motivation: Client-Server system (dynamic topology)



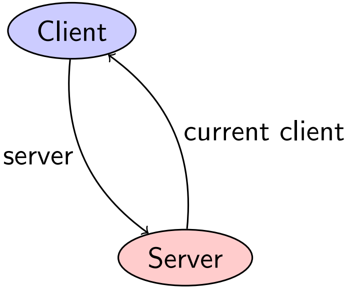
Motivation: Client-Server system (dynamic topology)



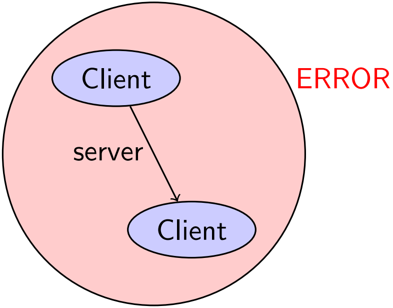
Motivation: Client-Server system (dynamic topology)



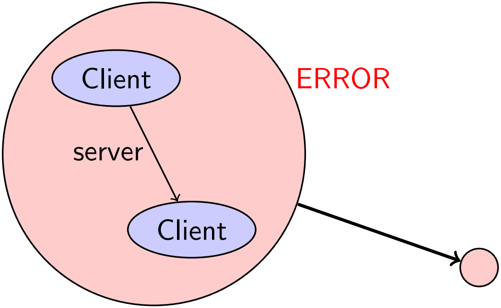
Motivation: Client-Server system (dynamic topology)



Covering problem



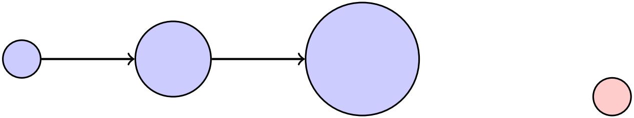
Covering problem



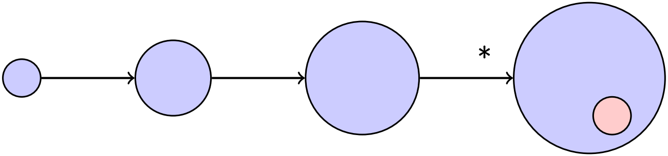
Covering problem

Covering problem

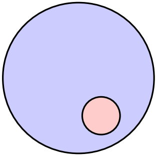
Covering problem



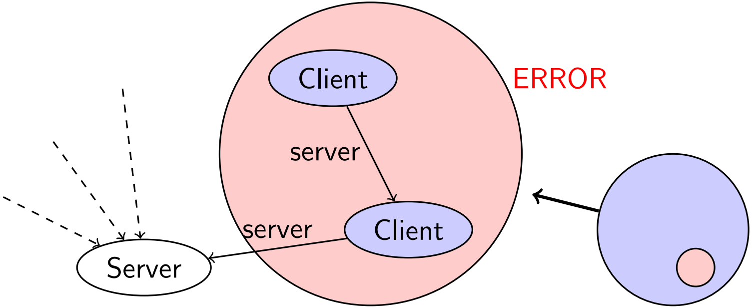
Covering problem



Covering problem



Covering problem

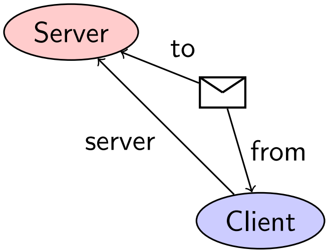


Outline

- π-calculus, depth-bounded systems
- WSTS
- Forward/Backward analysis
- ADL for depth-bounded systems

The π-calculus [Milner et al., 1992a, Milner et al., 1992b] is a process calculus that describes dynamic distributed computations in a message passing-setting.

$$(\nu x)(\mathit{Server}(x) \mid (\nu y)(\mathit{Client}(y,x) \mid \mathit{Messages}(x,y)))$$



π-calculus: Concepts

The π-calculus is build around the notions of

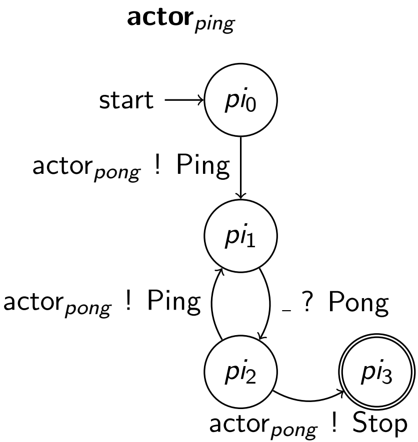
- Names : channels as first class values.
- Threads : concurrent execution of parallel threads: $P \mid Q$.
- i/o prefixes : sending/receiving messages.

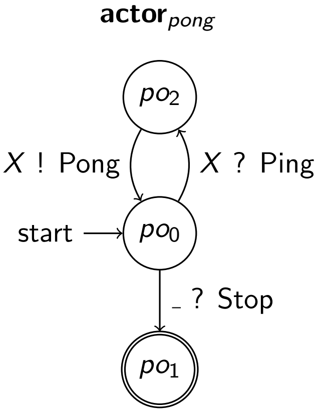
$$
\begin{array}{rcll}
P & ::= & x(y).P & \text{(input prefix)} \\
 & \mid & \overline{x}\langle y\rangle.P & \text{(output prefix)} \\
 & \mid & \sum_i a_i(b_i).P_i & \text{(external choice)} \\
 & \mid & P \mid P & \text{(parallel composition)} \\
 & \mid & !P & \text{(replication)} \\
 & \mid & (\nu x)P & \text{(name creation)} \\
 & \mid & 0 & \text{(unit process)}
\end{array}
$$

```scala
class Ping(count: Int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    loop {
      react {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            println("Ping: stop")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

