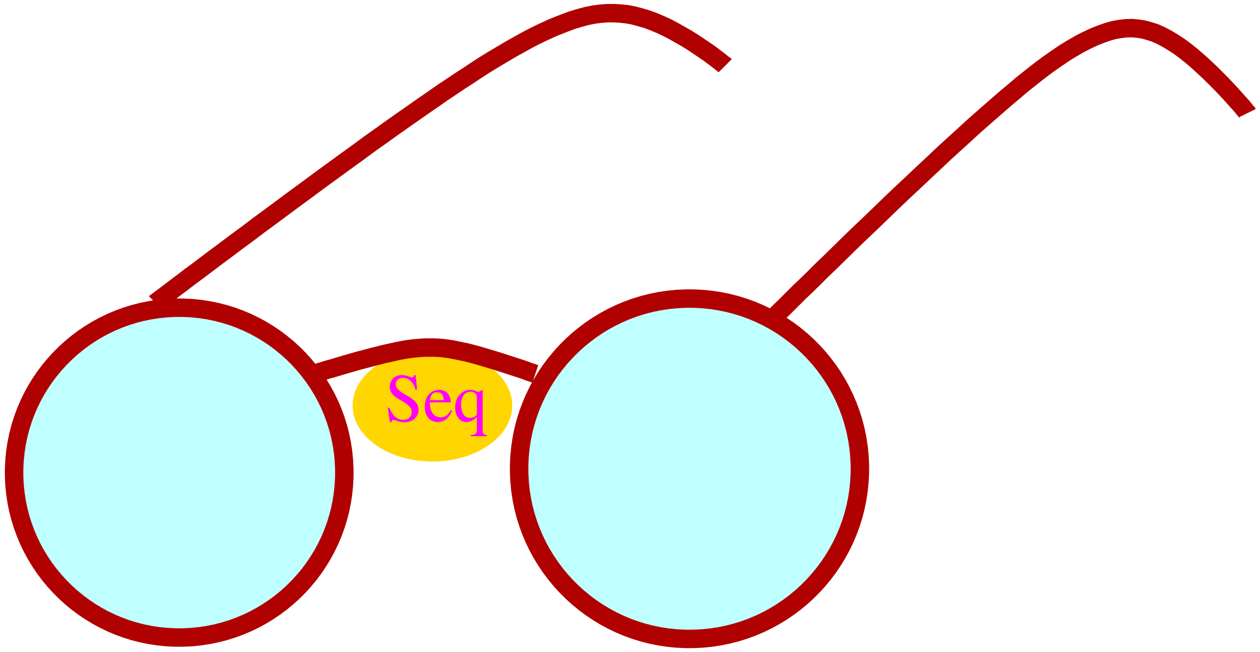
Grammars as Processes

An automata-theoretic approach to program analysis

Javier Esparza

Institut für Informatik
Technische Universität München

A different way of looking at grammars ...



Linear Rules

$$X \longrightarrow aY$$

Put your glasses on ...

$$X \xrightarrow{a} Y$$

Process X can do an action a and become process Y.

$$X \longrightarrow a$$

Put your glasses on ...

$$X \xrightarrow{a} \varepsilon$$

Process X can do an action a and terminate.

Context-free Rules

$$X \longrightarrow aYZ$$

Put your glasses on ...

$$X \xrightarrow{a} Y \cdot Z$$

Process X can do an action a and become the sequential composition of the processes Y and Z.

This choice leads to prefix rewriting as the right way of applying the rules.

Without glasses you see

$$\frac{X \longrightarrow w}{uXv \longrightarrow uwv}$$

and with them you see

$$\frac{X \xrightarrow{a} w}{X \cdot v \xrightarrow{a} w \cdot v}$$

General Rules

$$XY \longrightarrow aZ$$

If you put your glasses on, you see

$$X \cdot Y \xrightarrow{a} Z$$

Canbe used to model value passing along sequential composition.

Problem: model $X \cdot Y$, where

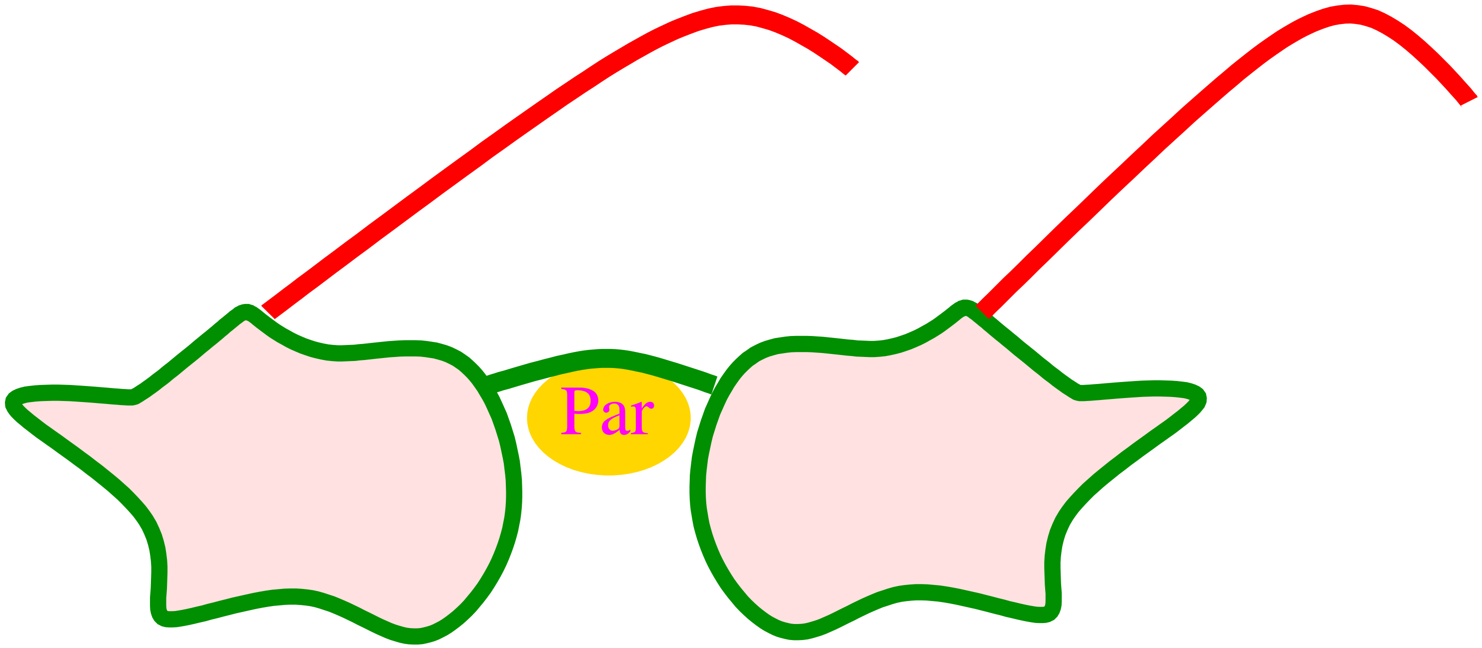
- X computes a boolean (somehow), and
- Y sends this boolean to the environment.

