Pattern-based Verification of Concurrent Programs

Tomáš Poch, Pierre Ganty

IMDEA internship talk

Jan 20, 2011

Reachability for sequential/concurrent programs

```
L1: bit = F;
    if bit == T
     goto error;
    else
     goto L1;
error : print "busted";
```

Is error reachable?

error reachability	unbounded data	bounded data
sequential prgs	Undecidable	Decidable

Reachability for sequential/concurrent programs

Thread t_1

```
L1:     bit = F;
        if bit == T
         goto error;
        else
         goto L1;
error : print "busted";
```

Thread t_2

```
L: bit = T;
   goto L;
```

Reachability for sequential/concurrent programs

Thread t_1

```
L1:     bit = F;
        if bit == T
         goto error;
        else
         goto L1;
error : print "busted";
```

Thread t_2

```
L: bit = T;
   goto L;
```

SM: bit

Reachability for sequential/concurrent programs

Thread t_1

```
L1:    bit = F;
       if bit == T
        goto error;
       else
        goto L1;
error : print "busted";
```

Thread t_2

```
L: bit = T;
   goto L;
```

SM: bit

Is error reachable?

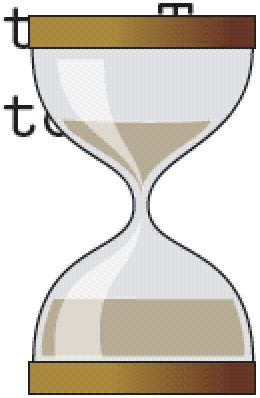
Reachability for sequential/concurrent programs

Thread t_1

```
L1: bit = F;
    if bit == T
      goto error;
    else
      goto L1;
error : print "busted";
```

Thread t_2

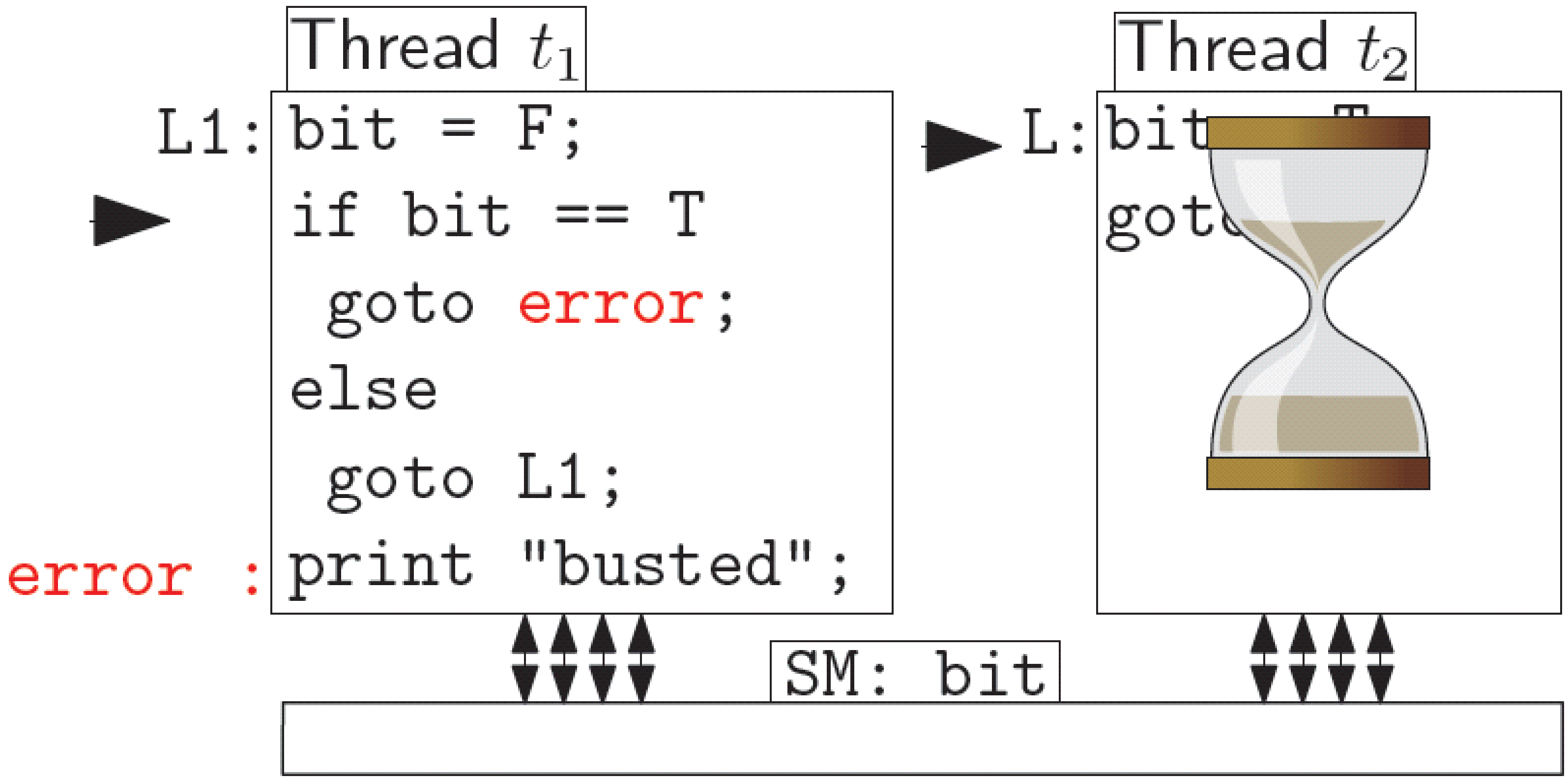
```
L: bit
   got
```



SM: bit

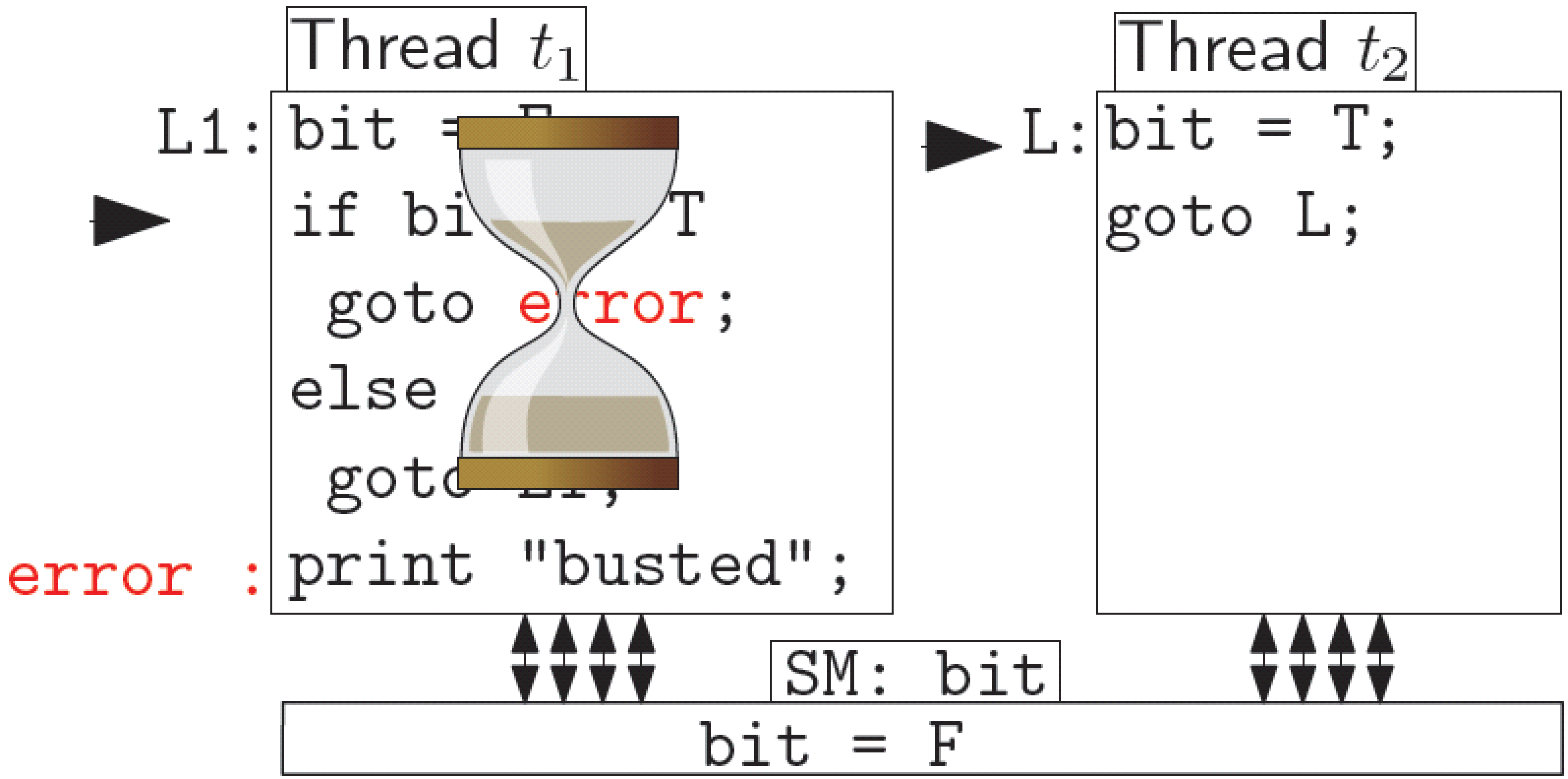
Is error reachable?

Reachability for sequential/concurrent programs



Is error reachable?

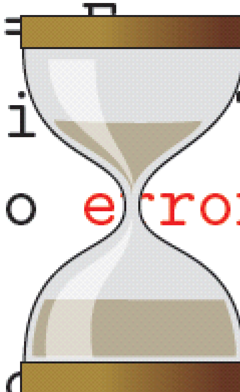
Reachability for sequential/concurrent programs



Is error reachable?

Reachability for sequential/concurrent programs

Thread t_1



```
L1:     bit = F;
        if bit == T
          goto error;
        else
          goto L1;
error : print "busted";
```

Thread t_2

```
L: bit = T;
   goto L;
```

SM: bit

```
bit = F
```

Is error reachable?

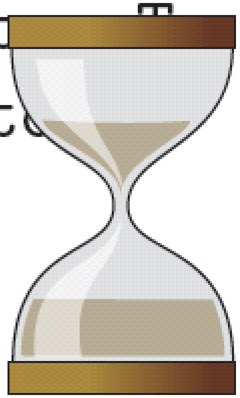
Reachability for sequential/concurrent programs

Thread t_1

```
L1:    bit = F;
       if bit == T
        goto error;
       else
        goto L1;
error: print "busted";
```

Thread t_2

```
L: bit
   goto
```

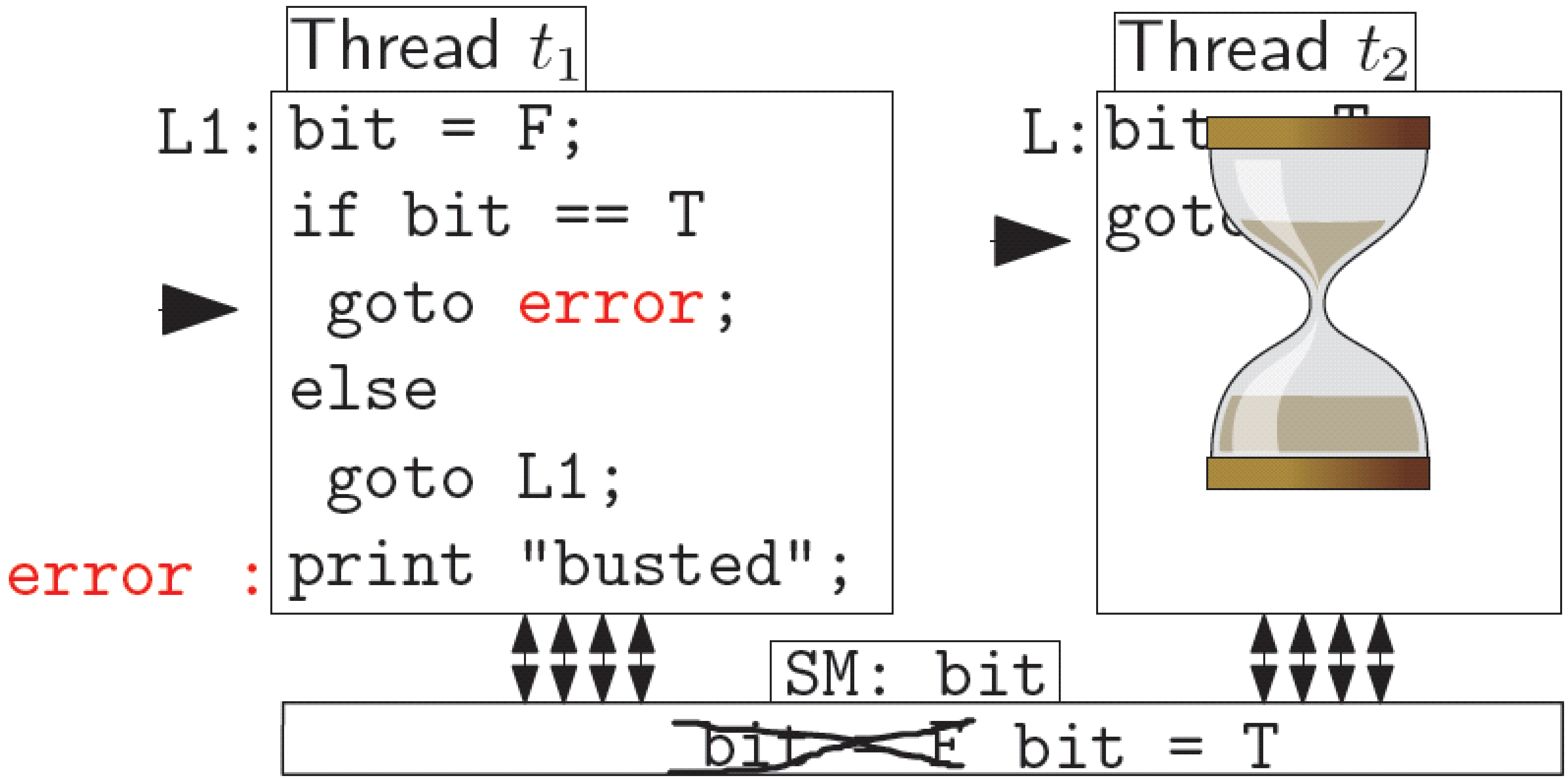


SM: bit

~~bit = F~~ bit = T

Is error reachable?

Reachability for sequential/concurrent programs



Is error reachable?