```scala
class Pong extends Actor {
  def act() {
    var pongCount = 0
    loop {
      react {
        case Ping =>
          if (pongCount % 1000 == 0)
            println("Pong: ping "+pongCount)
          sender ! Pong
          pongCount += 1
        case Stop =>
          println("Pong: stop")
          exit()
      }
    }
  }
}
```

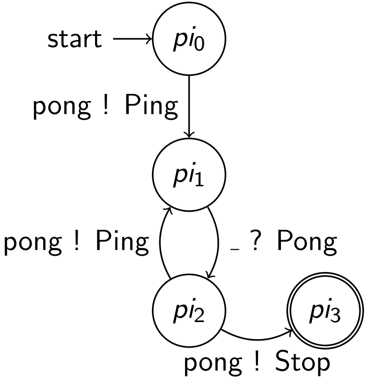
Example (2): scala/docs/examples/actors/pingpong.scala



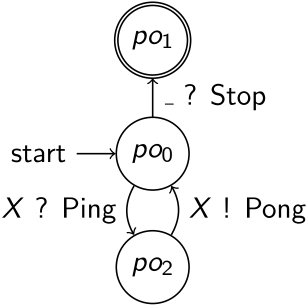


π-calculus: Example (1)

$$
\begin{array}{rcl}
pi_0 & = & \overline{\text{pong}_{Ping}}\langle \text{ping}_{Pong}\rangle | pi_1 \\
pi_1 & = & \text{ping}_{Pong}().pi_2 \\
pi_2 & = & \overline{\text{pong}_{Ping}}\langle \text{ping}_{Pong}\rangle | pi_1 \\
 & \oplus & \overline{\text{pong}_{Stop}}\langle\rangle | pi_3 \\
pi_3 & = & 0
\end{array}
$$



$$
\begin{array}{rcl}
po_0 & = & \text{pong}_{Stop}().po_1 \\
 & + & \text{pong}_{Ping}(X).po_2(X) \\
po_1 & = & 0 \\
po_2(X) & = & \overline{X}\langle\rangle | po_0
\end{array}
$$



Evaluating a formula in π-calculus reduces to applying the rule:

$$\overline{a}\langle b\rangle.P \mid \sum_{i\in I} a_i(b_i).Q_i \quad\rightarrow\quad P \mid Q_x[b/b_x] \quad \text{where } a_x = a$$

What happens:

- channel a carries b;
- b is sent through a and replace b_x in the continuation Q_x.

π-calculus: Semantics

Evaluating a formula in π-calculus reduces to applying the rule:

$$\overline{a}\langle b\rangle.P \mid \sum_{i\in I} a_i(b_i).Q_i \quad\rightarrow\quad P \mid Q_x[b/b_x] \qquad \text{where } a_x = a$$

What happens:

- channel a carries b;
- b is sent through a and replace b_x in the continuation Q_x.

π-calculus: Semantics

Evaluating a formula in π-calculus reduces to applying the rule:

$$\overline{a}\langle b\rangle.P \mid \sum_{i \in I} a_i(b_i).Q_i \quad \rightarrow \quad P \mid Q_x[b/b_x] \qquad \text{where } a_x = a$$

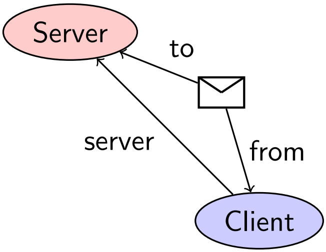
What happens:

- channel a carries b;