Solution:

$$\begin{array}{rcl} X & \longrightarrow & T \\ X & \longrightarrow & F \\ T \cdot Y & \xrightarrow{t} & \varepsilon \\ F \cdot Y & \xrightarrow{f} & \varepsilon \end{array}$$

$$X \cdot Y \xrightarrow{t} T \cdot Y \xrightarrow{t} \varepsilon$$
$$X \cdot Y \xrightarrow{t} F \cdot Y \xrightarrow{f} \varepsilon$$

Parallel composition: Concurrency glasses



Context-free Rules

$$X \longrightarrow aYZ$$

Put your concurrency glasses on ...

$$X \xrightarrow{a} Y \| Z$$

Process X can do an action a and become the parallel composition of the processes Y and Z.

General Rules

$$XY \longrightarrow aZW$$

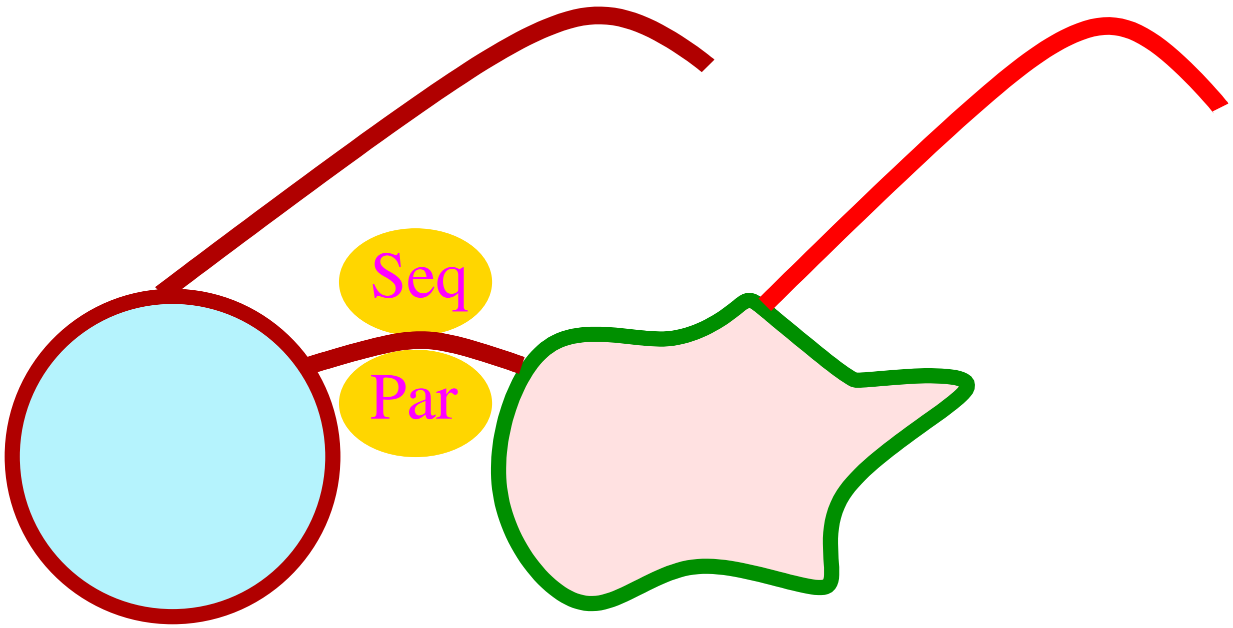
Put yout concurrency glasses on ...

$$X \| Y \xrightarrow{a} Z \| W$$

Can be used to model synchronisation of parallel processes.

Processes X and Y can synchronize on a and become processes Z and W.

And putting everything together ...



$$X \cdot Y \xrightarrow{a} Z \cdot (W \| V)$$

Process Rewrite Systems

Process terms or states have the following syntax:

$$t ::= \varepsilon \mid X \mid t_1 \cdot t_2 \mid t_1 \| t_2$$

where X is a process variable.
Syntactic identifications:

$$\begin{array}{rclcrcl} t & \equiv & \varepsilon \cdot t & \quad & t & \equiv & t \| \varepsilon \\ t & \equiv & t \cdot \varepsilon & \quad & t & \equiv & \varepsilon \| t \end{array}$$

A process rewrite system is a tuple (Var, Act, Δ), where

- $Var = \{X, Y, Z, \ldots\}$ is a set of process variables;
- $Act = \{a, b, c, \ldots\}$ is a set of actions;
- Δ is a set of rewrite rules of the form $t \xrightarrow{a} t'$.