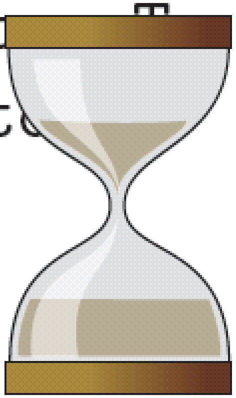
Reachability for sequential/concurrent programs

Thread t_1

```
L1:      bit = F;
         if bit == T
           goto error;
         else
           goto L1;
error :  print "busted";
```

Thread t_2

```
L: bit
   got
```

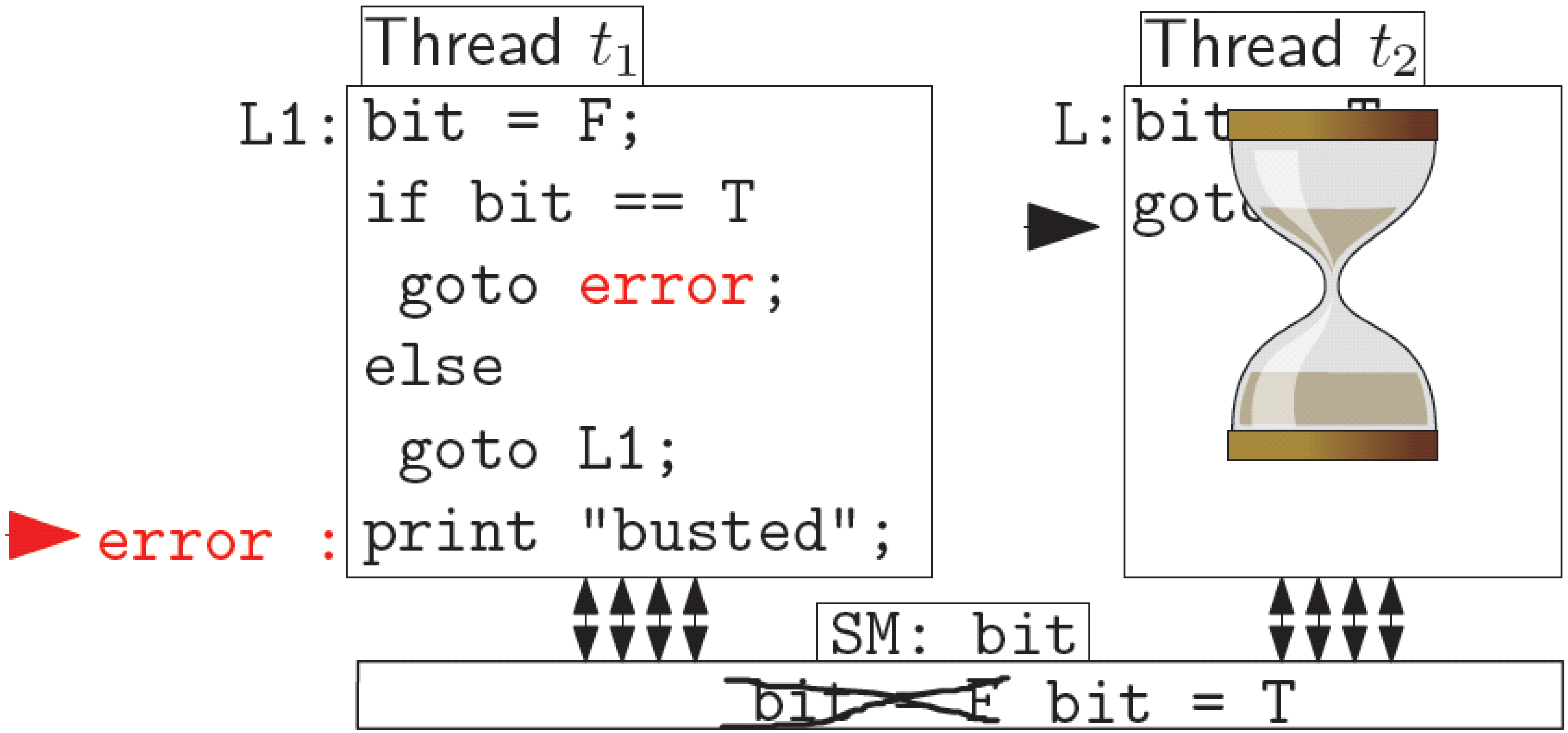


SM: bit

~~bit = F~~ bit = T

Is error reachable?

Reachability for sequential/concurrent programs



error reachability	unbounded data	bounded data
sequential prgs	Undecidable	Decidable
multithreaded prgs	Undecidable	Undecidable

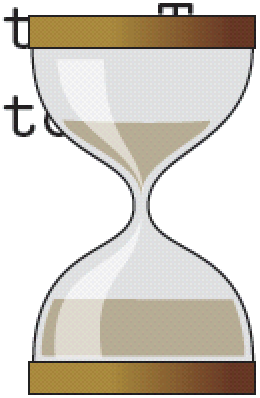
Reachability for sequential/concurrent programs

Thread t_1

```
L1:     bit = F;
        if bit == T
         goto error;
        else
         goto L1;
error : print "busted";
```

Thread t_2

```
L: bit
   goto
```



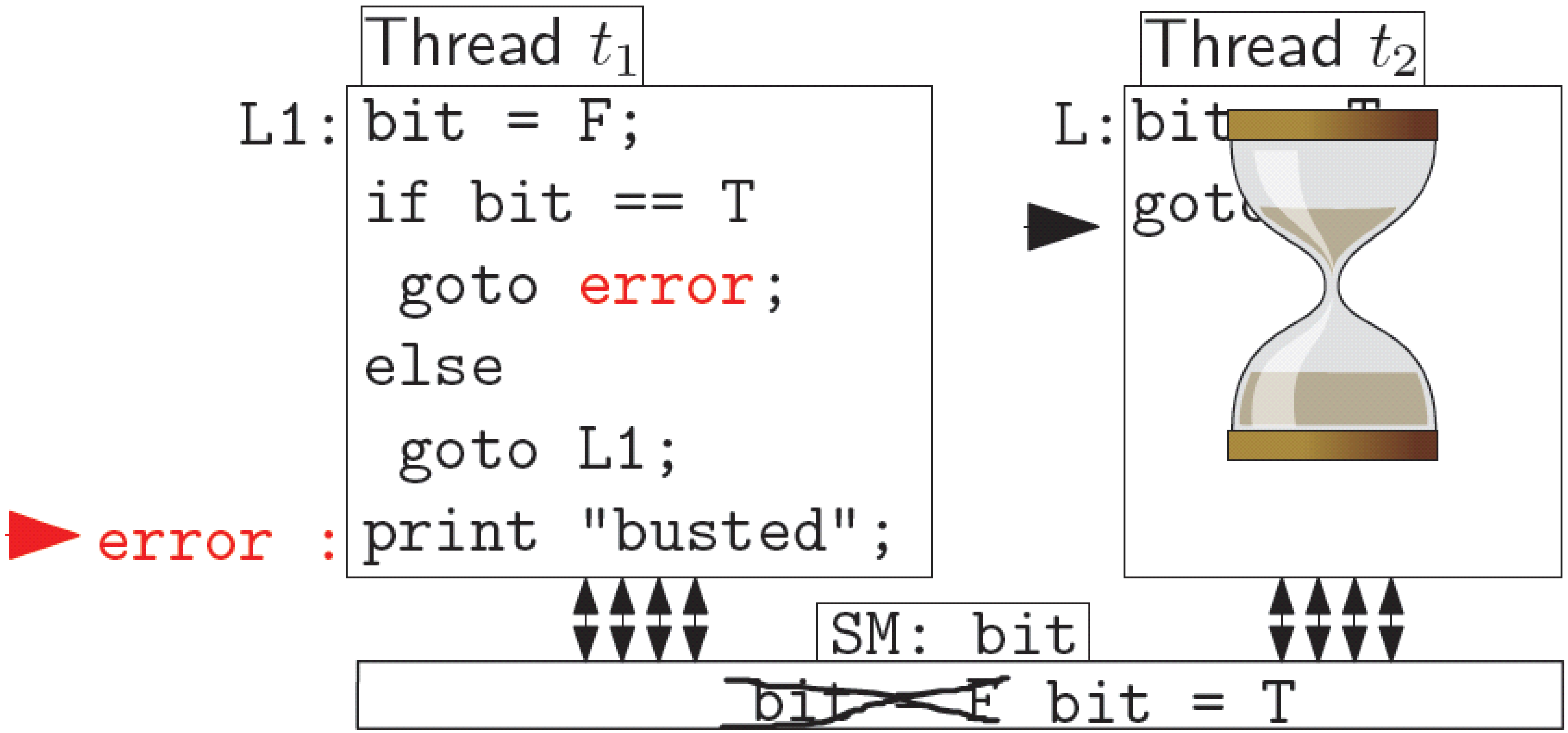
SM: bit

~~bit = F~~ bit = T

Is error reachable?

Decidability by further restricting $\underbrace{\text{SM visited states}}_{\text{coverage}}$

Reachability for sequential/concurrent programs



Is error reachable for k writes to SM ?

k-context switches reachability is decidable, and NPc *

* Shaz Qadeer, Jakob Rehof. *Context-Bounded Model Checking of Concurrent Software* in TACAS '05

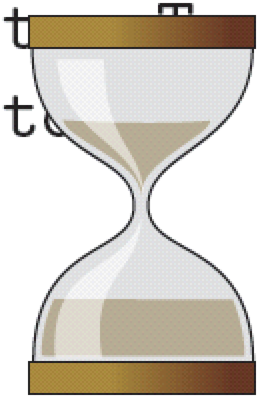
Reachability for sequential/concurrent programs

Thread t_1

```
L1:    bit = F;
       if bit == T
         goto error;
       else
         goto L1;
error : print "busted";
```

Thread t_2

```
L: bit
   got
```



SM: bit

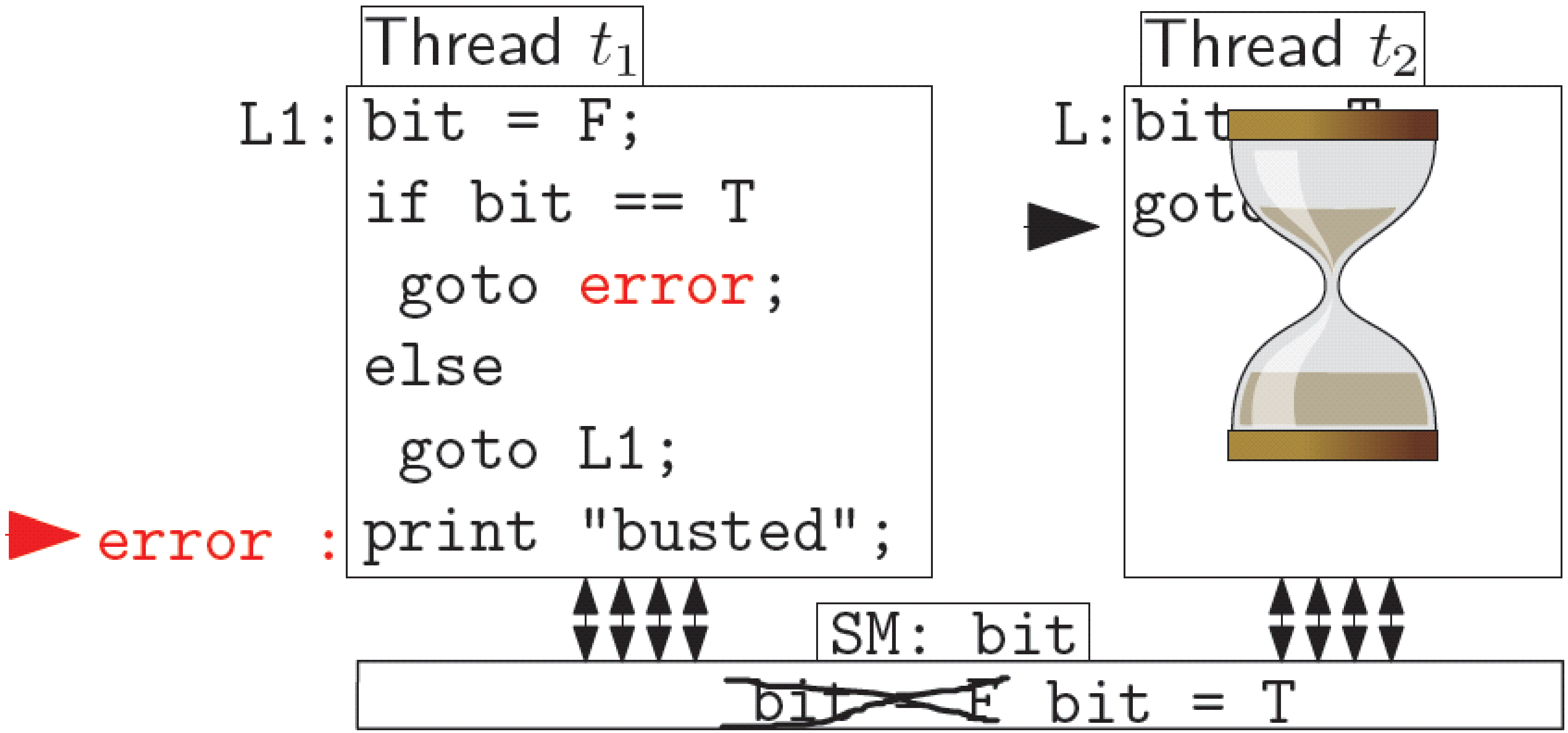
~~bit = F~~ bit = T

Is error reachable for pattern $(\texttt{bit} = \texttt{F} \cdot \texttt{bit} = \texttt{T})^*$?