- b is sent through a and replace b_x in the continuation Q_x.

Client-Server: communication topology

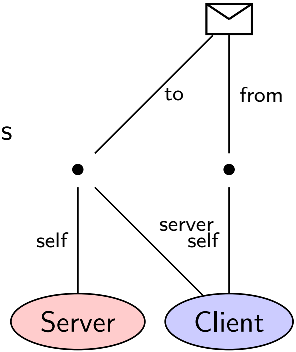
$$(\nu x)(Server(x) \,|\, (\nu y)(Client(y,x) | Messages(x,y)))$$

$$\begin{array}{rcl} Server(self) & = & self(sender).\ldots \\ Client(self, server) & = & self().\ldots \\ Messages(to, from) & = & \overline{to}(from) \end{array}$$

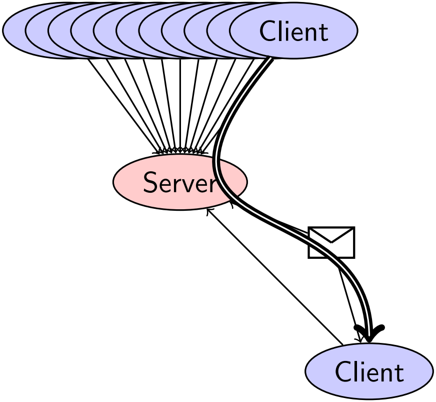


explicit names

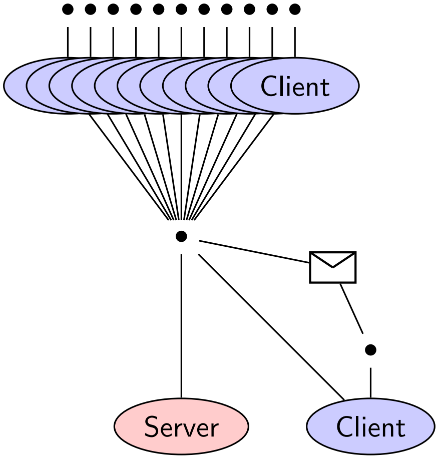
$\longrightarrow$



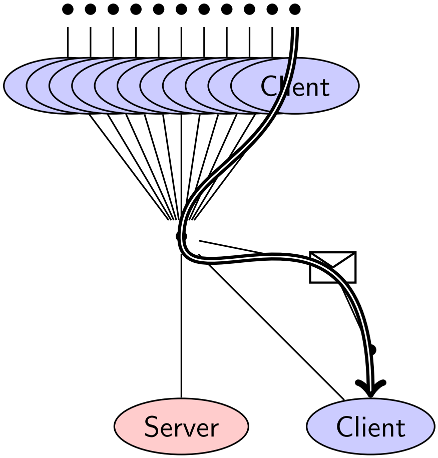
System with a bound on the longest acyclic path.
(Concretely: it is not possible to encode an infinite memory.)



Depth-bounded systems: [Meyer, 2008] (2)



Depth-bounded systems: [Meyer, 2008] (2)



Nesting of names:

$$nest_\nu((\nu x)P) = 1 + nest_\nu(P),$$
$$nest_\nu(P_1 \mid P_2) = \max\{nest_\nu(P_1), nest_\nu(P_2)\},$$
$$\dots$$

The Depth of a configuration:

$$depth(P) = \min\{\, nest_\nu(Q) \mid Q \equiv P \,\}$$

A process $\mathcal{P}$ is *depth-bounded* if:

$$\exists k \in \mathbb{N},\ \forall P \in Reach(\mathcal{P}),\ depth(P) \leq k$$

About Depth-bounded systems

- Depth-bounded systems are well-structured transition systems [Meyer, 2008].
- Reachability is undecidable.
- Termination is decidable.
- Coverability is decidable for system of known depth.
- Coverability for any depth-bounded system was an open problem.

Our contribution:
Coverability is decidable for any depth-bounded system.

Well-structured transition system

A well-structured transition system (WSTS) is a transition system $\langle S, \rightarrow, \leq \rangle$ such that:

- $\leq$ is a well-quasi-ordering (wqo),
  i.e. well-founded + no infinite antichain.
- compatibility of $\leq$ w.r.t. $\rightarrow$

$$\forall \quad \begin{array}{ccc} t & \xrightarrow{*} & t' \\ \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} \\ s & \longrightarrow & s' \end{array} \quad \exists$$

for more detail see:
[Finkel and Schnoebelen, 2001, Abdulla et al., 1996]

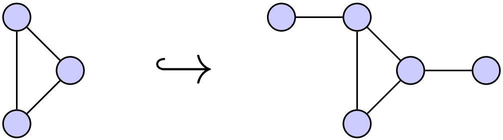
A better-quasi-ordering is a wqo closed under the powerset construction.

$\uparrow x = \{x' \in S \mid x \leq x'\}$ is an upward-closed set.
$\downarrow x = \{x' \in S \mid x' \leq x\}$ is a downward-closed set.

Depth bounded systems as WSTS

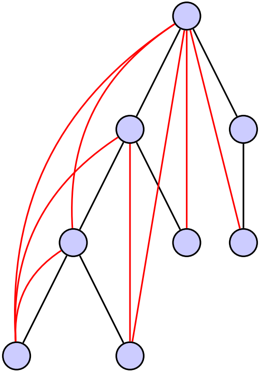
[Meyer, 2008] showed that depth-bounded processes are WSTS for

- their *reachable* configurations and
- the quasi-ordering $\hookrightarrow$ induced by *subgraph isomorphism*.



[Meyer, 2008] showed that $\hookrightarrow$ is a well-quasi-ordering on the reachable configurations.
We show that it is a better-quasi-ordering.

Closure of a tree



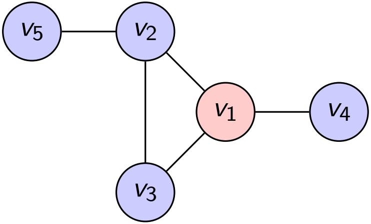
Add edges according to the transitive closure of the parent relation.

Every (undirected) graph is included in the closure of some tree.

Tree-Depth of a graph

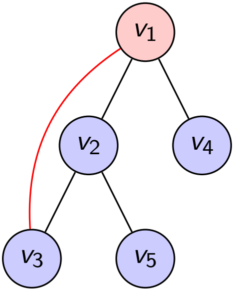
Tree-Depth

The tree-depth td(G) of a graph G is the minimal height of all trees whose closure contains G.



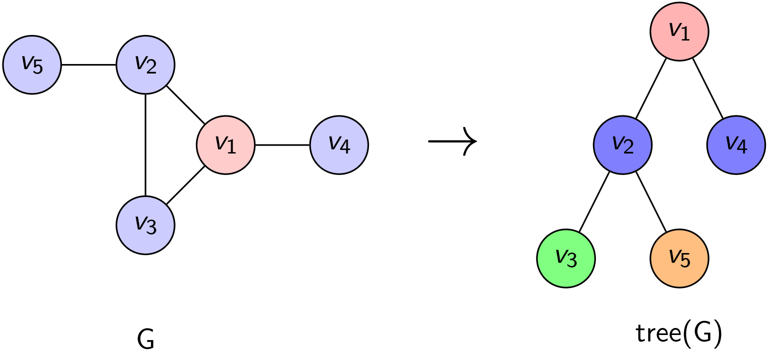
tree-depth $= 2$

$\equiv$



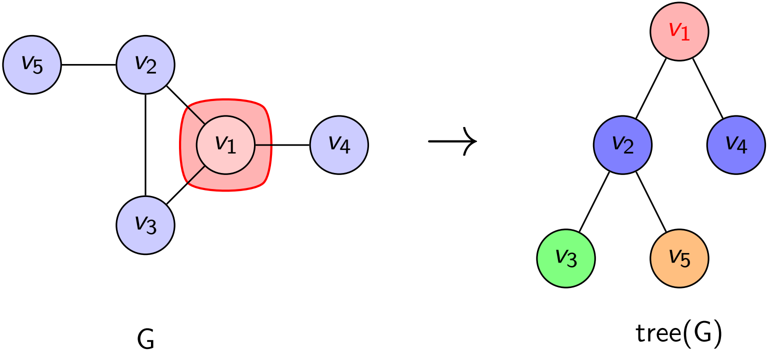
height $= 2$

Tree encoding of depth bounded graph



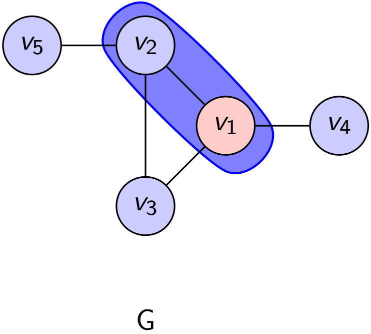
The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Tree encoding of depth bounded graph

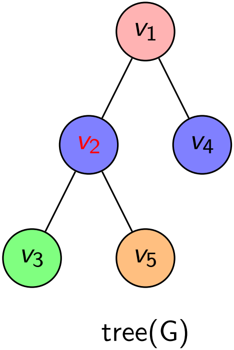


The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Tree encoding of depth bounded graph

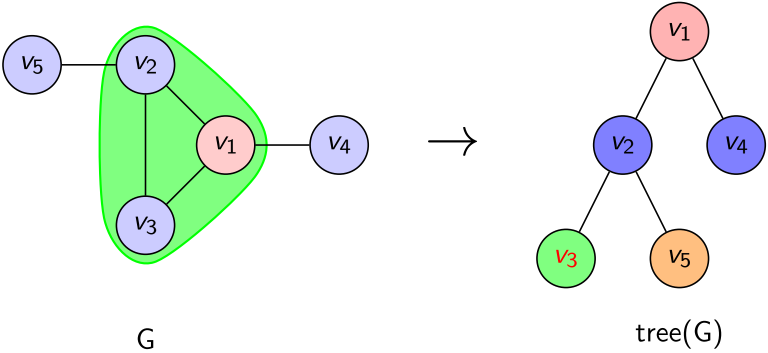


$\rightarrow$



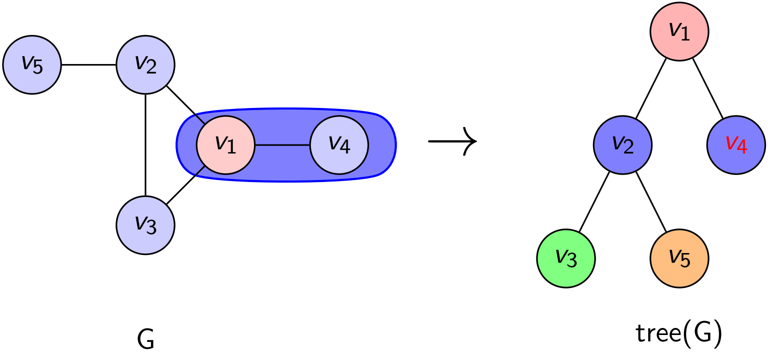
The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Tree encoding of depth bounded graph



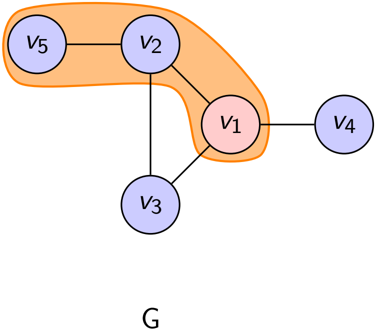
The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Tree encoding of depth bounded graph

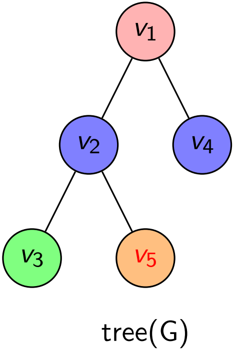


The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Tree encoding of depth bounded graph