Together with the following inference rules, Δ induces a (possibly infinite!) transition system:

$$\mathbf{\Delta} \quad \frac{(t \xrightarrow{a} t') \in \Delta}{t \xrightarrow{a} t'}$$

$$\mathbf{seq} \quad \frac{t \xrightarrow{a} t'}{t \cdot u \xrightarrow{a} t' \cdot u}$$

$$\mathbf{par1} \quad \frac{t \xrightarrow{a} t'}{t \parallel u \xrightarrow{a} t' \parallel u}$$

$$\mathbf{par2} \quad \frac{u \xrightarrow{a} u'}{t \parallel u \xrightarrow{a} t \parallel u'}$$

An example with sequential composition

$$
\begin{aligned}
A\cdot B &\xrightarrow{a} A\cdot B\cdot B\\
A\cdot B &\xrightarrow{b} C\\
A\cdot B &\xrightarrow{c} D\\
C\cdot B &\xrightarrow{d} C\\
D\cdot B &\xrightarrow{d} E\cdot B\\
E\cdot B &\xrightarrow{d} D
\end{aligned}
$$

$$
\begin{tikzcd}
D & EB \arrow[l, "d"'] & DB \arrow[l, "d"'] & EBB \arrow[l, "d"'] & DBB \arrow[l, "d"'] & \cdots \arrow[l] \\
AB \arrow[u, "c"] \arrow[d, "b"'] \arrow[rr, "a"] & & ABB \arrow[u, "c"] \arrow[d, "b"'] \arrow[rr, "a"] & & ABBB \arrow[u, "c"] \arrow[d, "b"'] & \cdots \arrow[l] \\
C & & CB \arrow[ll, "d"'] & & CBB \arrow[ll, "d"'] & \cdots \arrow[l]
\end{tikzcd}
$$

An example with parallel composition

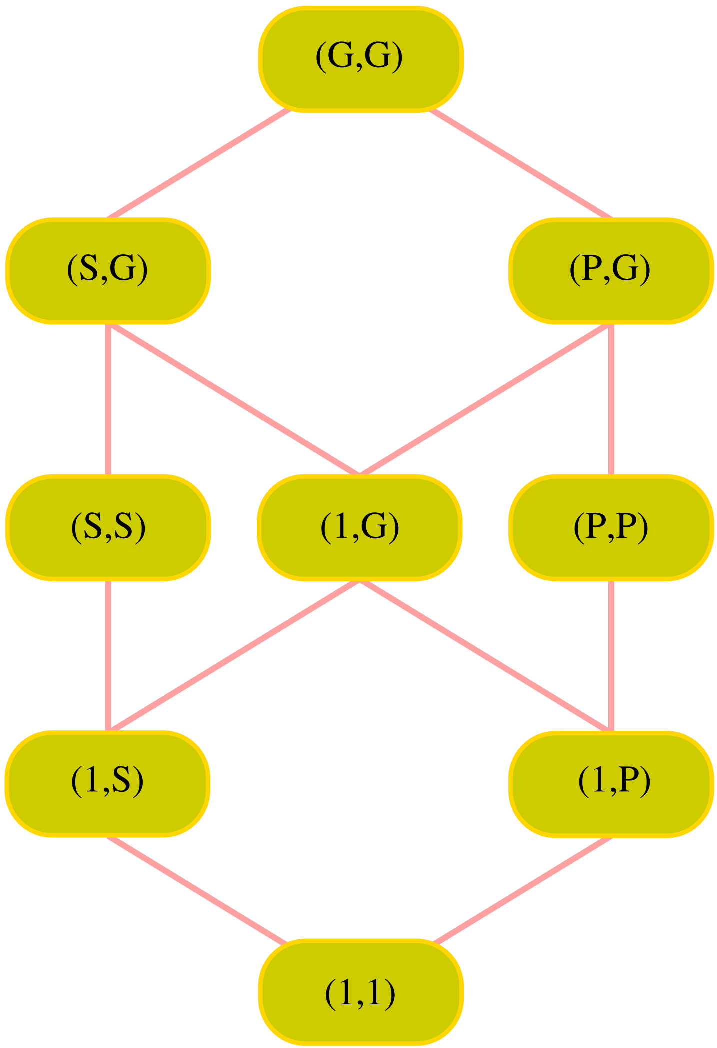
$$
\begin{aligned}
R &\xrightarrow{a} A \\
R &\xrightarrow{b} B \\
A &\xrightarrow{a} A \parallel A \\
A &\xrightarrow{b} A \parallel B \\
B &\xrightarrow{a} A \parallel B
\end{aligned}
$$

$$
\begin{tikzcd}
R \arrow[r, "b"] \arrow[d, "a"'] & B \arrow[r, "b"] \arrow[d, "a"'] & B \parallel B \arrow[r, "b"] \arrow[d, "a"'] & B \parallel B \parallel B \\
A \arrow[r, "b"] \arrow[d, "a"'] & A \parallel B \arrow[r, "b"] \arrow[d, "a"'] & A \parallel B \parallel B & \cdots \\
A \parallel A \arrow[r, "b"] \arrow[d, "a"'] & A \parallel A \parallel B & \cdots & \\
A \parallel A \parallel A & \cdots & &
\end{tikzcd}
$$

A Chomsky-like process hierarchy

Types of terms:

- **1** single variables
- **P** only parallel composition
- **S** only sequential composition
- **G** general terms



Is the process-hierarchy interesting?

The Chomsky-hierarchy is interesting because (a.o.):