Pattern-based reachability is decidable †

† Pierre Ganty, Rupak Majumdar, Benjamin Monmege. *Bounded Underapproximations* in CAV '10

Reachability for sequential/concurrent programs



Pattern takes the form $w_1^* \, w_2^* \ldots w_n^*$

w_i is a word, symbols represent data in SM

Pattern-based reachability is decidable †

† Pierre Ganty, Rupak Majumdar, Benjamin Monmege. *Bounded Underapproximations* in CAV '10

From SM programs to grammars

Shared memory program consist of

- Set of procedures accessing local and global variables (bounded data)
- Set of threads having initial points

Message passing program consist of

- Set of threads and procedures accessing local variables (bounded data), sending/receiving messages

Message passing example

```
void main()
  int x = 5;
  while (x>0) {
    b();
    x--;
  }
}

void b(){
  int y;
  reicv(ch1,y);
  send(ch2,y);
}
```

- The example copies 5 items from the channel ch1 to ch2
- Semantics
  - receive(ch1,var)
    - Block until someone calls send(ch1,data) and all the others call receive(ch1,var2)
    - assign var=data
  - send(ch1,data)
    - Block until all other threads call receive(ch1,var)

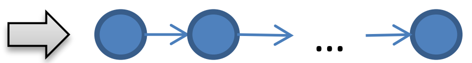
Shared memory as message passing

- Modify SM program:
  - All variables are local
  - At each program location simulate context switch
    - Send the content of ‘global’ data
    - Switch to the inactive state
    - Wait until someone else sends the content of memory

- Two messages for each context switch
  - yield(gmem) – go to inactive state
  - go(TID) – go back to the active state

Context switching as msg passing

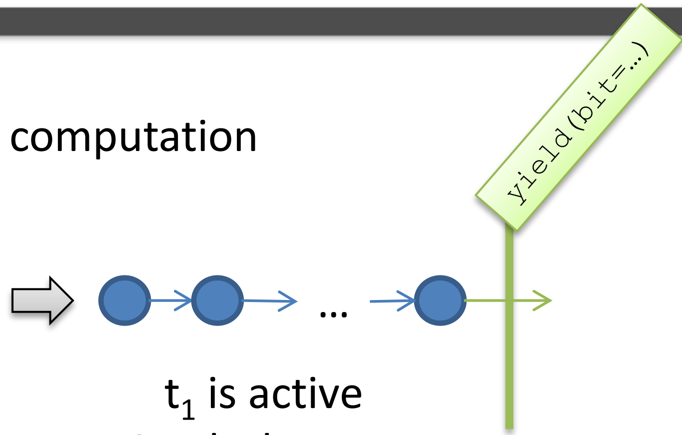
t_1 computation



t_1 is active
Stack changes
No messages produced

Context switching as msg passing

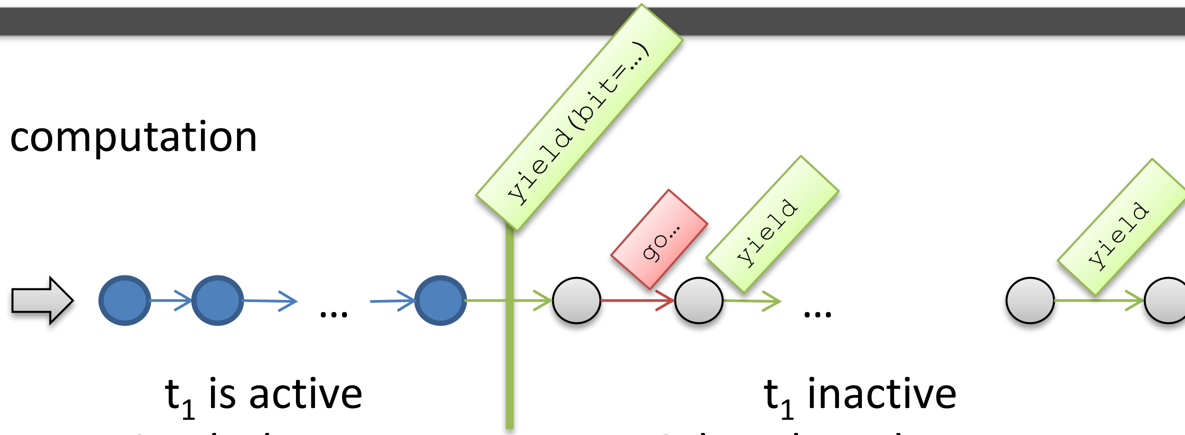
t_1 computation



t_1 is active
Stack changes
No messages produced

Context switching as msg passing

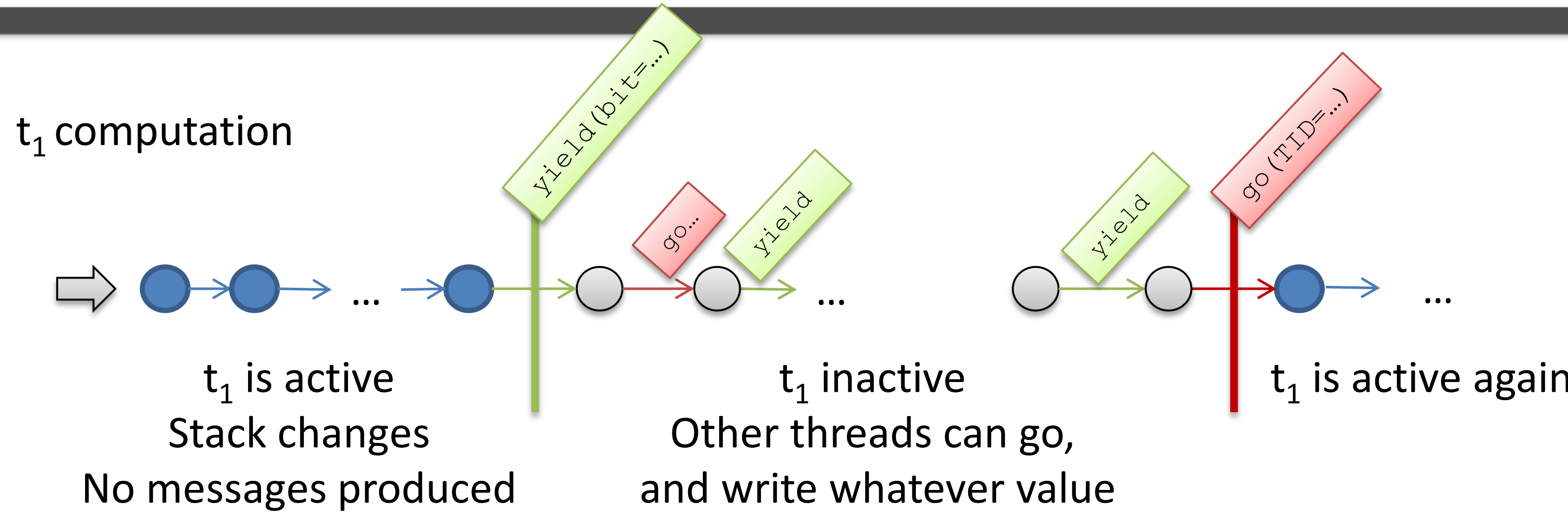
t_1 computation



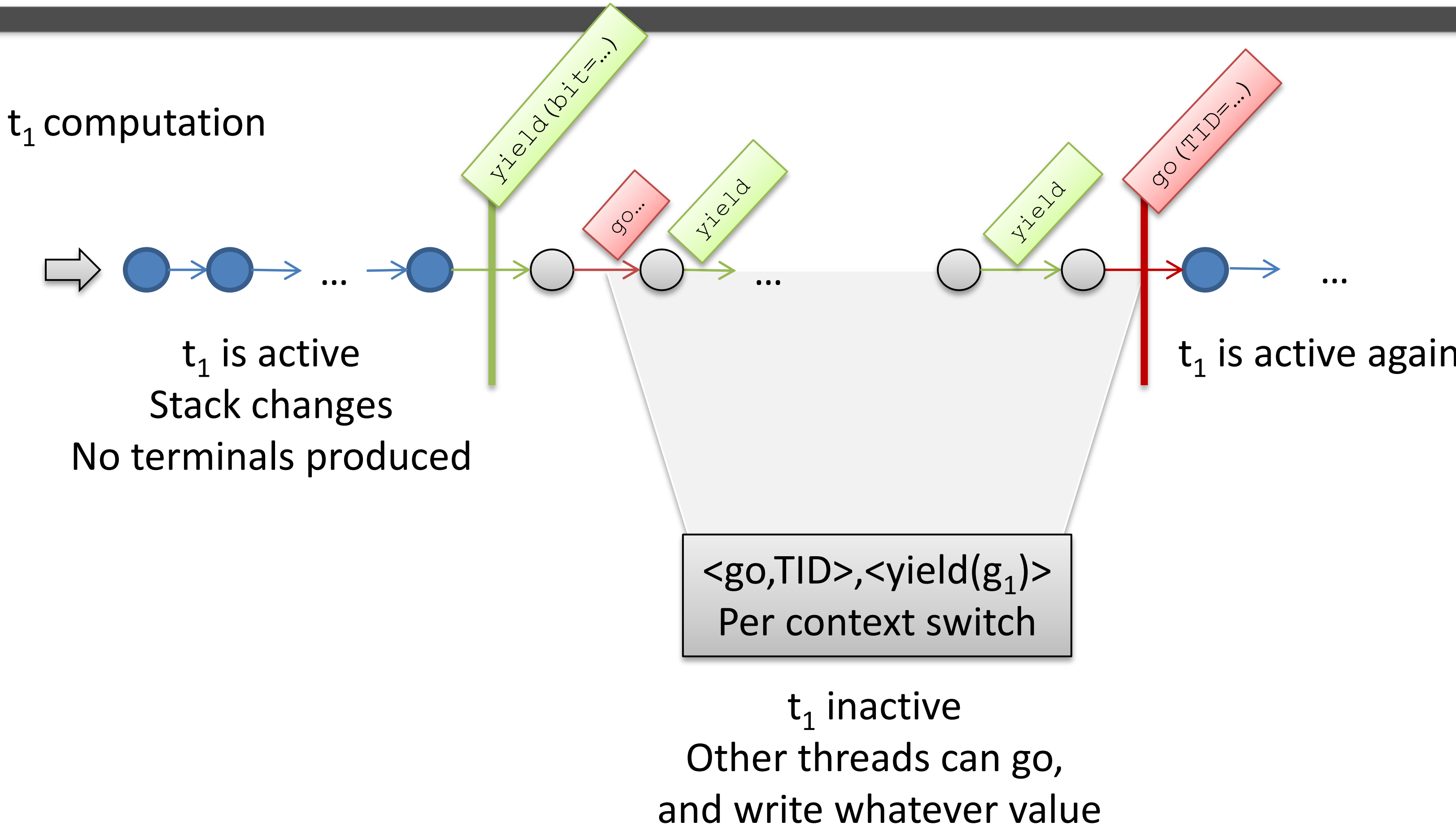
t_1 is active
Stack changes
No messages produced