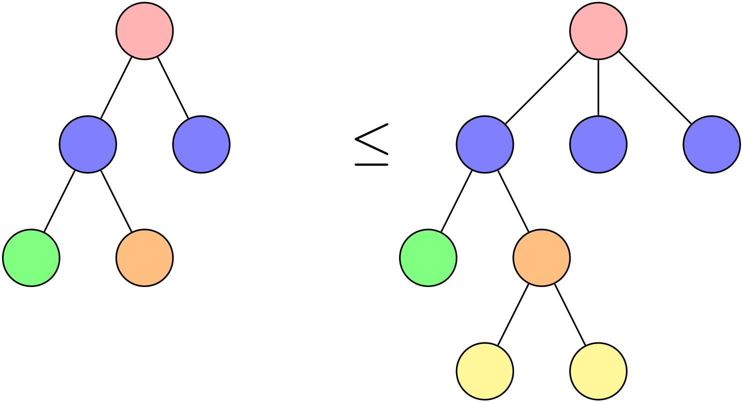


$\rightarrow$



The labels of tree(G) are graphs.
The number of labels used in the encodings is finite.

Homeomorphic tree embedding



We can show for all graphs G_1, G_2:

$$\mathrm{tree}(G_1) \leq \mathrm{tree}(G_2) \text{ implies } G_1 \hookrightarrow G_2$$

Kruskal's tree theorem

Extension of Kruskal's tree theorem [Laver, 1971]

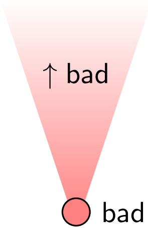
Homeomorphic tree embedding is a better-quasi-ordering on finite trees, where the labels are better-quasi-ordered.

Proposition

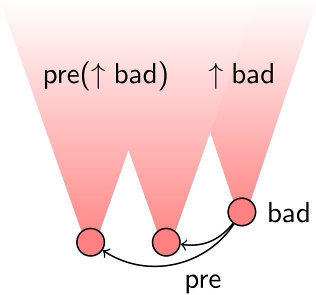
Labelled graphs of bounded tree-depth are better-quasi-orderered by the relation induced by subgraph isomorphisms.

$\Rightarrow$ Subgraph isomorphisms induce a better-quasi-ordering on the reachable configurations of a depth-bounded system.

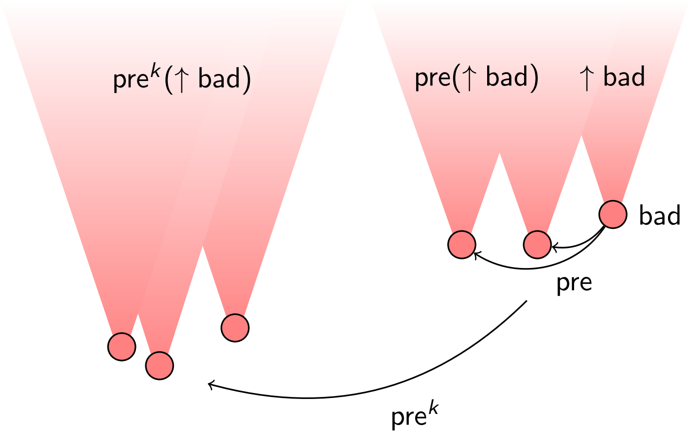
Backward analysis



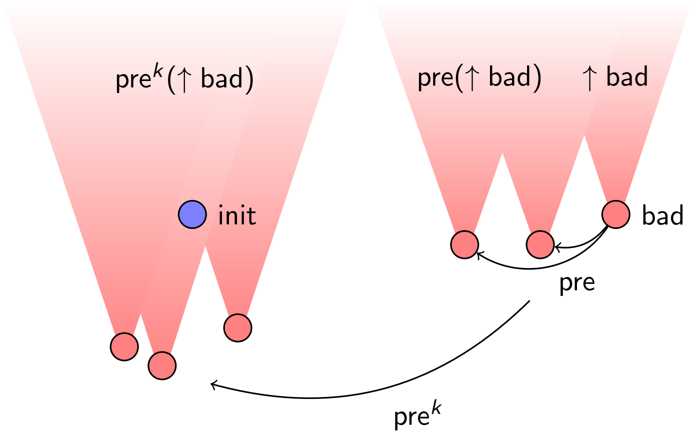
Backward analysis



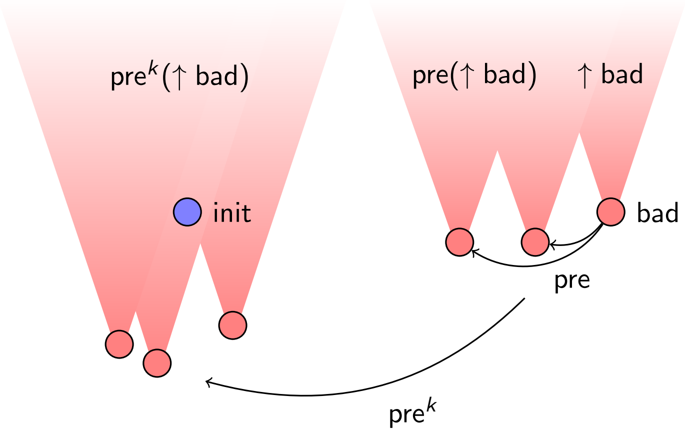
Backward analysis



Backward analysis



Backward analysis



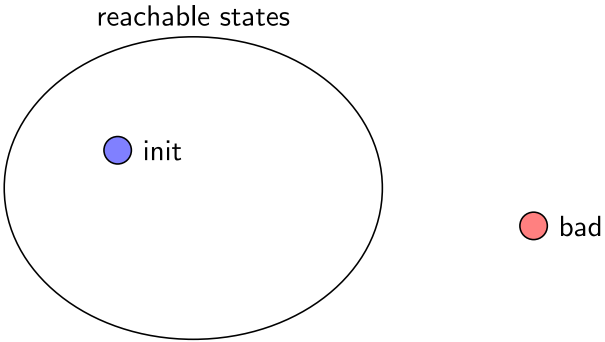
Termination

Let $I_n = \bigcup_{i=0}^{n} pre^i(\mathrm{bad})$ then $I_0 \subseteq I_1 \subseteq \ldots \subseteq I_n \subseteq I_{n+1} \ldots$
This sequence stabilizes because a wqo has no infinite antichain.

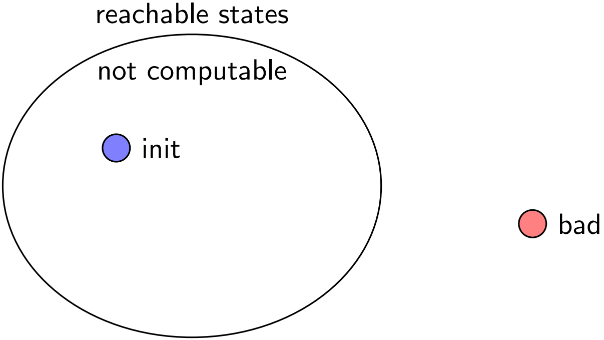
Analysis of depth-bounded systems: Backward analysis

- Backward analysis requires *pre* to be computable.
- The WSTS of a depth-bounded system is defined wrt. the *forward-reachable* configurations.
- pre generates unreachable configurations.
- The set of reachable configurations is not computable
- We need to known the depth to preserve the wqo.
- Backward algorithms for coverability works only with processes of *known depth*.

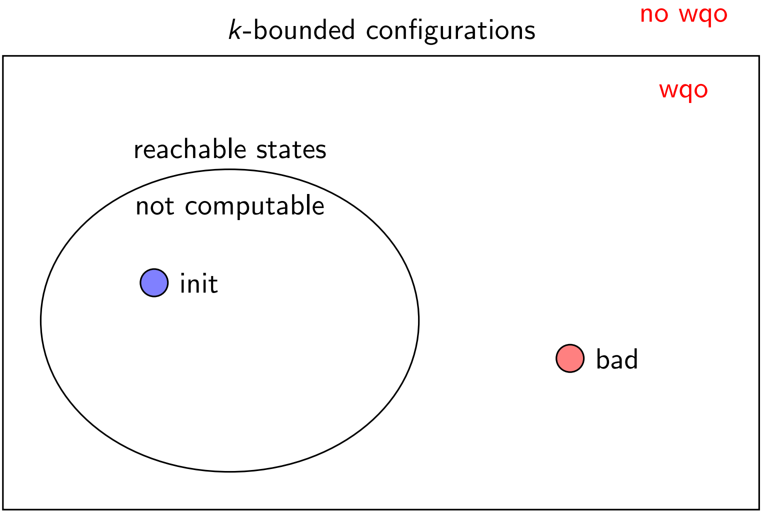
Analysis of depth-bounded systems: computing *pre*



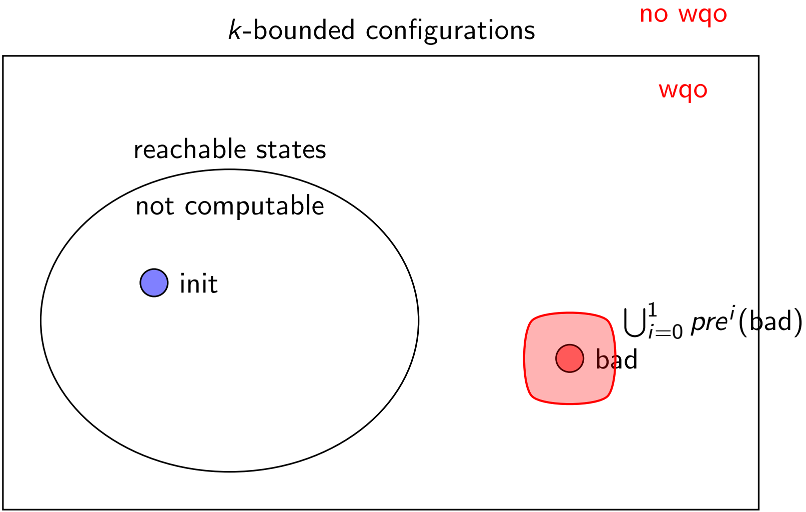