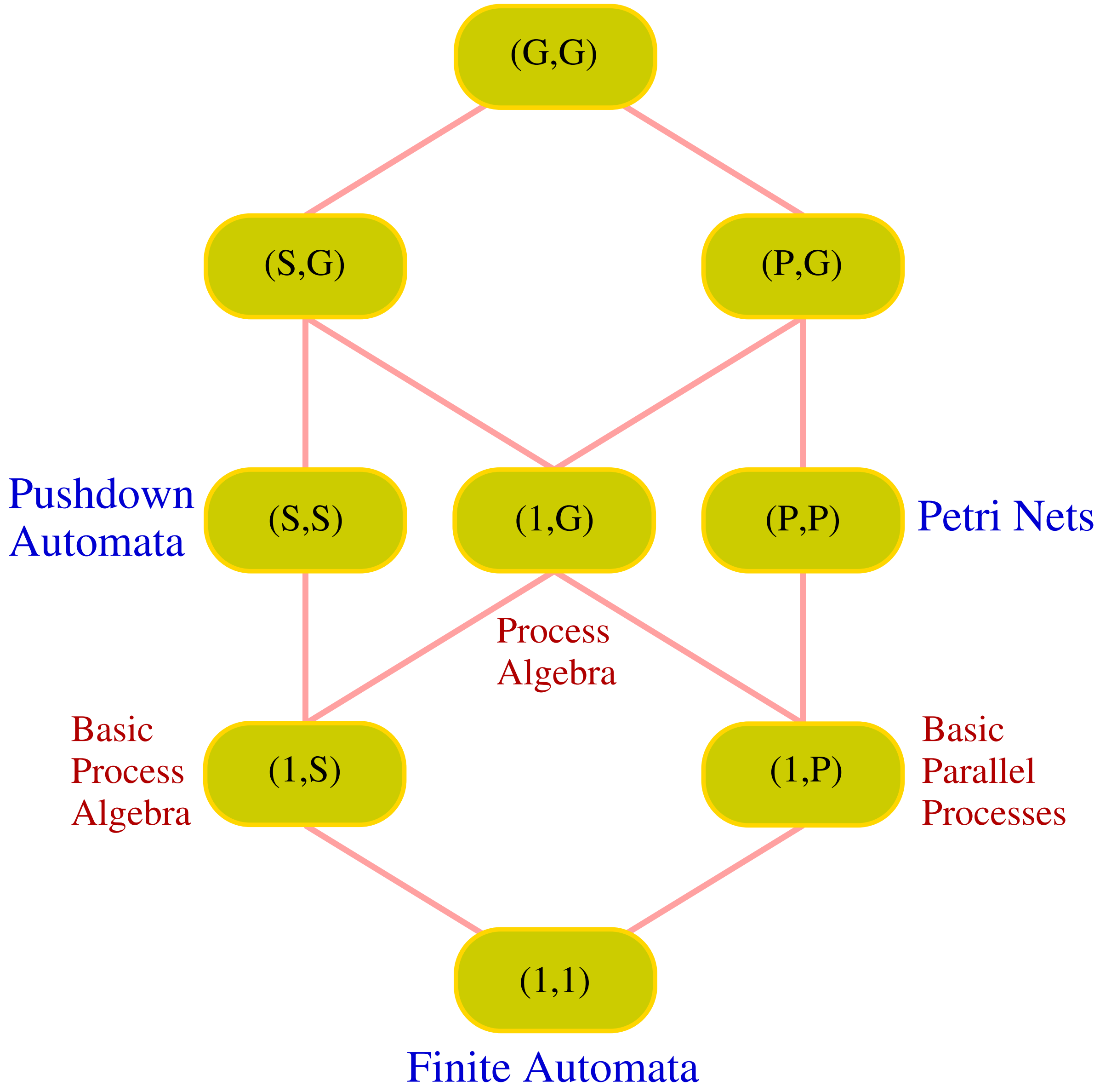
- it captures fundamental phenomena (e.g. computability),
- its captures language classes recognised by natural machines,
- it provides a formal framework for lexicographic and syntactic analysis of programming languages.

The process hierarchy is interesting because (a.o.):

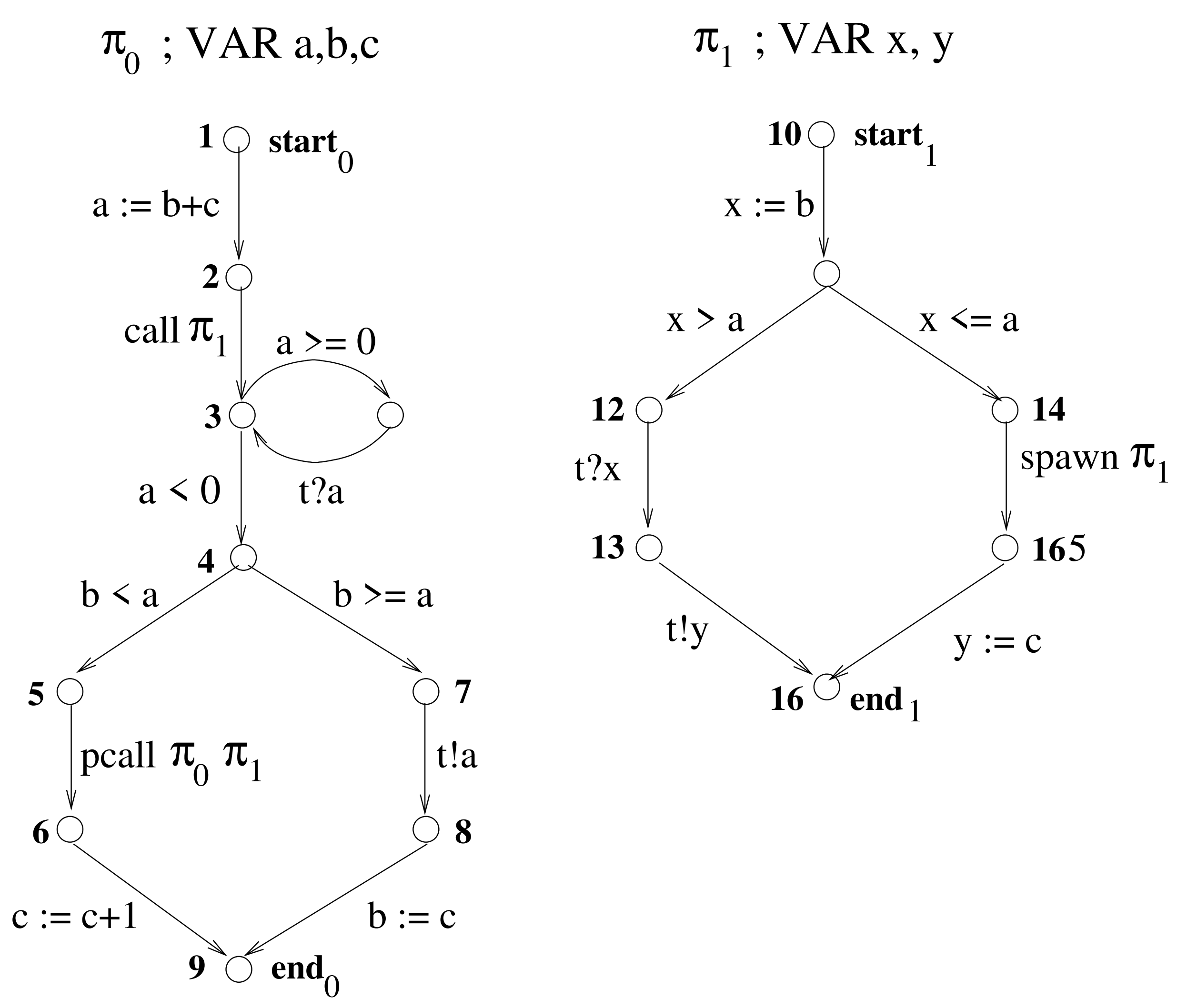
- it captures fundamental phenomena (e.g. sequential and parallel composition, recursion, synchronisation …),
- it captures classes of transition systems recognised by natural machines,
- it provides a formal framework for dataflow analysis of programming languages.