t_1 inactive
Other threads can go,
and write whatever value

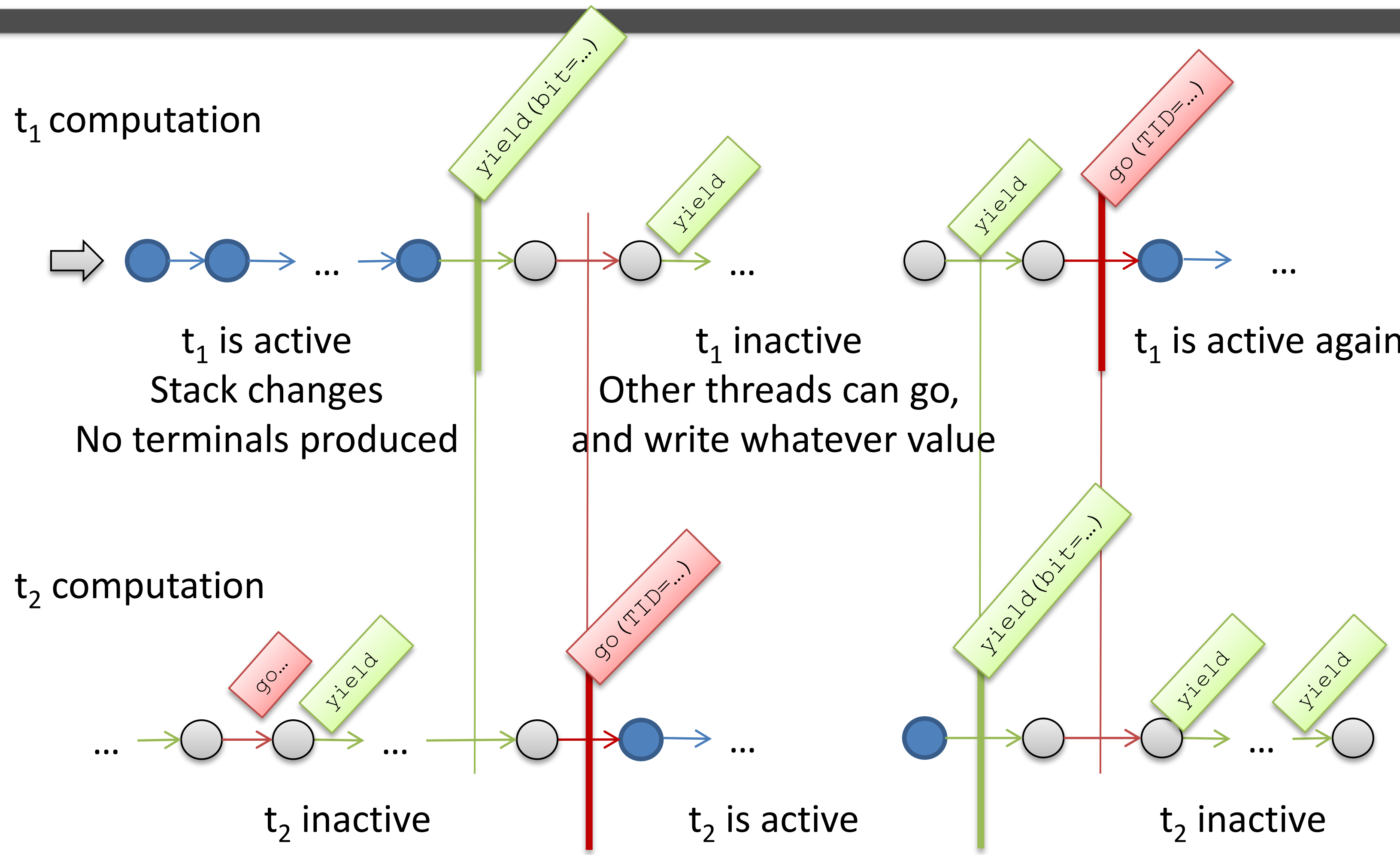
Context switching as msg passing



Context switching as msg passing

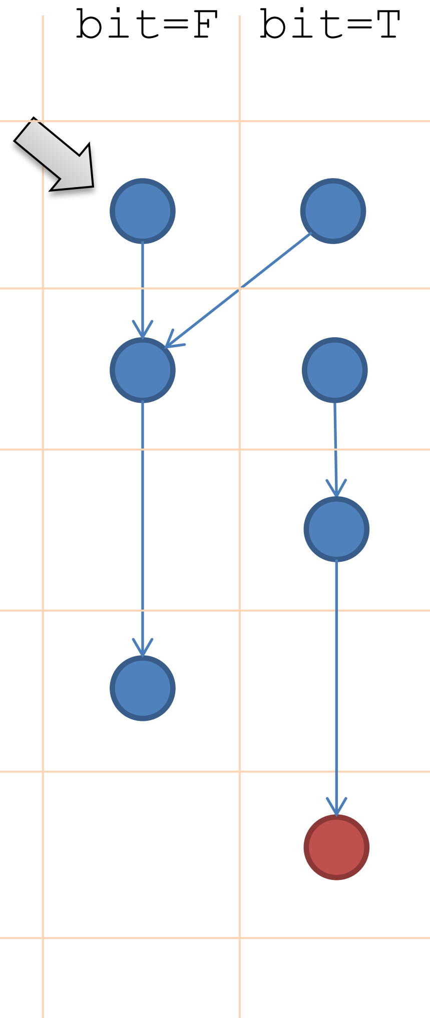


Context switching as msg passing



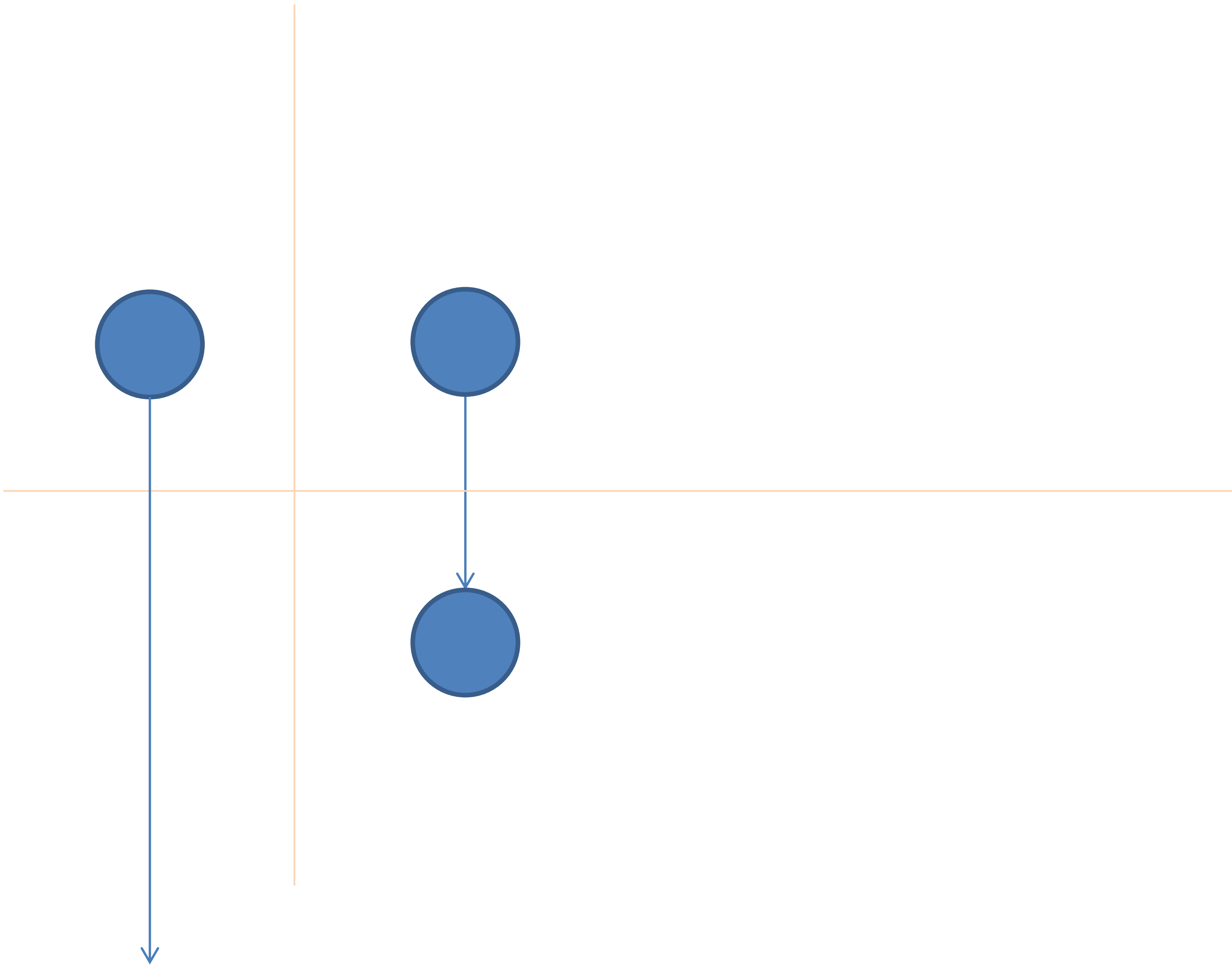
Modification of SM program

```
0   bit=F;
1   if bit==T
2     goto 4;
3   return;
4   print "busted"
```