Analysis of depth-bounded systems: computing *pre*



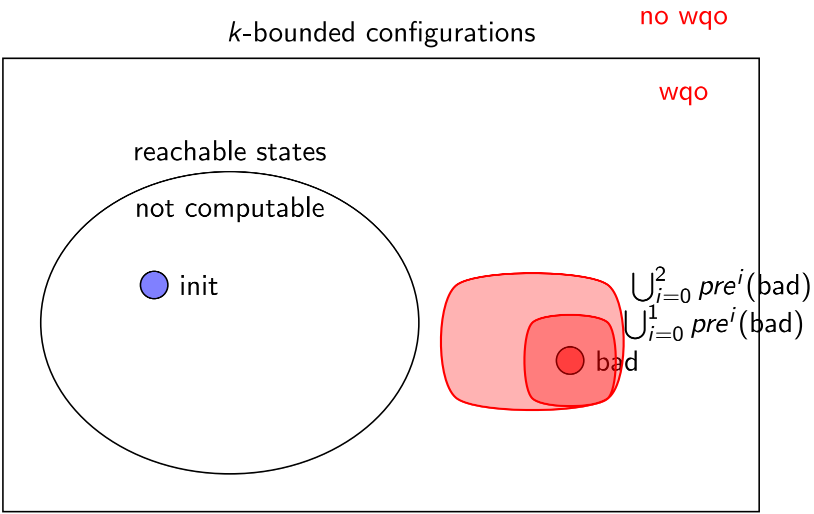
Analysis of depth-bounded systems: computing *pre*

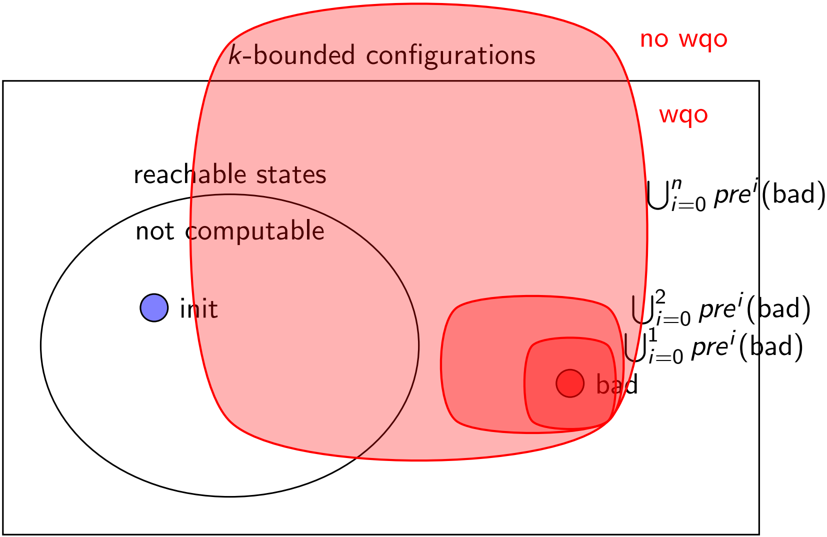


Analysis of depth-bounded systems: computing *pre*



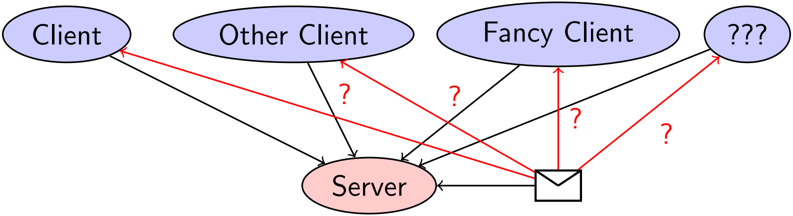
Analysis of depth-bounded systems: computing *pre*



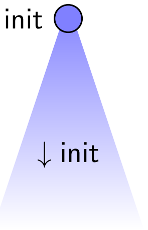


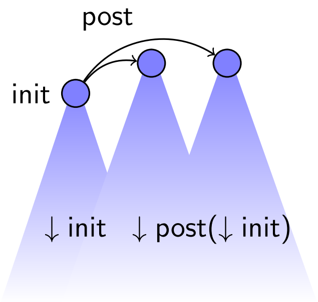
no wqo
k-bounded configurations
wqo
reachable states
not computable
$\bigcup_{i=0}^{n} pre^i(\text{bad})$
init
$\bigcup_{i=0}^{2} pre^i(\text{bad})$
$\bigcup_{i=0}^{1} pre^i(\text{bad})$
bad

Backward analysis has to guess the exchanged names of each reduction step.
$\rightarrow$ explosion in the nondeterminism.

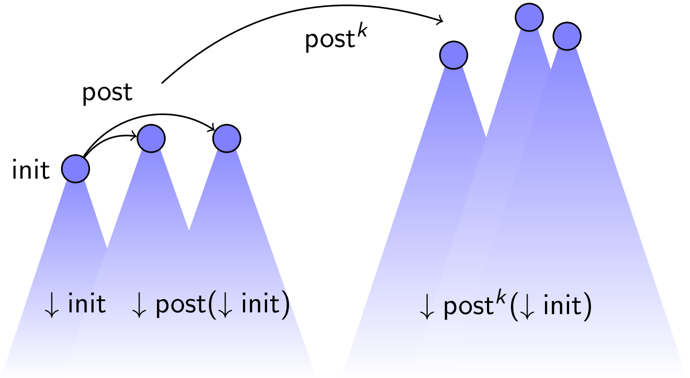


Forward analysis

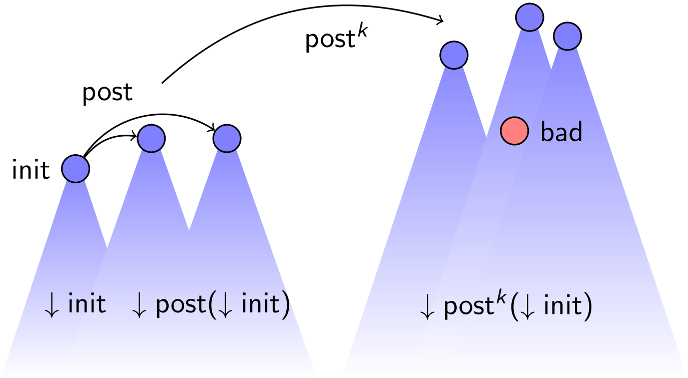



post
init
↓ init
↓ post(↓ init)

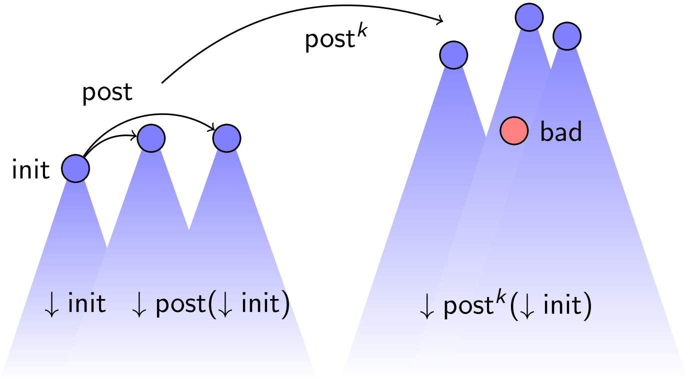
Forward analysis



Forward analysis



Forward analysis



Problem

How to represents downward-closed sets ?

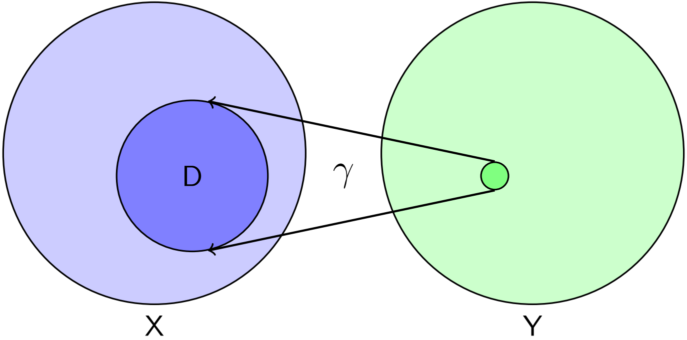
Analysis of depth-bounded systems: Forward analysis

- Forward algorithms terminate even if the *bound is not known*.
- The algorithm is an instance of the expand enlarge check algorithm [Geeraerts et al., 2006] that uses adequate domain of limits (ADL).
- [Finkel and Goubault-Larrecq, 2009b] provides a theoretical framework for the manipulation of downward-closed sets and the construction of ADL.
- We build such an ADL by extending configurations with '!'.

$\Rightarrow$ coverability is decidable for the entire class of depth-bounded systems.

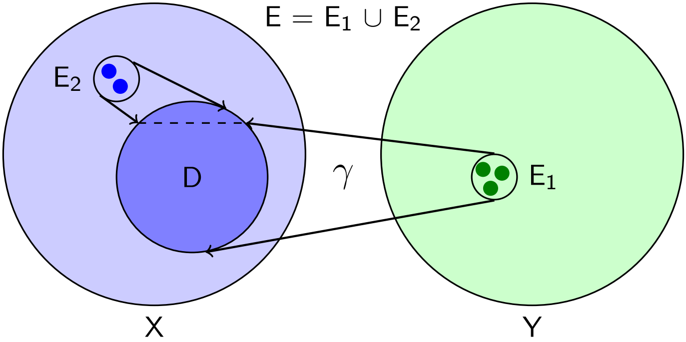
ADL: [Geeraerts et al., 2006]
let Y an ADL for wqo set X:

For every $z \in \mathrm{X} \cup \mathrm{Y}, \gamma(z)$ is a downward-closed subset of X.

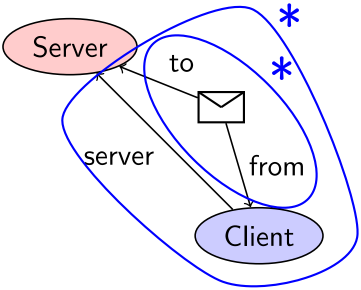


Adequate Domain of Limit (2)

Every downward-closed subset D of X is generated by a finite subset E of $Y \cup X$.