Machine Characterizations

Devices and formalisms generating the transition systems of the process hierarchy (up to bisimulation):



Flow-graphs



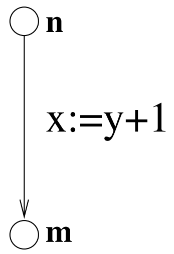
Semantics: the set of program paths.

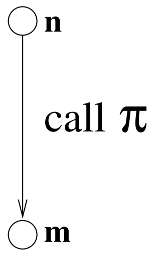
Program path: path with correct nesting of calls and returns, and correct interleaving of parallel executions.

A control-flow hierarchy

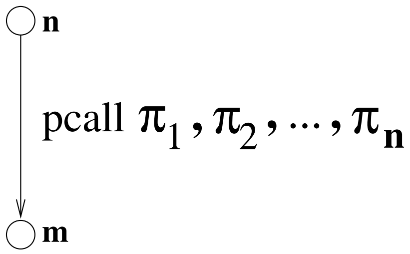
- “plain programs”
  (while programs, if-goto programs)
- \+ procedures
- \+ parallel threads (parbegin-parend)
- \+ thread synchronization (rendezvous)

PRS semantics of flow graph systems

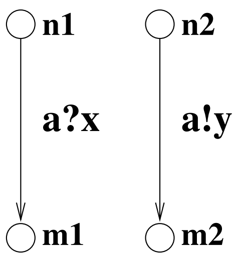




$$N \xrightarrow{x:=y+1} M \qquad N \xrightarrow{call\,\pi} START \cdot M$$

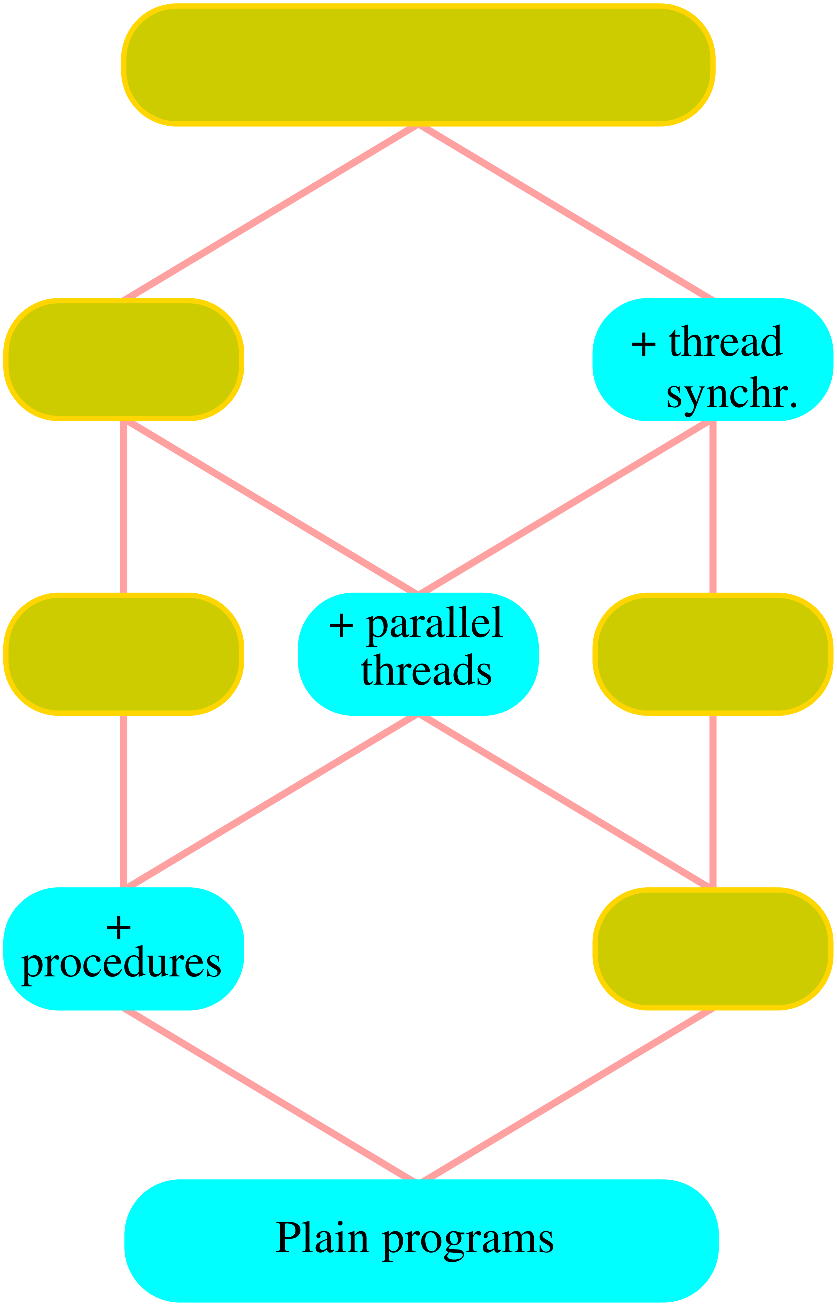


$$N \xrightarrow{pcall\ \pi_1,\ldots,\pi_n} (START_1 \parallel \ldots \parallel START_n) \cdot M$$