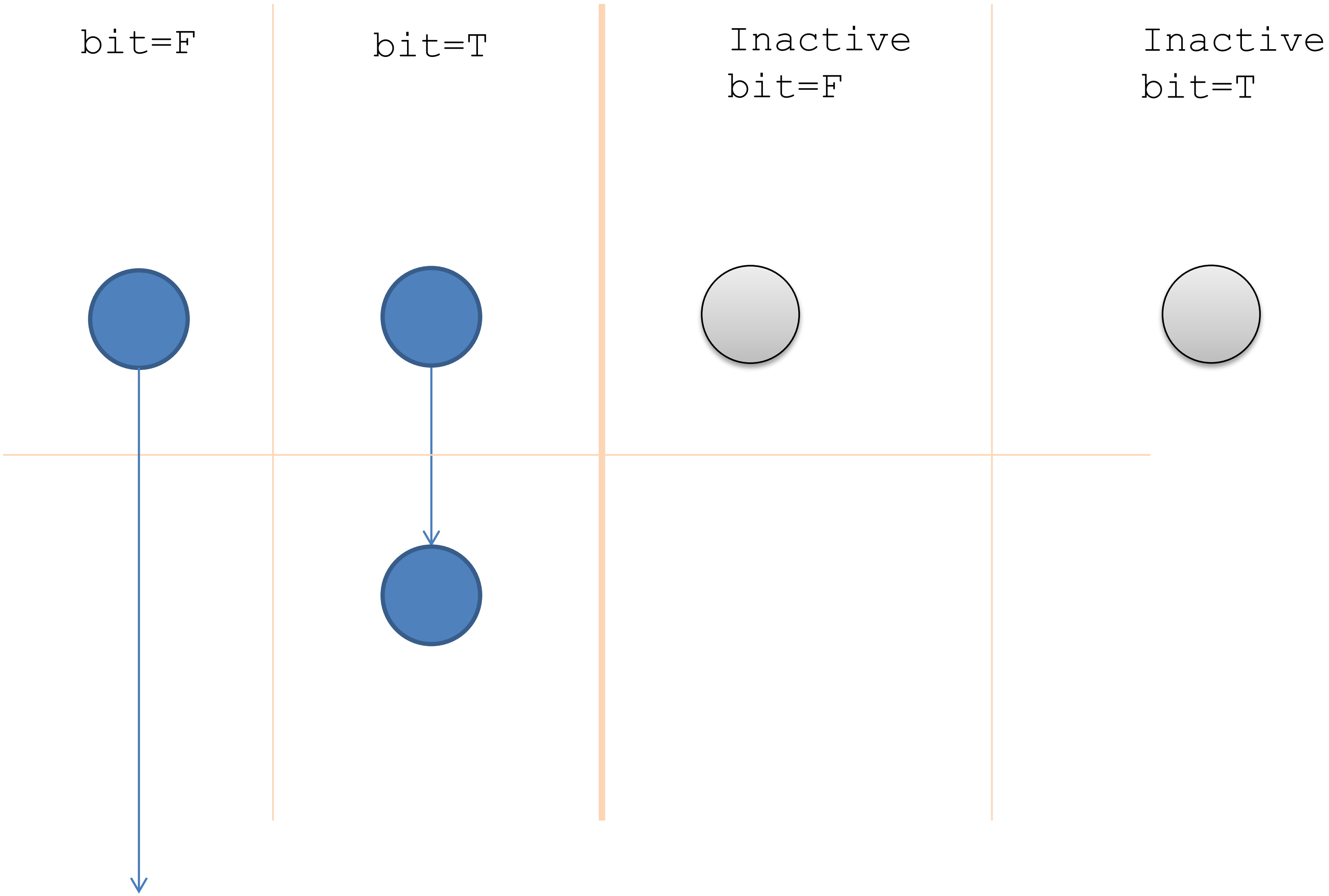
Inactive states, communication

TID=0



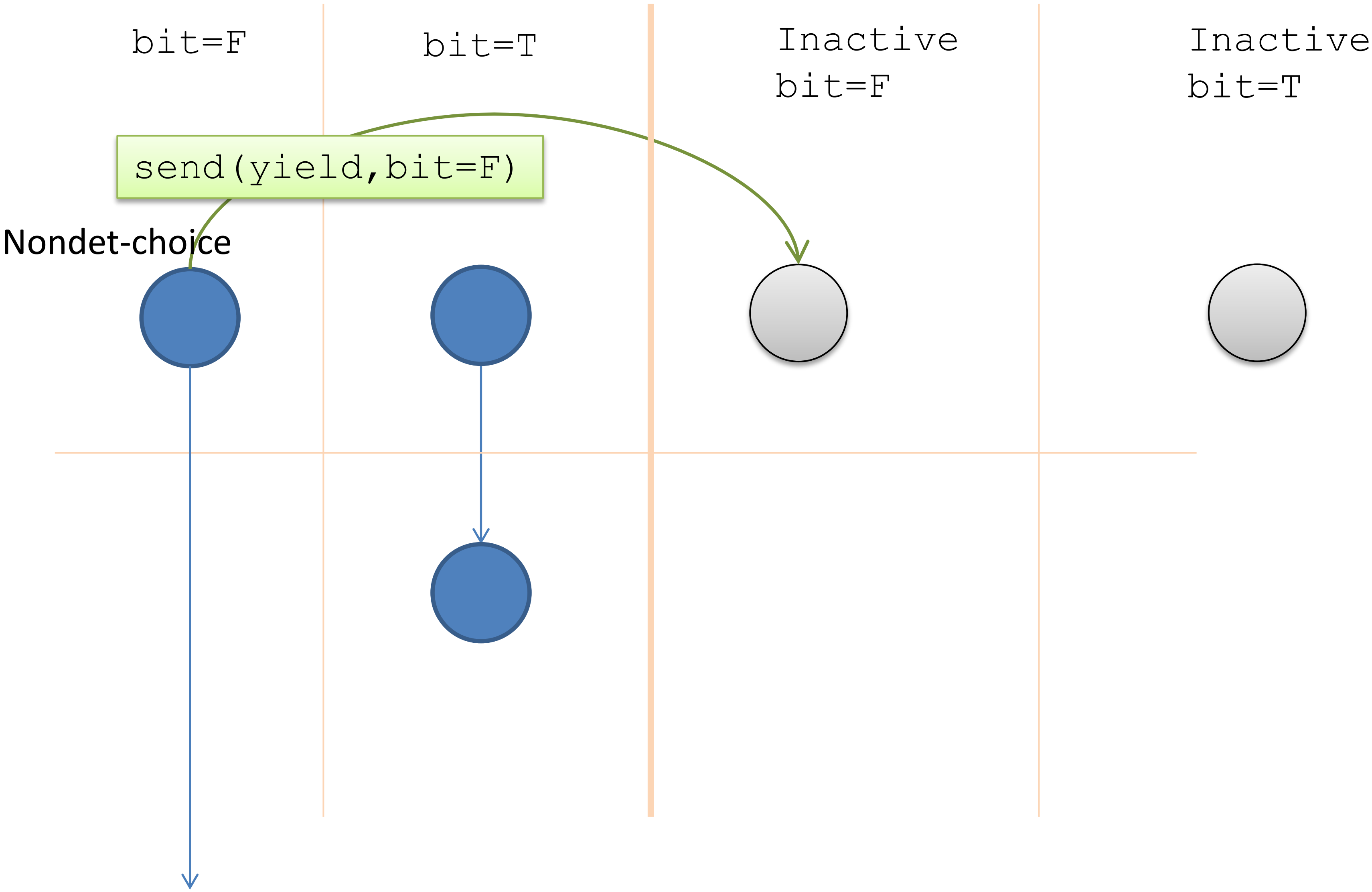
Inactive states, communication

TID=0



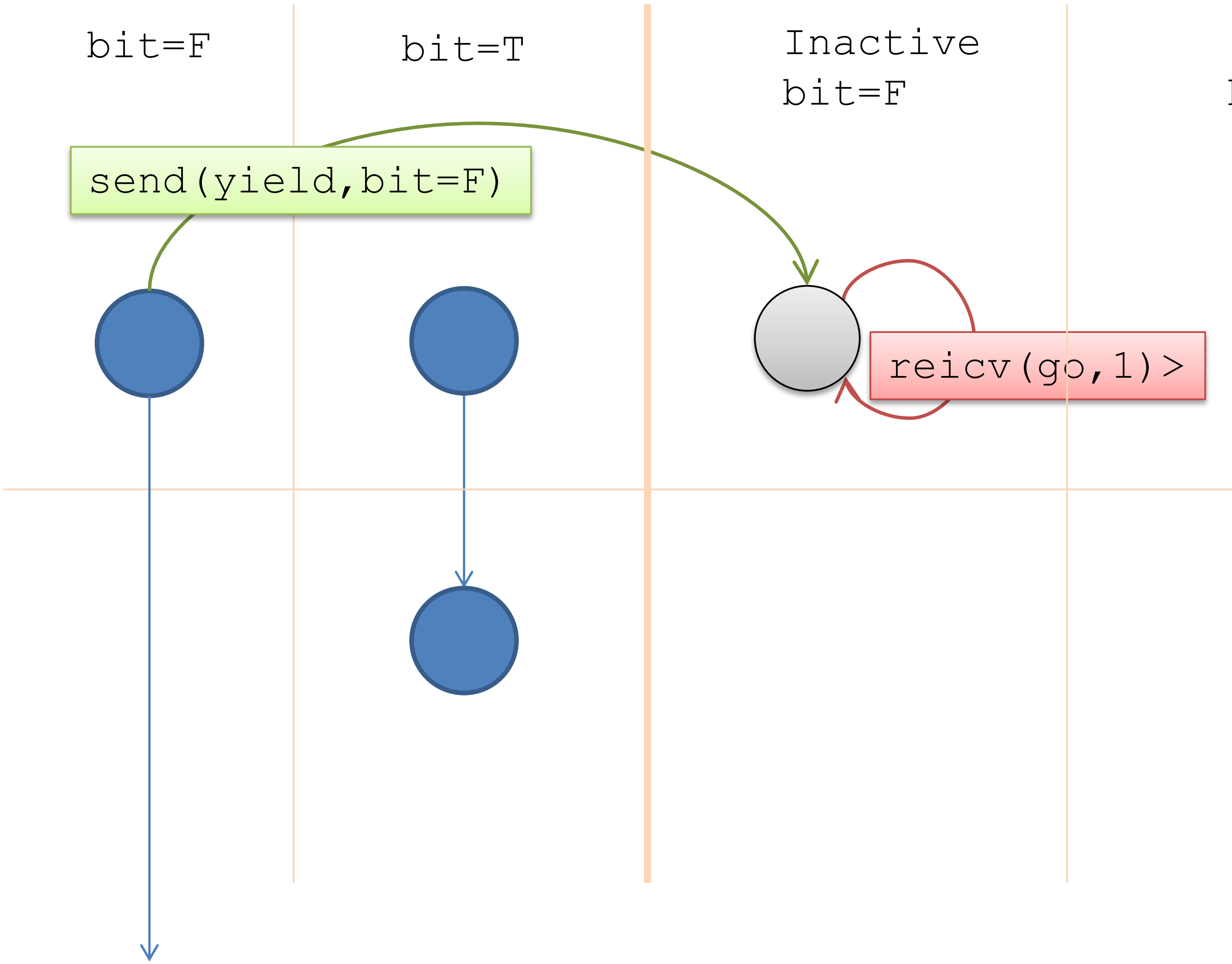
Inactive states, communication

TID=0



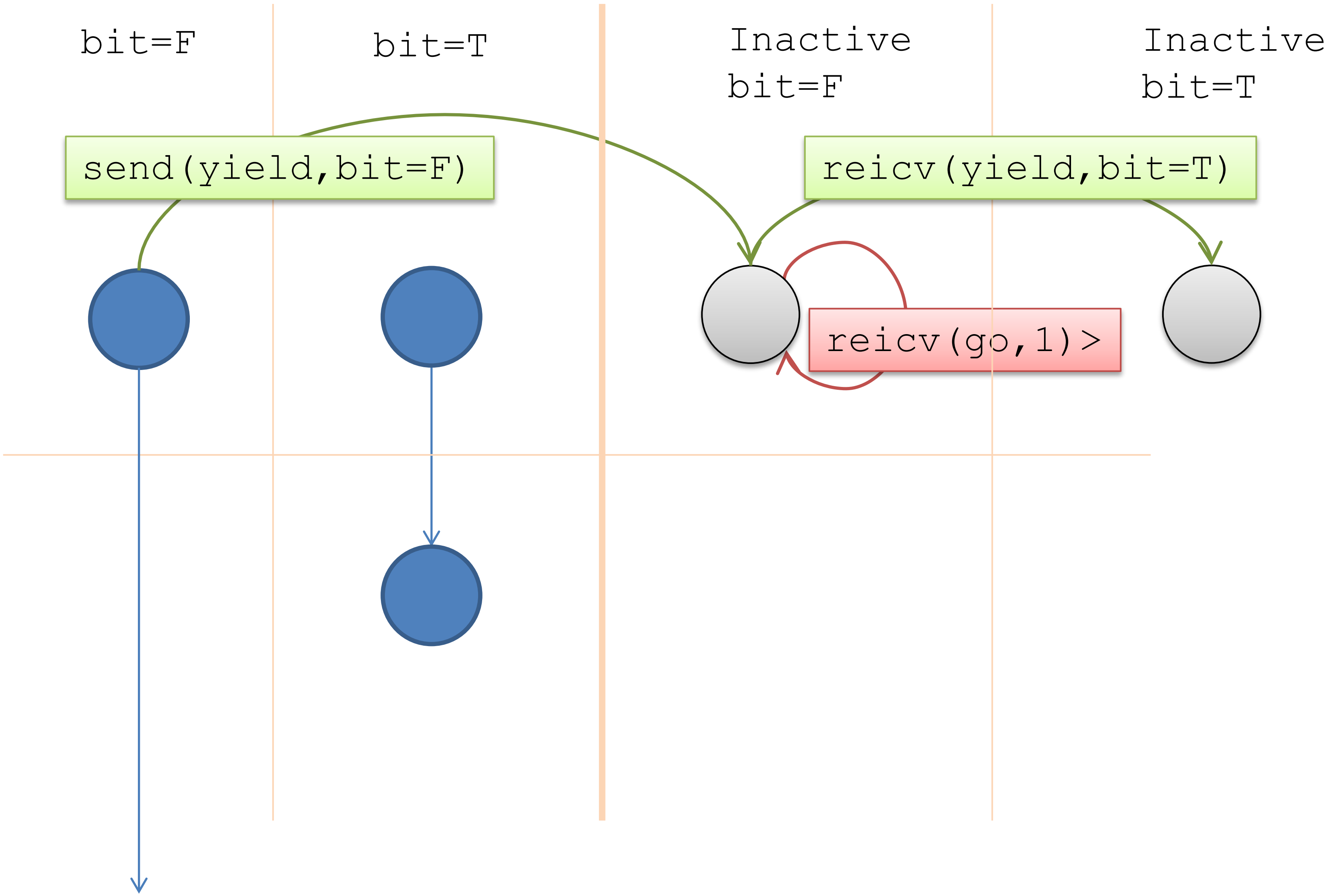
Inactive states, communication

TID=0



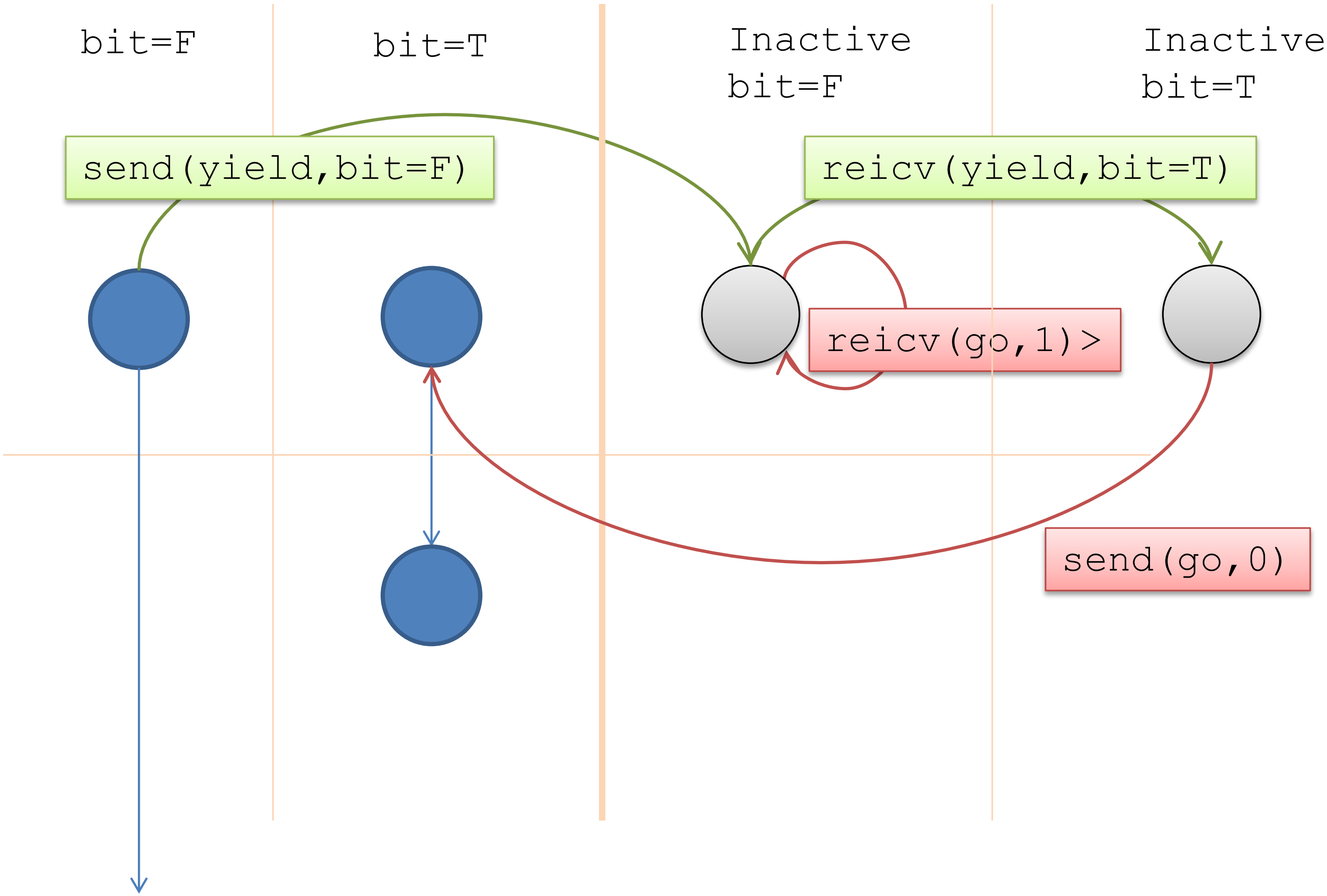
Inactive states, communication

TID=0



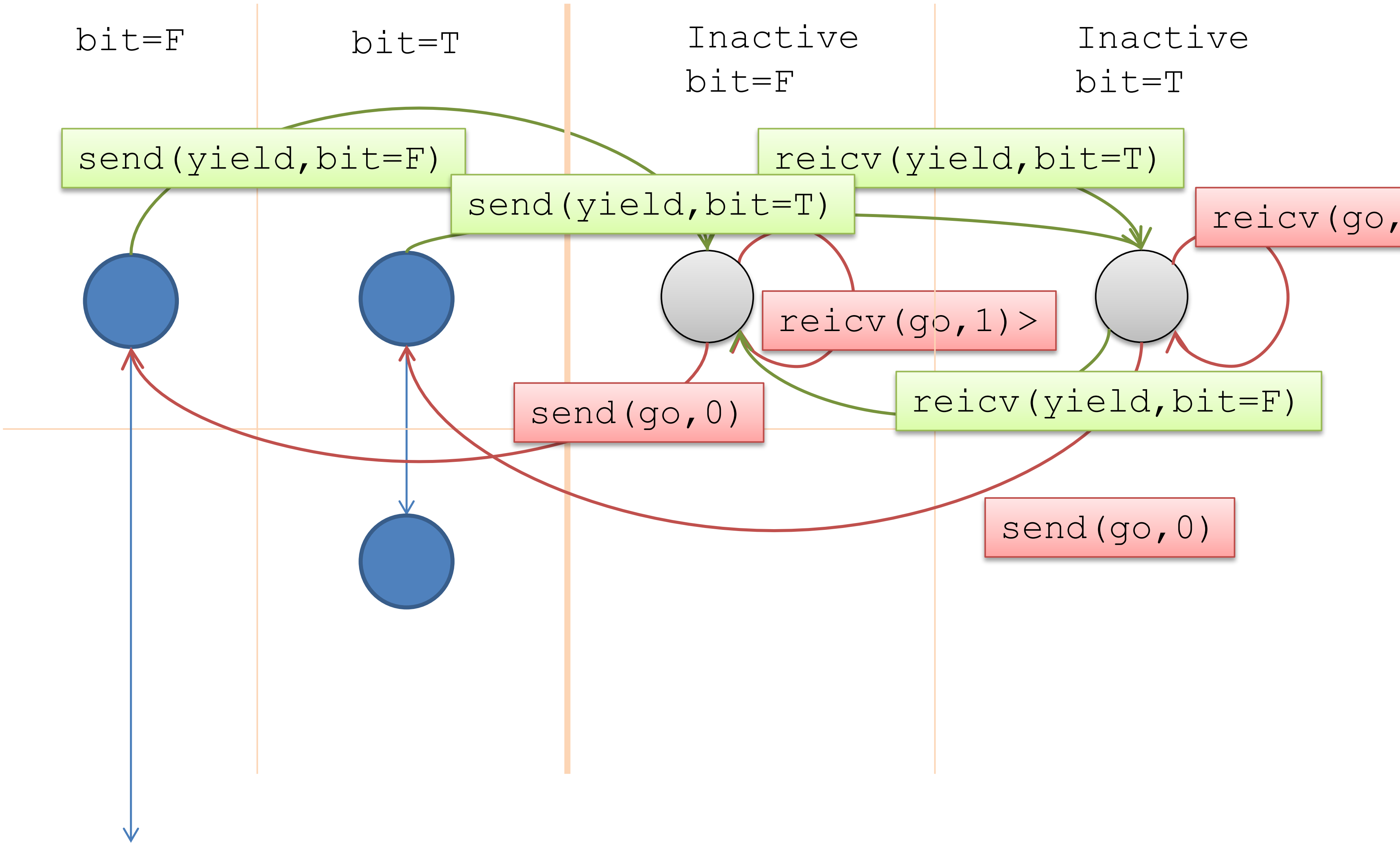
Inactive states, communication

TID=0



Inactive states, communication

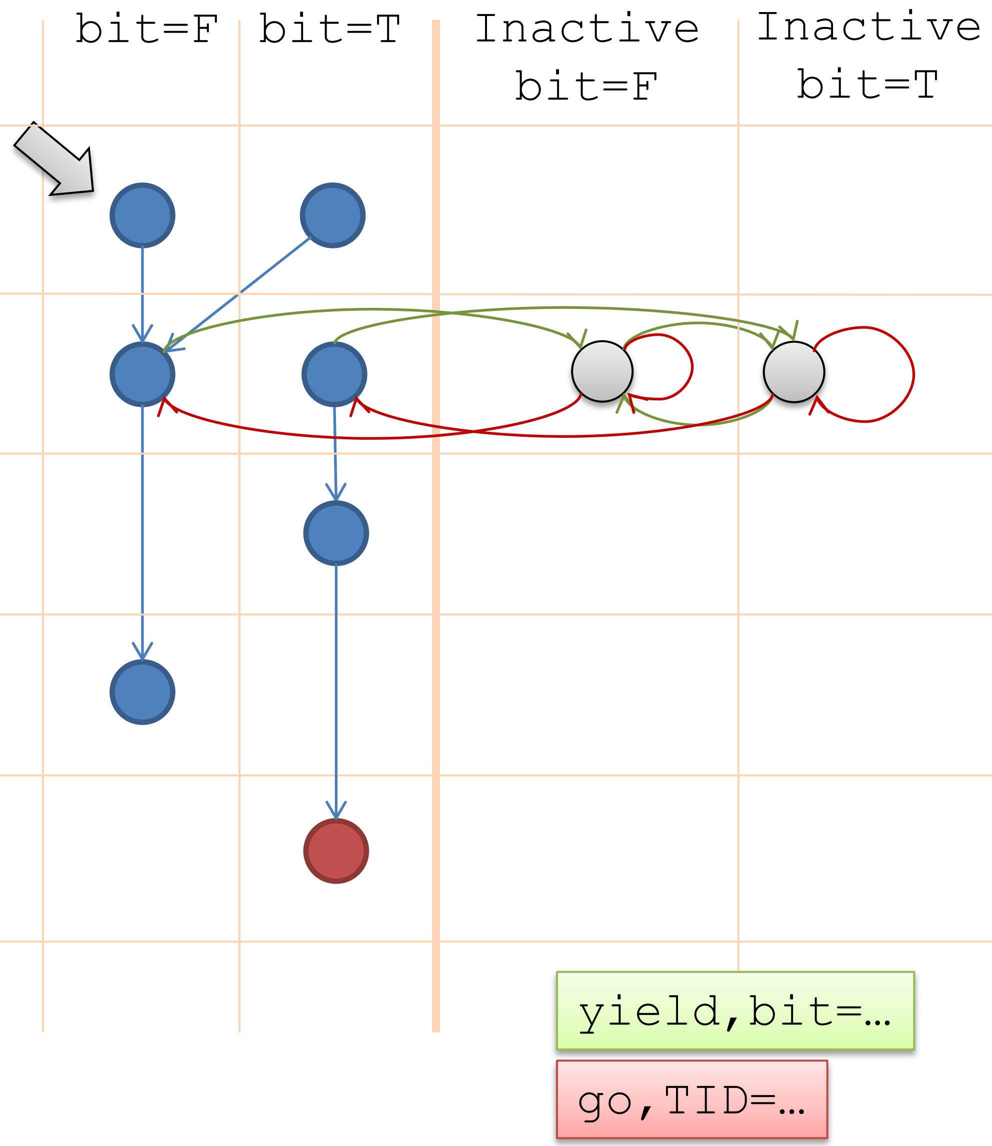
TID=0



Modification of SM program

TID = 0

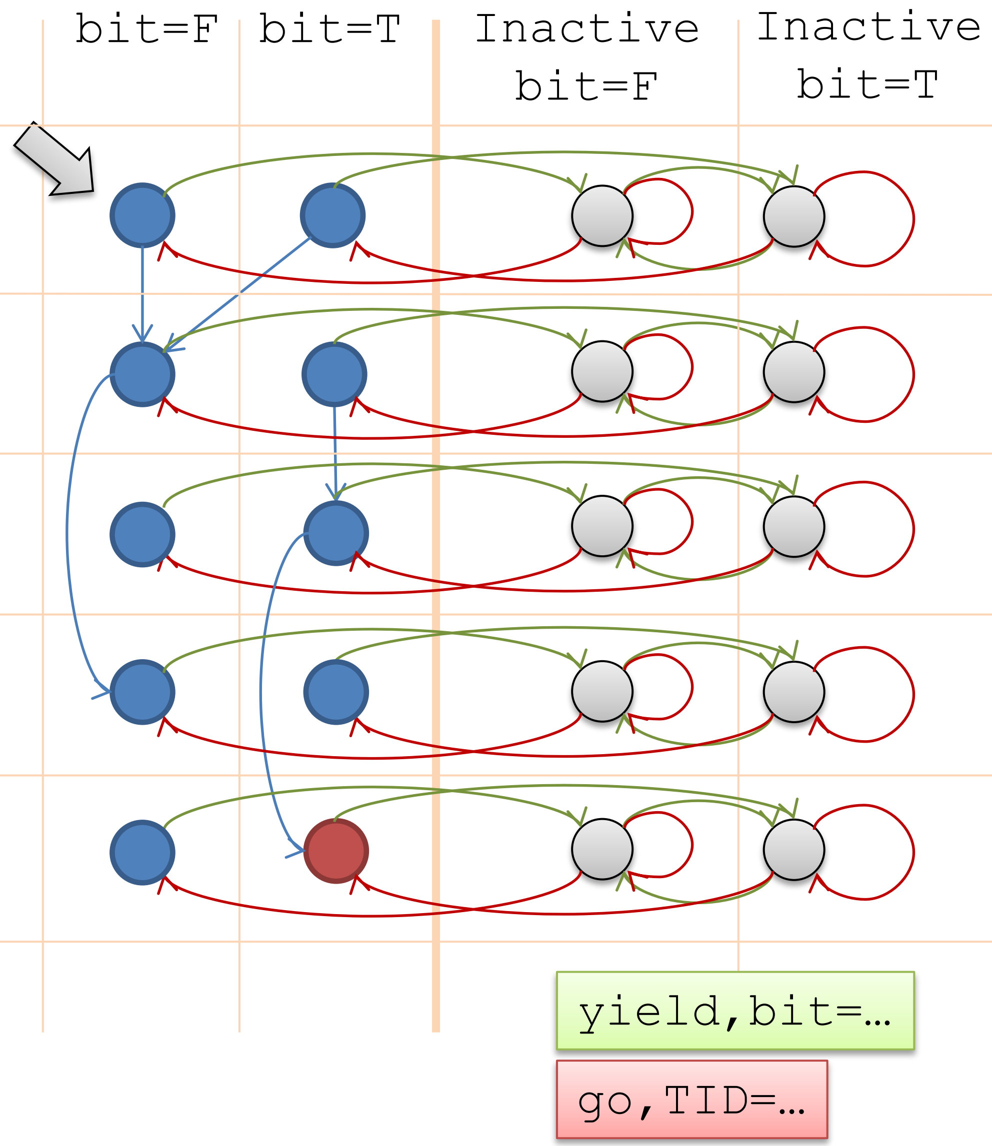
```
0  bit=F;
1  if bit==T
2    goto 4;
3  return;
4  print "busted"
```



Modification of SM program

```
TID = 0

0  bit=F;
1  if bit==T
2     goto 4;
3  return;
4  print "busted"
```



Msg passing prg. as ctx-free grammar

- t_i represented by context-free grammar G_i
  - Non-terminals encode program positions and variables
  - Grammar rules simulate program transition
    - Context-free grammar needed to support function calls
  - Terminals encode communication

- $w \in L(G_i)$ – sequence of t_i communications
- $w \in L(G_1) \cap L(G_2)$ – history allowed by both threads

Emptiness of $L(G_1) \cap L(G_2)$ is undecidable

Msg passing as grammar

```
  void main()
  int x = 5;
  while (x>0) {
    b();
    x--;
  }
}

void b(){
  reicv(ch);
}
```

Msg passing as grammar

```
  void main()
0  int x = 5;
1  while (x>0) {
2    b();
3    x--;
4  }
5}

 void b(){
6  reicv(ch);
7}
```

```
Init          -> [x'=5,pc=1]
[x>0,pc=1] -> [x'=x,pc=2]
[x=0,pc=1] -> [x'=0,pc=5]
[x,pc=2] -> [pc=6][x'=x,pc=3]
[x,pc=3] -> [x'=x-1,pc=1]
[x,pc=5] -> ε

[pc=6] -> <ch>
```

Msg passing as grammar

```
  void main()
0   int x = 5;
1   while (x>0) {
2     b();
3     x--;
4   }
5 }

  void b(){
6   reicv(ch);
7 }
```

```
Init          -> [x'=5,pc=1]
[x>0,pc=1] -> [x'=x,pc=2]
[x=0,pc=1] -> [x'=0,pc=5]
[x,pc=2] -> [pc=6][x'=x,pc=3]
[x,pc=3] -> [x'=x-1,pc=1]
[x,pc=5] -> ε

[pc=6] -> <ch>
```

Fcn call

Msg passing as grammar

```
  void main()
0  int x = 5;
1  while (x>0) {
2    b();
3    x--;
4  }
5}

 void b(){
6  reicv(ch);
7}
```

```
Init          -> [x'=5,pc=1]
[x>0,pc=1] -> [x'=x,pc=2]
[x=0,pc=1] -> [x'=0,pc=5]
[x,pc=2] -> [pc=6][x'=x,pc=3]
[x,pc=3] -> [x'=x-1,pc=1]
[x,pc=5] -> ε

[pc=6] -> <ch>
```

Init →* <msg><msg><msg><msg><msg>

The process will reach its final position (5) only if the cooperating thread can produce 5 messages as well

Decision procedure

- Reachability in concurrent program as a language problem
  - Intersection of context-free languages
    - Emptiness of $L(G_1) \cap L(G_2)$ undecidable

- Context bounded verification
  [Shaz Qadeer, Jakob Rehof. Context-Bounded Model Checking of Concurrent Software in Tacas ’05]
  - At most k context switches