$$(\nu x)(Server(x) \mid !(\nu y)(Client(y, x) | !Messages(x, y)))$$



Limits configuration for depth-bounded systems

We use '!' not as a recursion operator but as a mean to represent infinite sets of configurations.

$\mathcal{C}(PI, k)$ is the set of configurations.
$\mathcal{L}(PI, k)$ in the set of limit configurations.

Theorem

Let $k \in \mathbb{N}$ and let PI be a finite set of process identifiers. Then $(\mathcal{L}(PI, k), \sqsubseteq, \gamma)$ is a weak adequate domain of limits for the well-quasi-ordered set $(\mathcal{C}(PI, k), \leq)$.

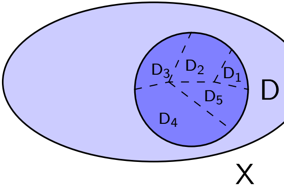
Corollary

Coverability is decidable for the entire class of depth-bounded systems.

Limits configuration for depth-bounded systems

Theorem [Finkel and Goubault-Larrecq, 2009b]

The downward-closed directed subsets of a wqo set X form an ADL for X.

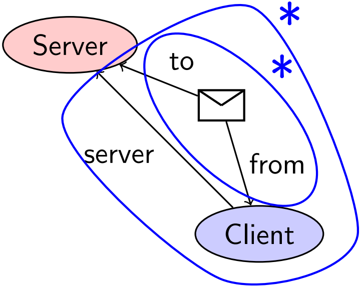


Proposition

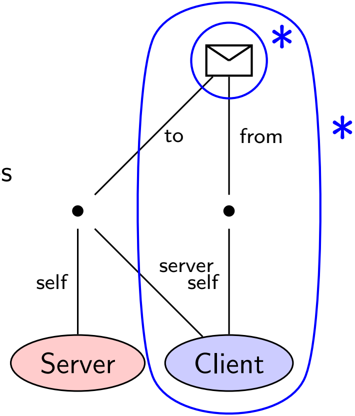
The directed downward-closed sets of depth-bounded configurations are exactly the denotations of limit configurations.

We characterize the tree encodings of downward-closed sets of configurations in terms of the languages of *hedge automata*.

$$(\nu x)(\mathit{Server}(x) \mid !(\nu y)(\mathit{Client}(y,x) | !\mathit{Messages}(x,y)))$$

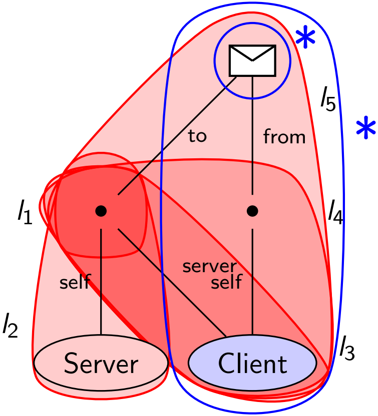


explicit names
$\longrightarrow$

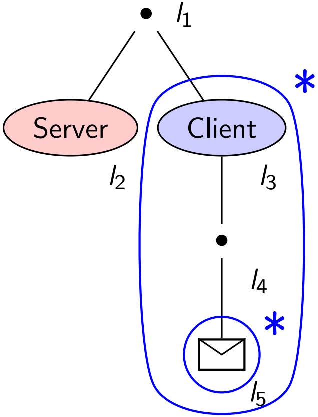


Regular language of unranked trees for Client-Server (2)

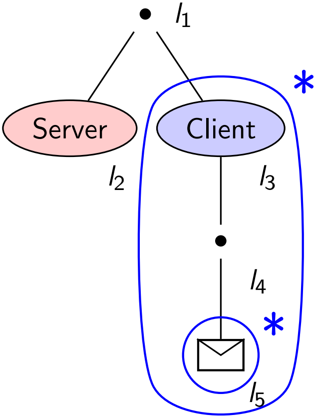
$$(\nu x)(Server(x) \mid !(\nu y)(Client(y,x) | !Messages(x,y)))$$



as tree
$\longrightarrow$



$$(\nu x)(Server(x) \mid !(\nu y)(Client(y,x) | !Messages(x,y)))$$



tree language
$\longrightarrow$

$$l_1(q_1 q_2^*) \to q_{final}$$
$$l_2(\epsilon) \to q_1$$
$$l_3(q_3) \to q_2$$
$$l_4(q_4^*) \to q_3$$
$$l_5(\epsilon) \to q_4$$

Further Work

We started an implemention to compute (an over-approximation of) the cover using [Finkel and Goubault-Larrecq, 2009a].