$$N_1 \parallel N_2 \xrightarrow{a(x,y)} M_1 \parallel M_2$$

The control-flow hierarchy



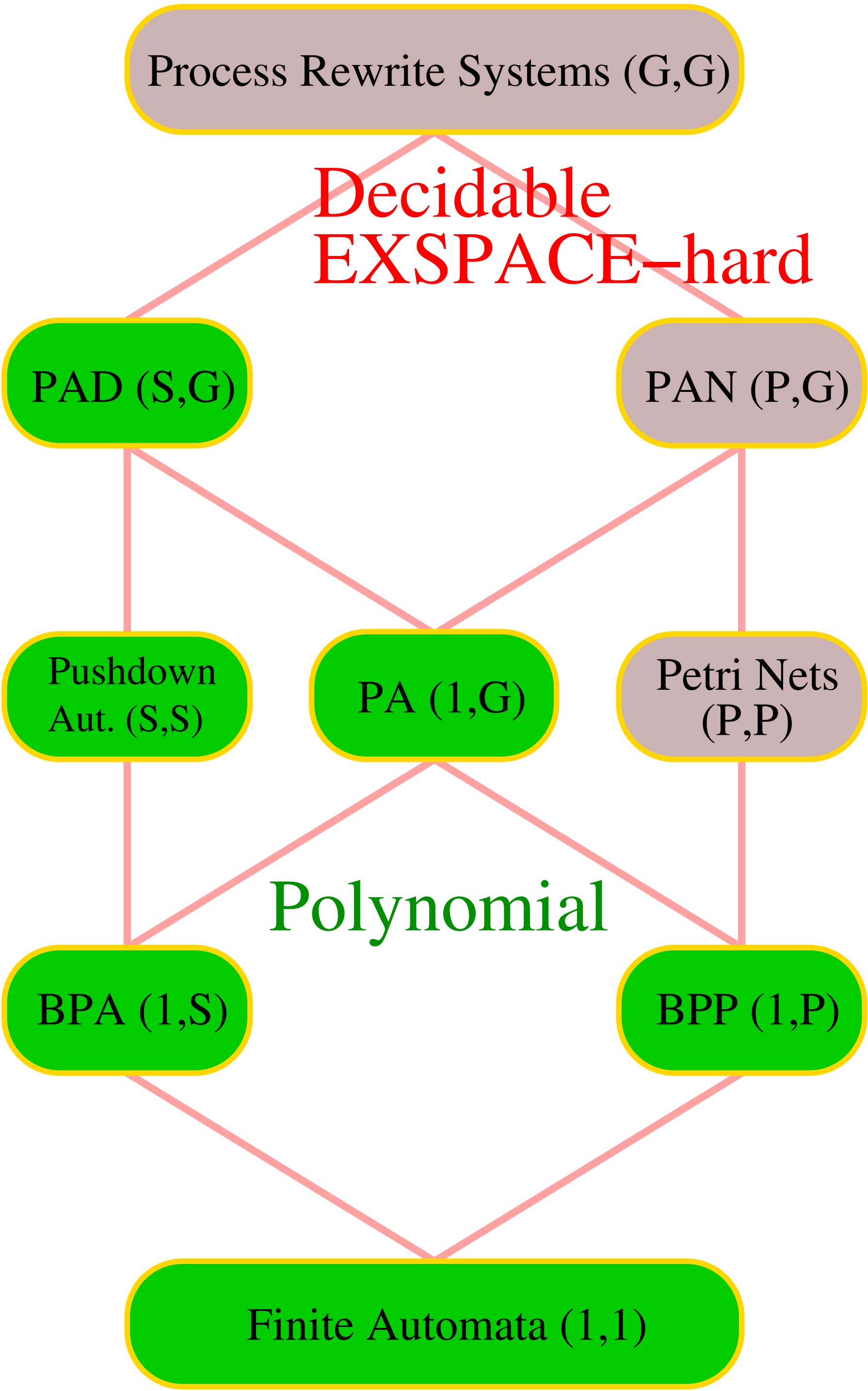
Studying the control-flow hierarchy

- Equivalence: do two given flow-graphs generate the same behaviours?
  (see e.g. Moller's survey in CONCUR '96)