    - Emptiness of $L(G_1) \cap L(G_2) \cap \{go(TID), yield(gmem)\}^{2k}$

- Pattern based verification
  [Pierre Ganty, Rupak Majumdar, Benjamin Monmege. Bounded Underapproximations in CAV’10]
  - Context switches follow the pattern
    - Emptiness of $L(G_1) \cap w_1^{*} \ldots w_n^{*} \cap L(G_2)$
    - $w_i \in \{go(TID), yield(gmem)\}^{*}$
    - Example – at most k ctx sw : $(go(0)^{*}\ go(1)^{*}\ yield(true)^{*}\ yield(false)^{*})^{K}$

Decision Procedure

$$L(G_1) \cap w_1^* \ldots w_n^* \cap L(G_2) = \varnothing$$

Decision procedure

- Counting of w matters
  $w_1^i\, w_2^j \ldots w_n^k \in L(G) \cap w_1^* \ldots w_n^*$

- Modify G to G'
  - Accept only words from pattern
    - CFL are closed to intersection w/ regular languages
  - Produce single terminal a_p instead of the word w_p

- $w_1^i \ldots w_n^k \in L(G) \cap w_1^* \ldots w_n^* \Leftrightarrow a_1^i \ldots a_n^k \in L(G')$

Parikh image

- Fixed linear order over alphabet
  - $\Sigma=\{a_1, a_2 \dots a_p\}$
- Parikh image of $w \in \Sigma^*$ is a p-dimensions vector
  - i-th part is the number of occurrences of i-th symbol in w
  - $\Pi(w) = \langle i_1, i_2, \dots, i_p \rangle$, $\Pi(a_1a_1a_1a_2) = \langle 3,1,0,\dots,0 \rangle$
- Parikh image of language $L \subseteq \Sigma^*$
  - set of Parikh images of words from L
  - $\Pi(L)=\{\pi, \exists w \in L\ \Pi(w)= \pi\}$
- Parikh image omits the order of symbols

Decision procedure

- $w_1^i \ldots w_n^k \in L(G) \cap w_1^* \ldots w_n^* \Leftrightarrow a_1^i \ldots a_n^k \in L(G')$
  - a_p are distinct, fit on their position by construction of G'
  - $\pi \in \Pi(G') \Leftrightarrow a_1^{\pi(1)} \ldots a_n^{\pi(k)} \in L(G')$
- Parikh image of a context free language can be characterized by an existential Presburger formula
  - $\Psi_{G'}(\pi) = \text{True} \Leftrightarrow \pi \in \Pi(G')$
- Satisfiability of existential formula is NP-complete
  [Verma, Seidl, Schwentick: On the Complexity of Equational Horn Clauses, 2005]

Existential Presburger formula ϕ
$t :== 0 | 1 | x | t_1 + t_2 | t_1 - t_2$
$\phi :== t_1 = t_2 | t_1 > t_2 | \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x\ \phi_1$

From Language to Formula

$$L(G_{T1}) \cap w_1^{*}\, w_2^{*} \ldots w_n^{*} \cap L(G_{T2}) = \varnothing$$

$$\Leftrightarrow$$

$$L(G_{T1}') \cap L(G_{T2}') = \varnothing$$

$$\Leftrightarrow$$

$$\Pi(G_{T1}') \cap \Pi(G_{T2}') = \varnothing$$

$$\Leftrightarrow$$

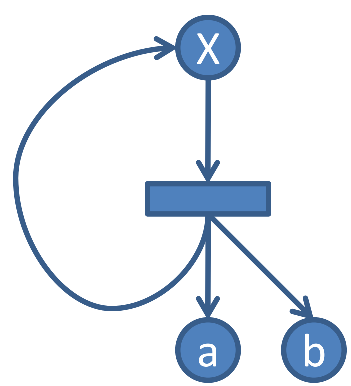
$\Psi_{T1'}$ & $\Psi_{T2'}$ is unsatisfiable

Construction of formula

- Petri-net intuition
  - net is simulating the grammar but disregards the ordering of terminals

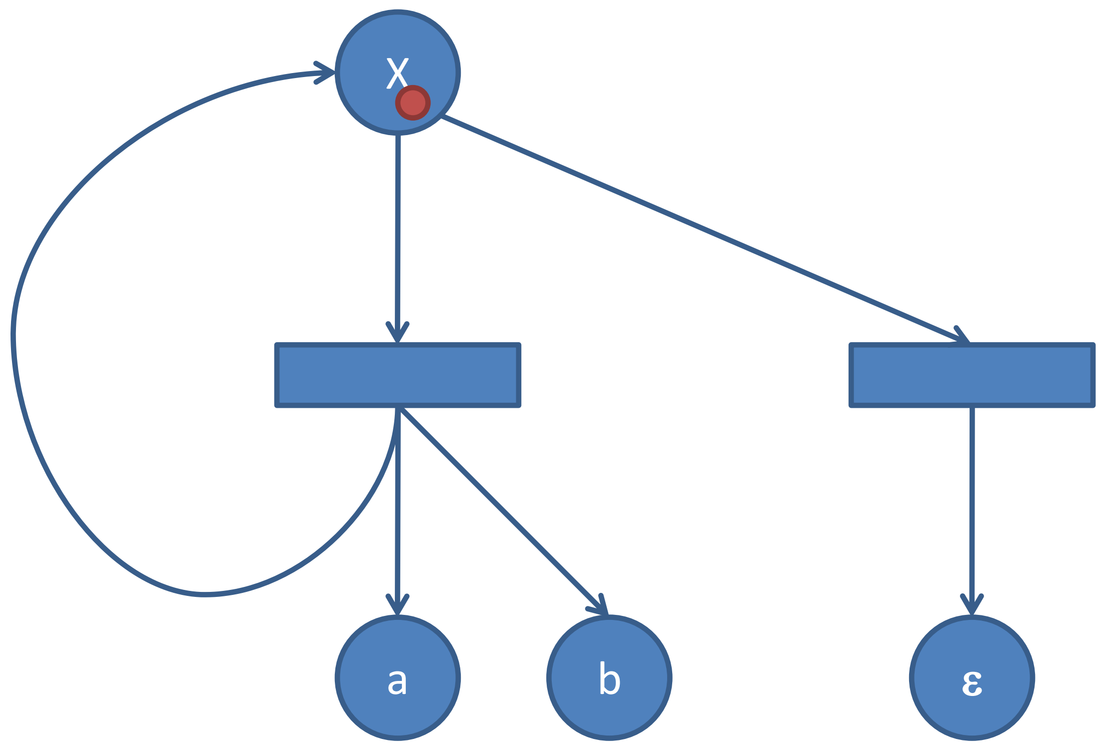
- Structure
  - Place for each terminal and non-terminal
  - Transition for each rule
  - One token to the initial non-terminal

```
X -> aXb
```