```
Equations:
----------------------------------------
client1(A, B) = (A().(client1(A, B) |
                     request1(B, A)))
answer1(A) = (A<>.0)
new1(A) = (A<>.0)
request1(A, B) = (A<B>.0)
server(A, B) = (A(C).(answer1(C) |
                       server(A, B)) +
                B().(ny D)
                      (client1(D, A) |
                       answer1(D) |
                       new1(B) |
                       server(A, B)))

Initial configuration:
----------------------------------------
(ny A, B)
    (new1(A) |
     server(B, A))
```

```
Computed cover:

(ny A, B)
    (!((ny C)
         (answer1(C) |
          client1(C, B))) |
    !((ny D)
         (client1(D, B) |
          request1(B, D))) |
    new1(A) |
    server(B, A))
```

Recap

Coverability is decidable for depth-bounded processes.

- We provide an ADL for depth-bounded processes;
- prepared the ground for a spectrum of forward algorithms for depth-bounded processes.

Questions ?

Abdulla, P. A., Cerans, K., Jonsson, B. and Tsay, Y.-K. (1996).
General Decidability Theorems for Infinite-State Systems.
In LICS pp. 313–321,.

Finkel, A. and Goubault-Larrecq, J. (2009a).
Forward Analysis for WSTS, Part II: Complete WSTS.
In ICALP (2) pp. 188–199,.

Finkel, A. and Goubault-Larrecq, J. (2009b).
Forward Analysis for WSTS, Part I: Completions.
In STACS vol. 09001, of Dagstuhl Sem. Proc. pp. 433–444,.

Finkel, A. and Schnoebelen, P. (2001).
Well-structured transition systems everywhere!
Theor. Comput. Sci. *256*, 63–92.

Geeraerts, G., Raskin, J.-F. and Van Begin, L. (2006).
Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS.
J. Comput. Syst. Sci. *72*, 180–203.

Laver, R. (1971).
On Fraïssé's Order Type Conjecture.
Ann. of Math. *93*, 89–111.

Meyer, R. (2008).
On Boundedness in Depth in the pi-Calculus.
In IFIP TCS vol. 273, of IFIP pp. 477–489, Springer.

References II

Milner, R., Parrow, J. and Walker, D. (1992a).
A Calculus of Mobile Processes, I.
Inf. Comput. *100*, 1–40.

Milner, R., Parrow, J. and Walker, D. (1992b).
A Calculus of Mobile Processes, II.
Inf. Comput. *100*, 41–77.