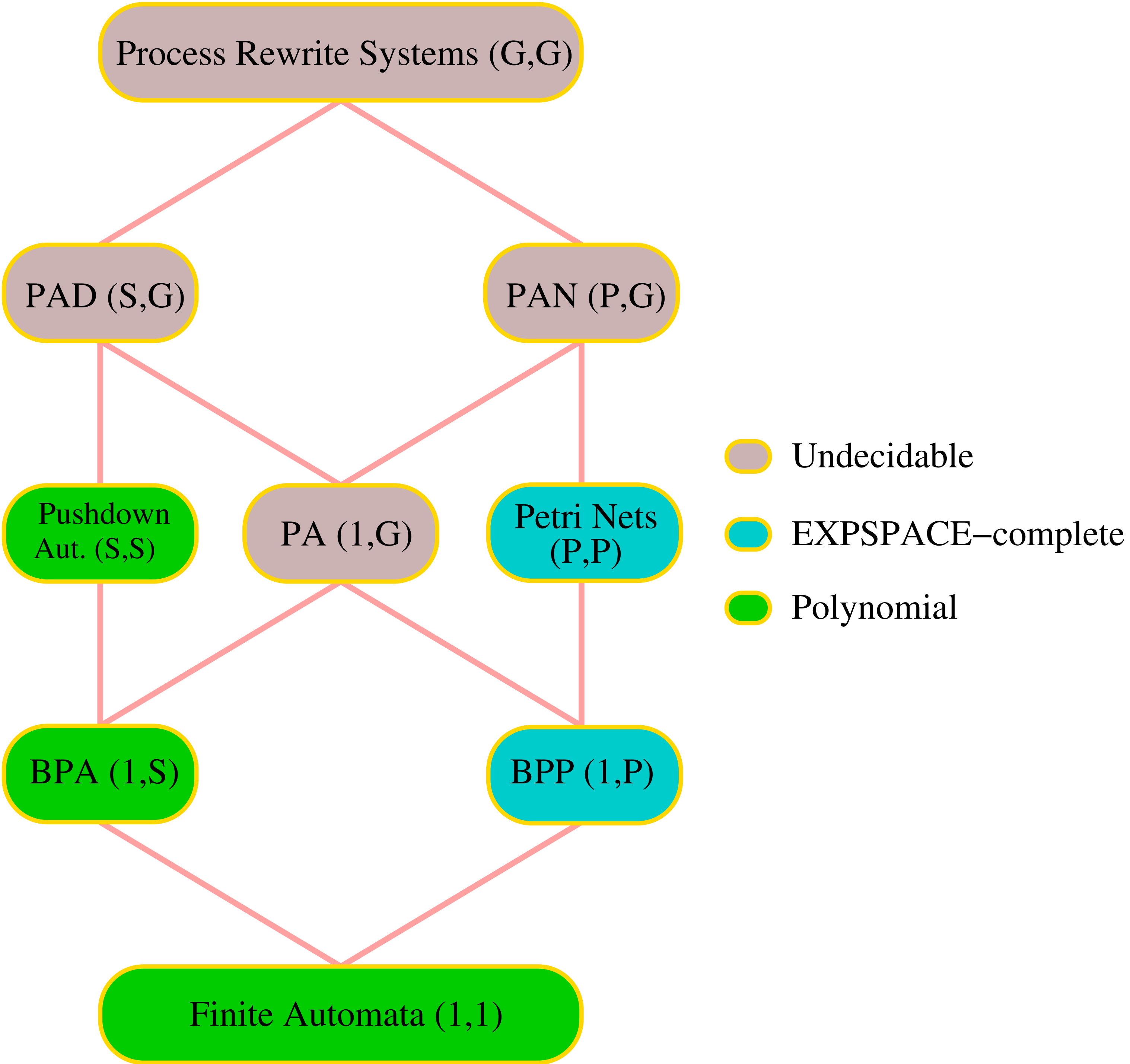
- Analysis:
  - Are there deadlocks? Is a given state reachable? Is a variable live at a program point? ...
  - Model checking: temporal logics as query languages.

  (see e.g. Burkart and Esparza's survey in EACTS Bulletin 62)

Reachability



Model-Checking LTL



The sources

[1] Bouajjani, E., Maler, CONCUR’97

[2] E., Acta Informatica ’97

[3] E., FCT ’95

[4] Kosaraju, STOC ’82

[5] Lipton, unpublished

[6] Mayr, E.W., SIAM J. of Computing, ’84

[7] Mayr, R., CONCUR ’97

[8] Mayr, R., EXPRESS ’97

BPA-processes

- Key problems: given a (possibly infinite) set of configurations, compute its (possibly infinite) set of predecessors and successors.
- Set of process terms $\Rightarrow$ set of words.
- Finite coding of infinite sets of terms by finite (word) automata.

- The set of predecessors (successors) of a regular set of process terms is regular.

  [Büchi 64], [Book, Otto 83]
- Word automata for these sets can be computed in $O(n_\Delta \cdot n_L^2)$ time and $O(n_\Delta \cdot n_L)$ space.

  [E., Hansel, Rossmanith, Schwoon 00]
- Model-checking takes $O(p \cdot b^3)$ time and $O(p \cdot b^2)$ space, where p is the size of the BPA-process, and b is the size of a Büchi automaton for the LTL formula.

  [E., Hansel, Rossmanith, Schwoon, 00]

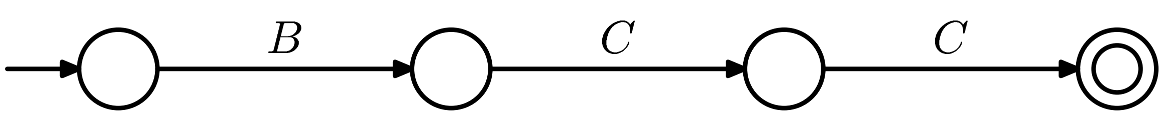
Computing predecessors

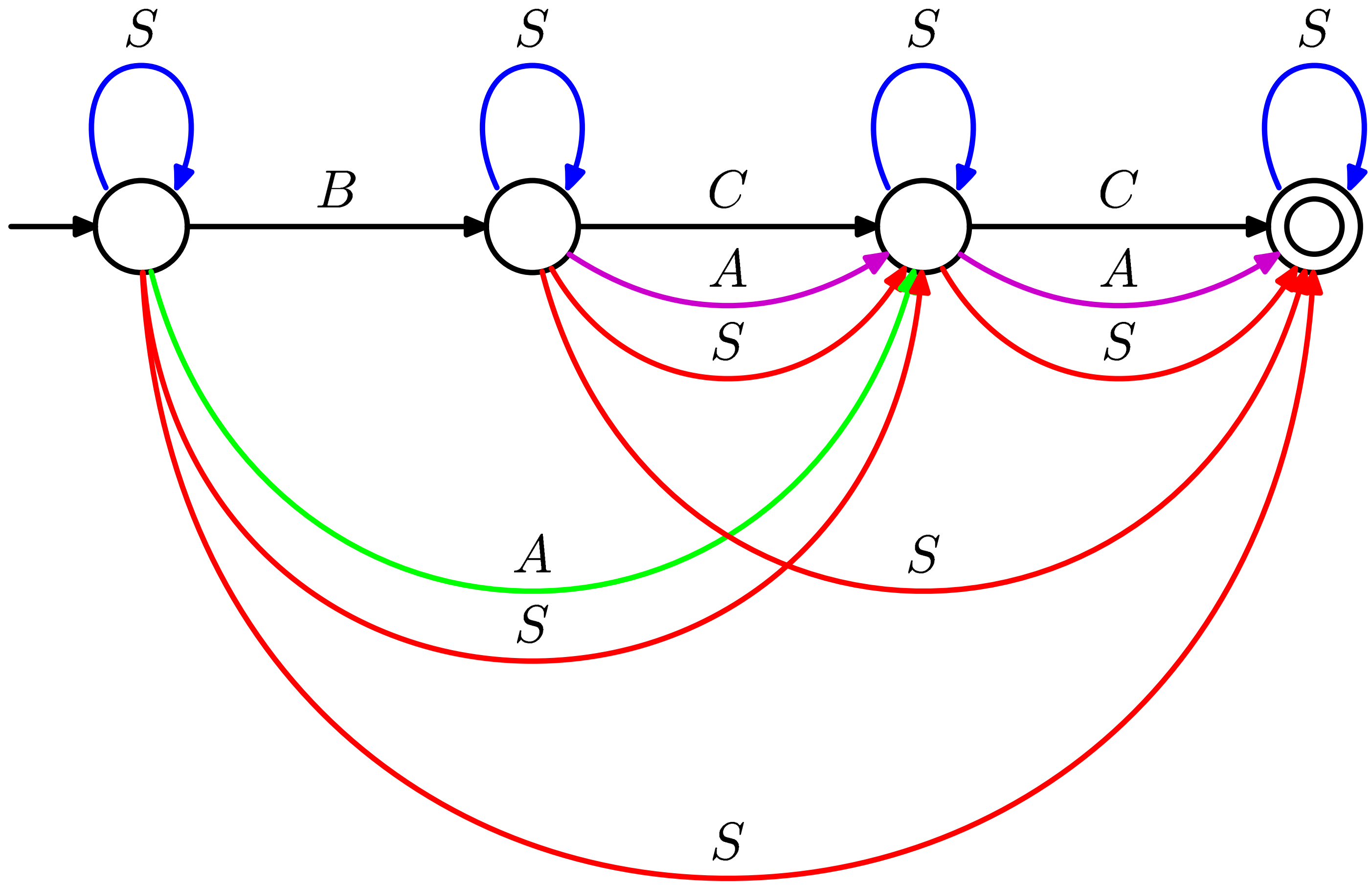
$S \longrightarrow SA \qquad A \longrightarrow BC$