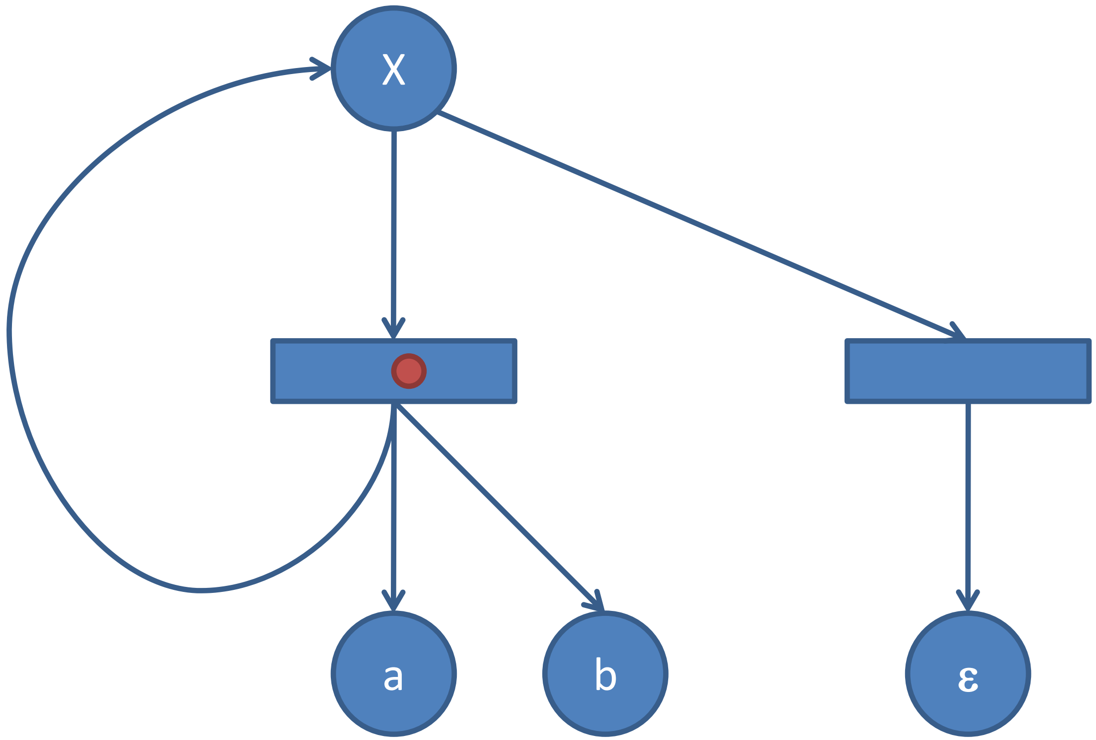
PN example

```
X -> aXb
X -> ε
```



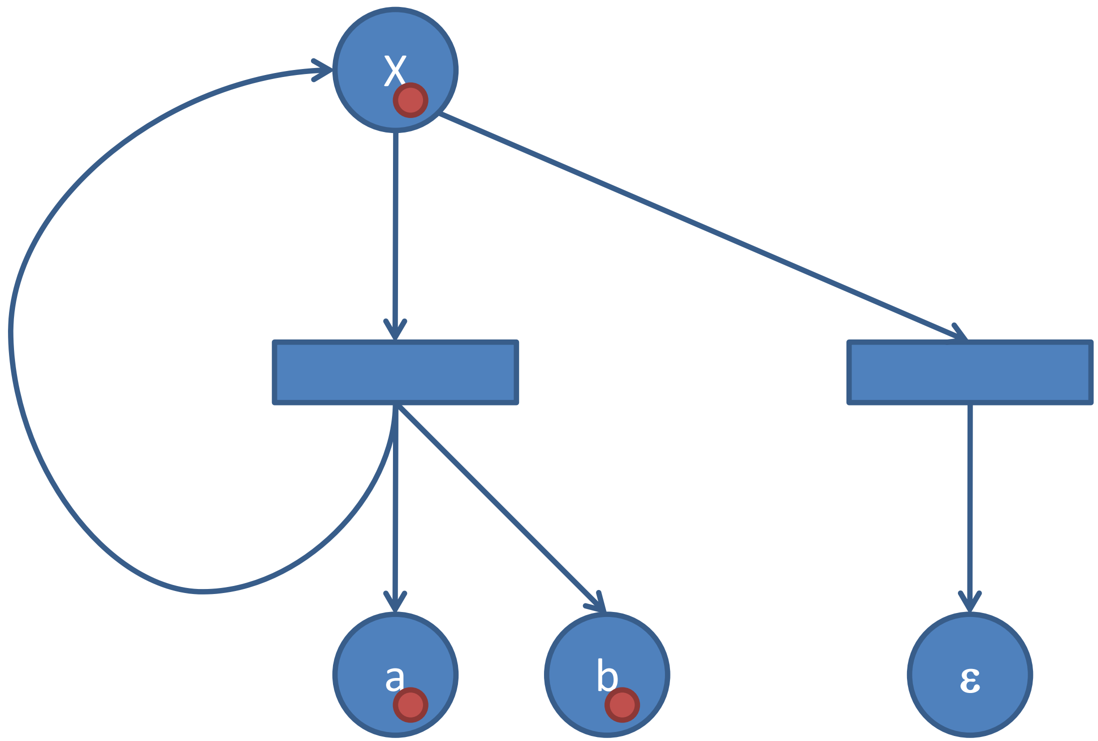
PN example

```
X -> aXb
X -> ε
```



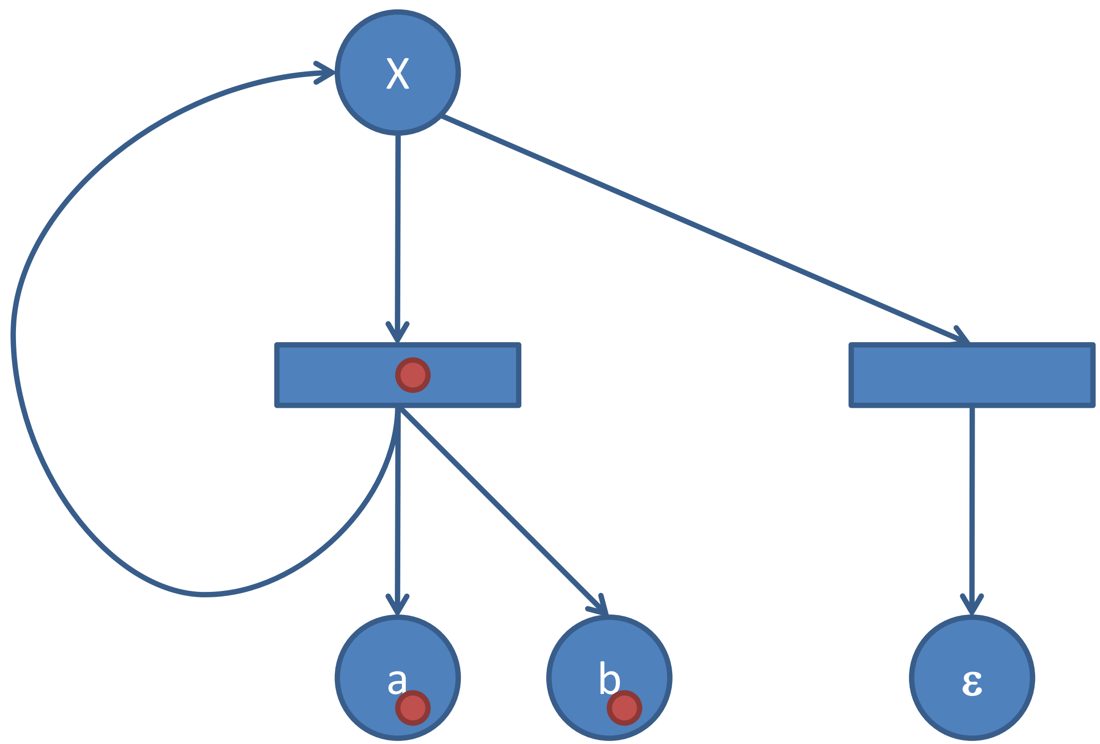
PN example

```
X -> aXb
X -> ε
```



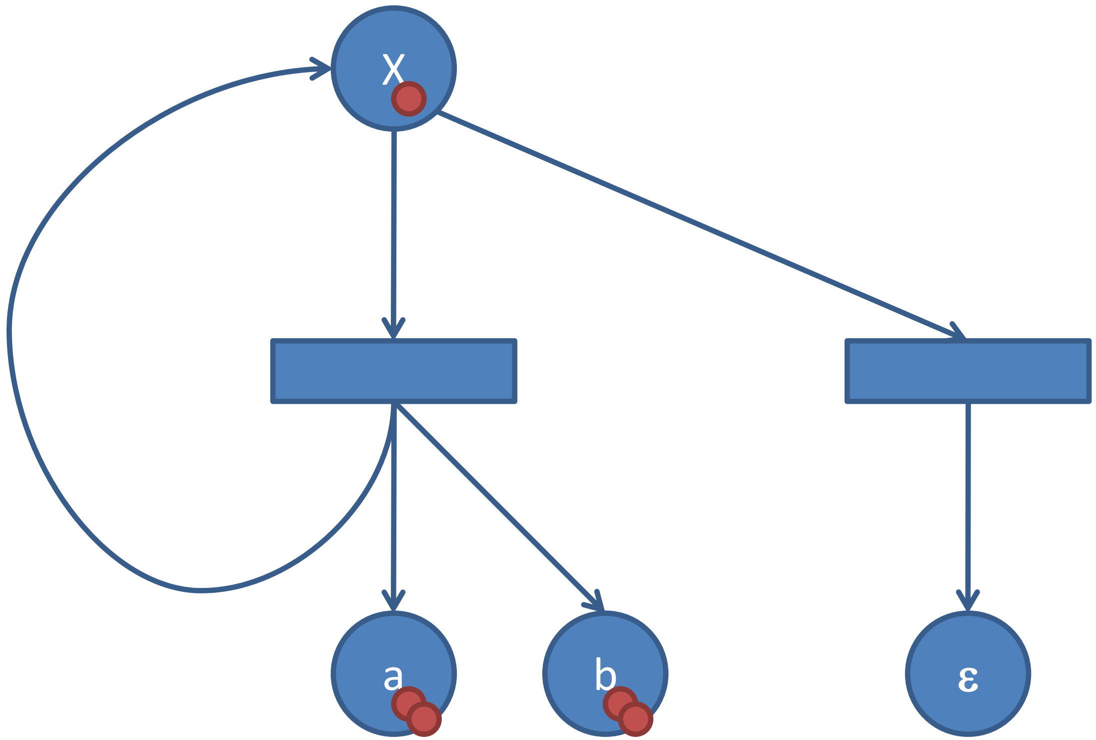
PN example

```
X -> aXb
X -> ε
```



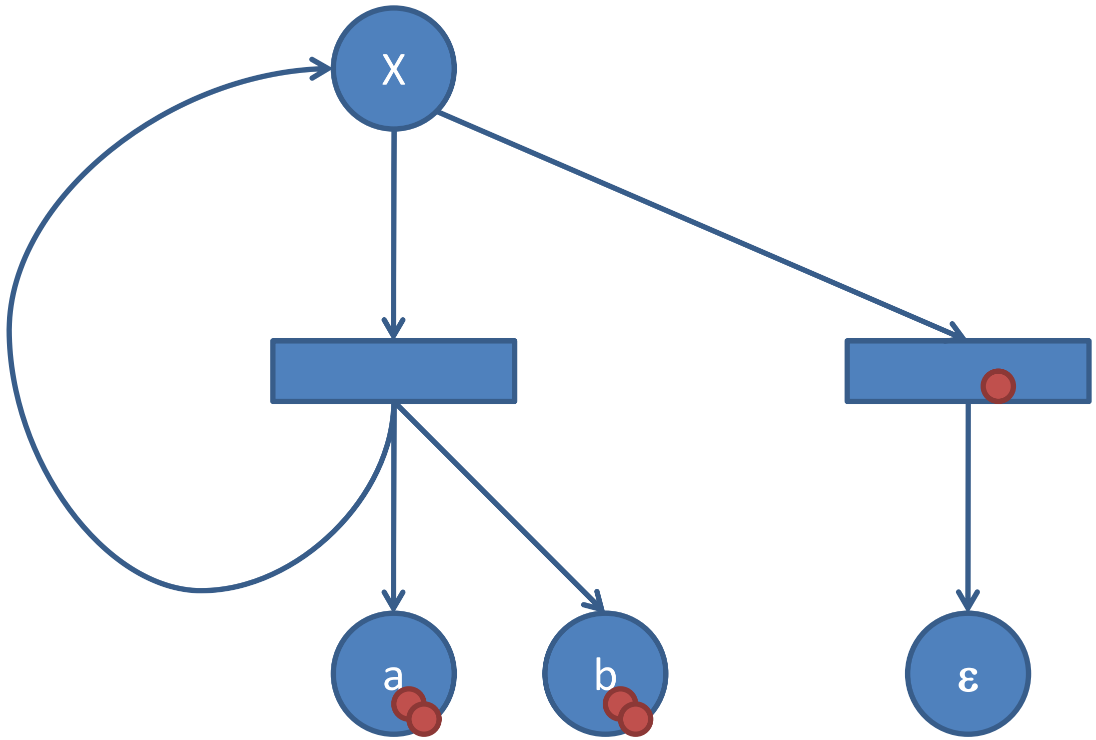
PN example

```
X -> aXb
X -> ε
```



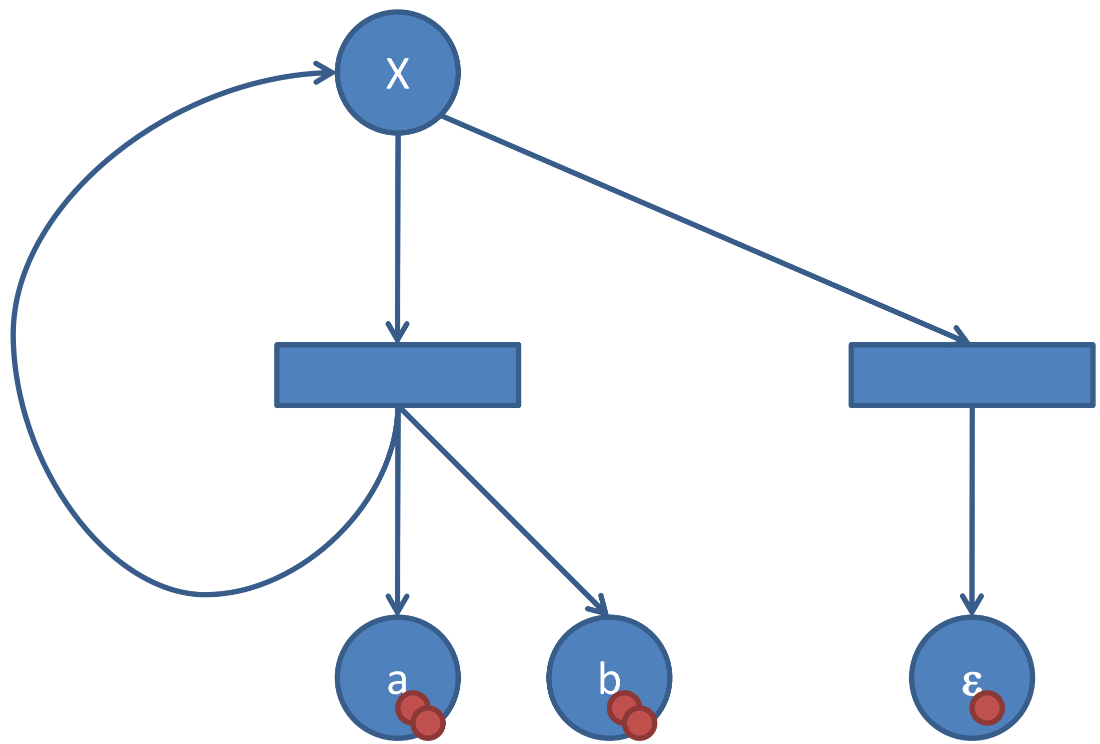
PN example

```
X  ->  aXb
X  ->  ε
```



PN example

X -> aXb
X -> ε



$L(X) = a^i b^i$ Configurations corresponding $w \in L(X)$

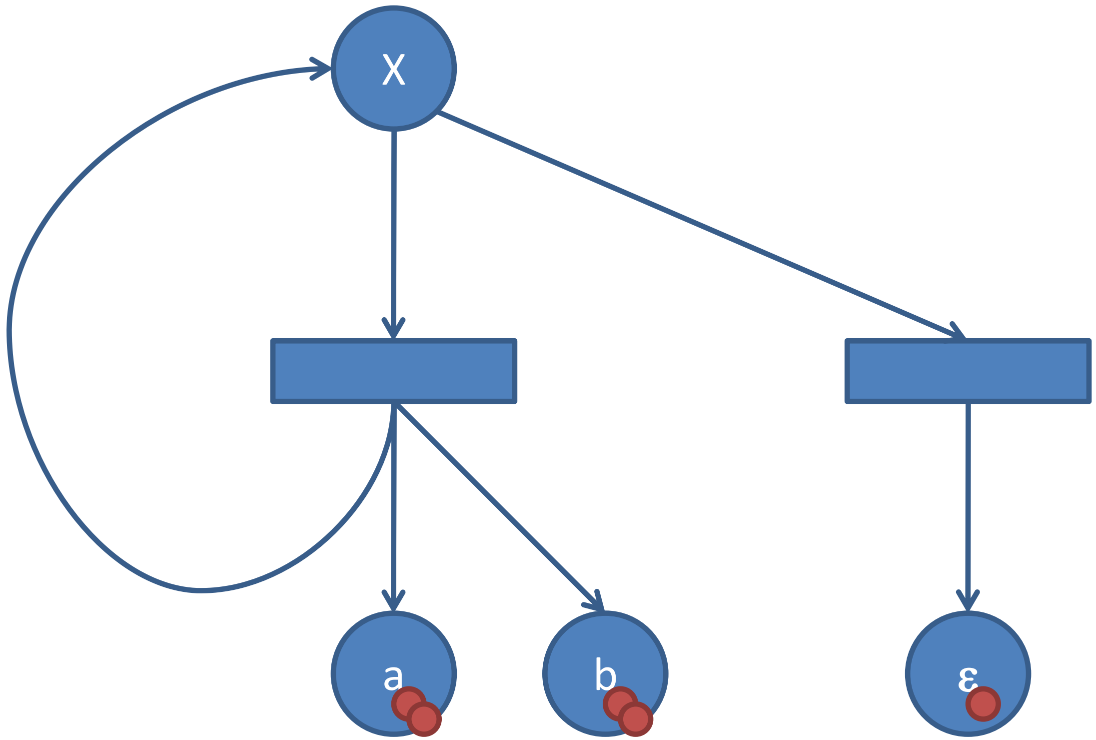
$\Pi(X) = \langle i,i \rangle$ → all tokens in terminal places