$S \longrightarrow \varepsilon \qquad A \longrightarrow C$

Compute the predecessors of the term BCC.

If $X \longrightarrow w$ and
$w_1 w w_2$ is a predecessor of BCC, then
$w_1 X w_2$ is also a predecessor of BCC




S
S
S
S
B
C
C
A
A
S
S
A
S
S
S

PA-processes

- Set of process terms $\Rightarrow$ set of trees.
- Finite coding of infinite sets of terms by finite tree automata.
- The set of predecessors (successors) of a regular set of (tree) terms is regular.

  [Lugiez, Schnoebelen 99]
- Tree automata for these sets can be computed in $O(n_\Delta \cdot n_L)$ time and space.

  [E., Podelski 00]

Dataflow-analysis [E., Knoop 99]

- Goal: obtain partial information about the possible executions.
- If … then … else $\Rightarrow$ nondeterminism
- Compute dataflow information for each program point $\Rightarrow$ compute the set of program points satisfying an LTL formula.
- Direct mapping into model-checking problems. No coincidence theorems needed.
- Experiments for small LTL-formulae:

Random programs

	avg. 20 lines/procedure			avg. 40 lines/procedure		
lines	*pre**	total	space	*pre**	total	space
1000	0.23	1.31	1.00 M	0.24	1.28	0.98 M
5000	1.38	29.14	4.12 M	0.99	11.80	3.66 M
10000	3.04	123.18	7.98 M	2.93	99.38	7.58 M

Real programs: 10^4 lines in 5 – 13 seconds.

Model-checking recursive programs [E., Schwoon 01]

- Total correctness (including termination) for programs with finite datatypes but (possibly recursive) procedures.
- Counterexamples are provided when property fails.
- Finite automata for finitely encoding infinite sets of stack contents.
- BDDs for concisely encoding finite but large sets of values of global and local variables.

Example: Quicksort

```
void qs (int left,int right)
{
  int lo,hi,piv;
  if (left >= right) return;
  piv = a[right]; lo = left; hi = right;
  while (lo <= hi)
  {
    if (a[hi] > piv) { hi--; rrbr
    else {swap a[lo],a[hi]; lo++; }
  }
  qs(left,hi); qs(lo,right);
}
```

Question: total correctness.

Full information		
length	time (s)	memory (MB)
6	114	25

Abstracting from cell contents		
length	time (s)	memory (MB)
6	7	10

Machine: Ultra-60, 295 Mhz, 1.5 GB.

Exhaustive testing requires to examine about 260,000 cases.

Example: nested non-recursive procedures

Parametrised program with n procedures, benchmark for the `bebop` checker [Ball, Rajamani 00].

```
bool g;
void main() {
  level_1();
  level_1();
  if (!g) {
    reach: skip;
  }
}
```

```
void level_i() {
  int (0..7) i;
  if (g) {
    i = 0;
    while (i < 7) i++;
  } else if (i < n) {
    level_{i+1}();
    level_{i+1}();
  }
  g = !g;
}
```

Question: Is the labelled statement reachable?

- No recursion – but inlining would blow up.
- Exponential number of states – but linear solution.
- $O(n)$ variables – but locality can be exploited.

Full information		
nesting depth	time (s)	memory (MB)
800	1.8	6.7
8000	21.1	28.0

Current work

- Model-checking functional and logic programs.
- Use automata to encode infinite sets of values of global and local variables [Boigelot, Wolper 00].
- List and trees as data structures.
- Automatic abstractions.

Security properties of Java programs

- Model of JDK [Jensen, LeMétayer, Thorn 99].
- Method `CheckPermission`:

  Checkpoints at which stack contents are inspected.
  E.g., for a permission to be granted, all the methods in the call stack must have also been granted permission.

- Security property $\rightarrow$ all stack contents belong to a certain regular language L.
- Property violation: initial configuration is a predecessors of $\overline{L}$.
- Requires to deal with atomic propositions interpreted over arbitrary regular sets of configurations.

Credits

- "Grammars as processes" idea due to Caucal, extensions due to Hirshfeld.
- PRS-hierarchy in Mayr's Ph. D. Thesis.
- Application to dataflow analysis first pointed out by Steffen, developed by Knoop, Schmidt, Steffen, and others.
- Java security application communicated by Seidl.