PN examples

```
X -> aXb
X -> ε
```

```
X -> abX
X -> ε
```

```
X -> Xba
X -> ε
```



Configurations corresponding $w \in L(X)$

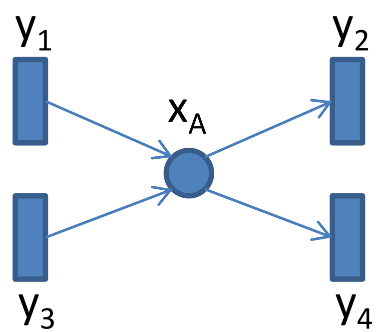
$\Pi(X) = \langle i,i \rangle$ $\rightarrow$ all tokens in terminal places

Formula

- Petri net is communication-free
  - Each transition has one input place
  - Context-free grammar (one NT on left-hand side)
- Set of admissible configurations of CF-PN can be characterized by Presburger formula

Formula

- Formula based on
  - Kirchhoff-like rules
    - For each place “# of tokens” = “# of input transition applications” - “# of output transition applications”
  - Reachability rules
    - Each applied transition is reachable from the initial place
- Variables
  - For each place A, x_A is number of tokens in the place, z_A distance from initial place
  - For each transition y_i is the number of applications



$x_A=y_1+y_3-y_2-y_4$

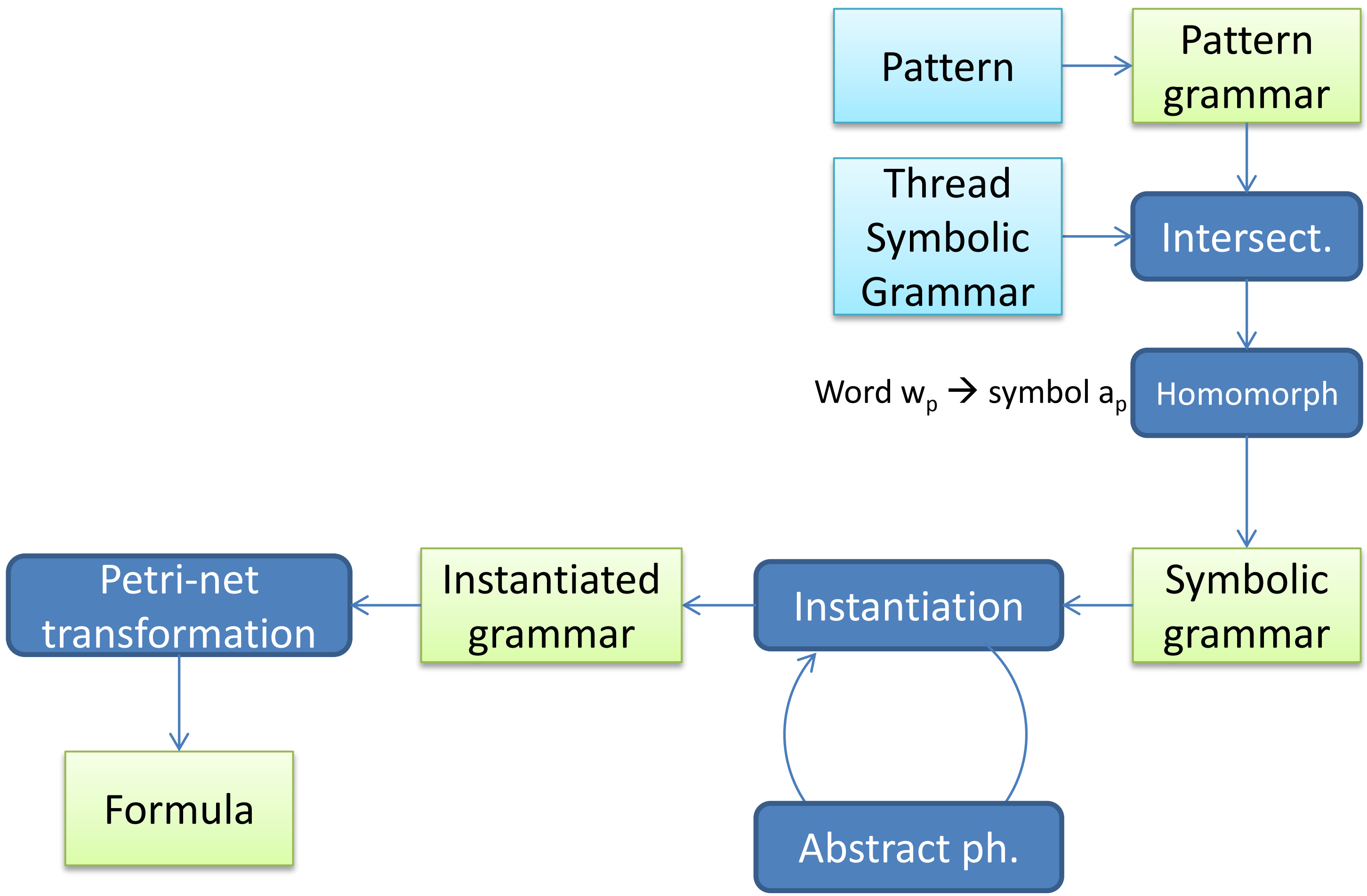
Verma, Seidl, Schwentick: On the Complexity of Equational Horn Clauses, 2005

Javier Esparza: Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes, 97

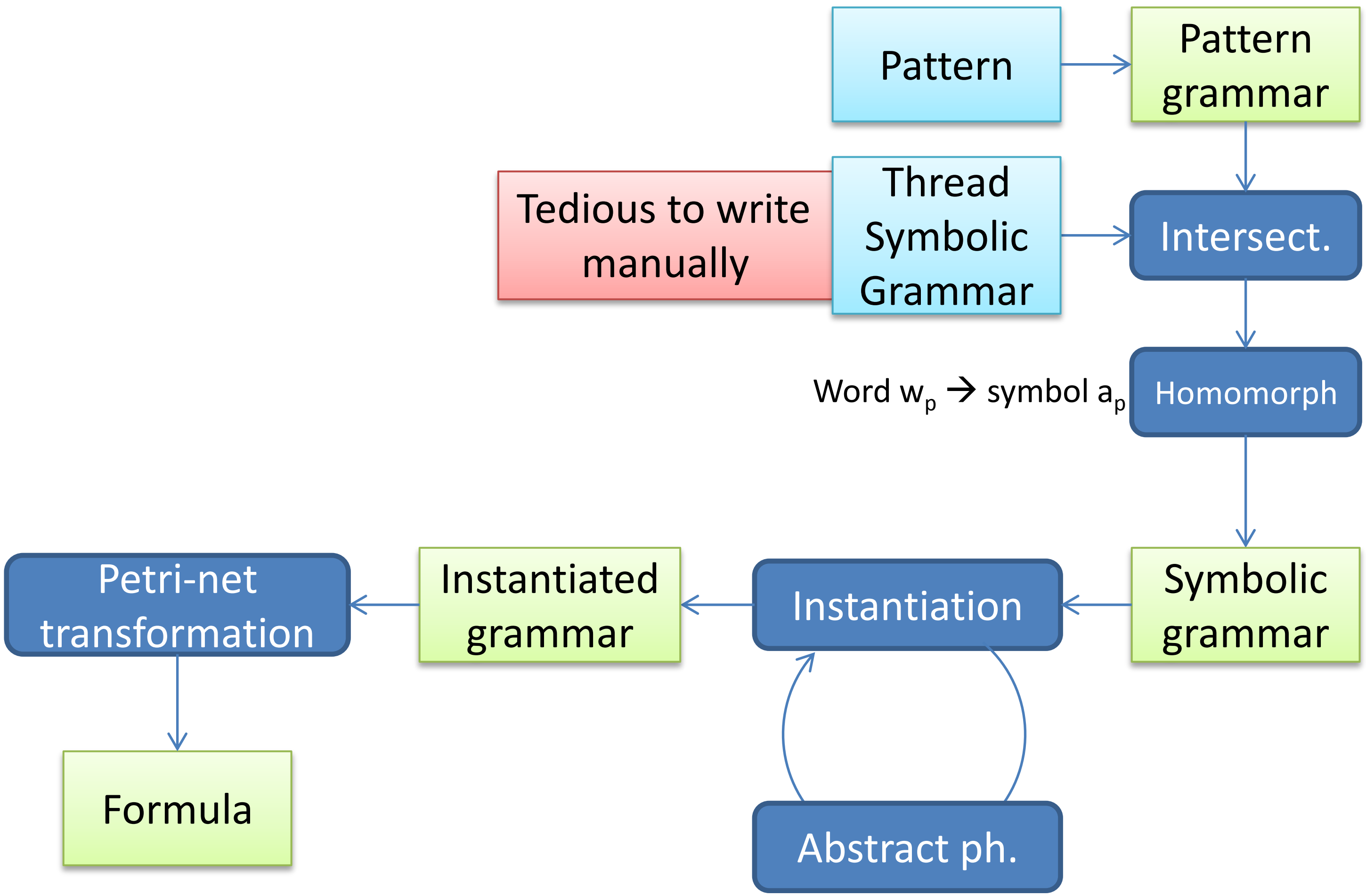
Implementation

- Input
  - Control-flow for each thread
  - Definition of global and local variables
  - Pattern
- Goal
  - Transform grammars into formula, run solver

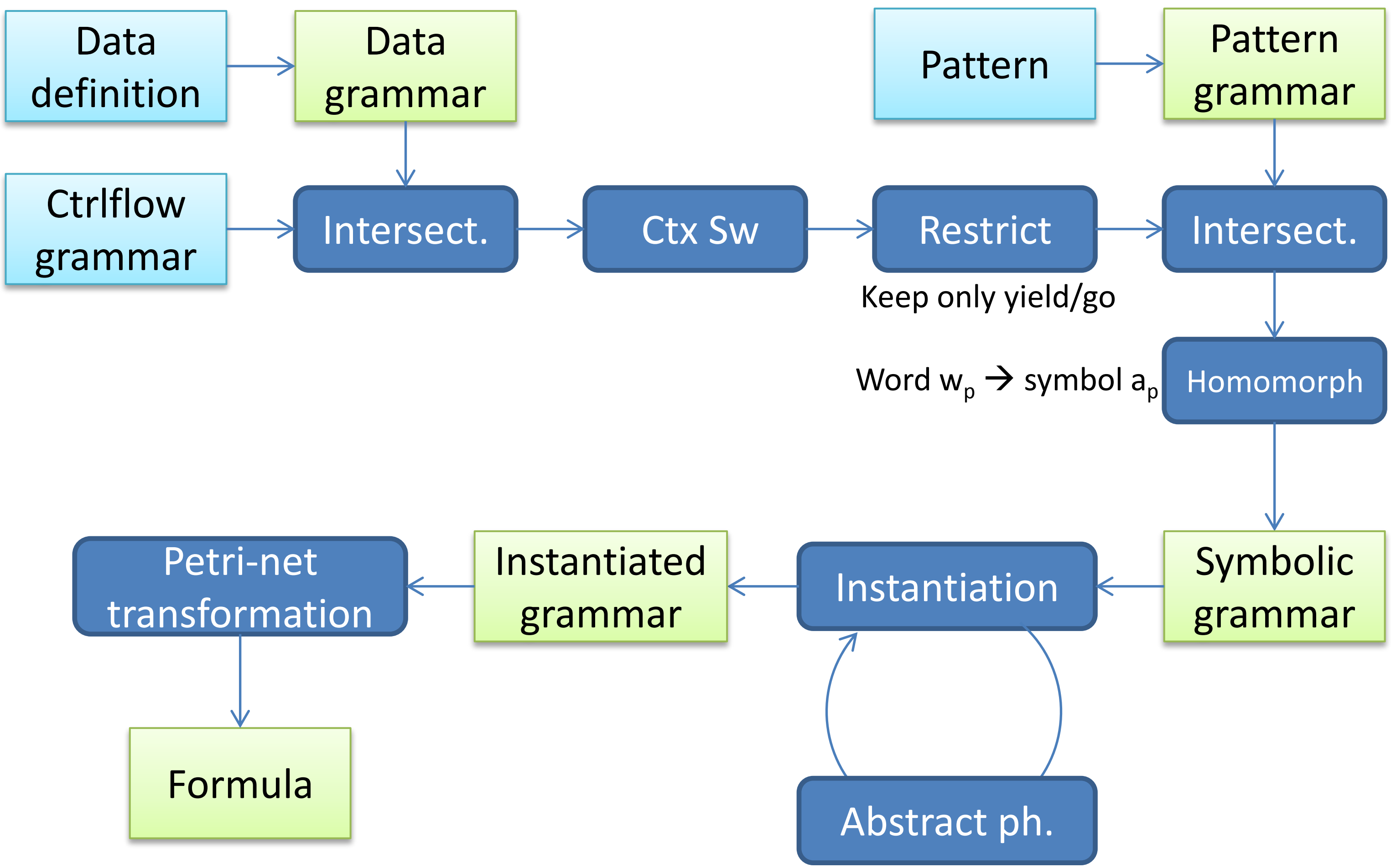
Transformation chain for thread



Transformation chain for thread



Transformation chain for thread



Input – Ctrlflow

- In form of context-free grammar
  - Non-terminals – program locations
  - Terminals – data access

Input – Ctrlflow

- In form of context-free grammar
  - Non-terminals – program locations
  - Terminals – data test/operations

```
L0  bit=F;

L1  if bit==T

L2    goto 4;

L3  return;

L4  print “busted”
```

Input – Ctrlflow

- In form of context-free grammar
  - Non-terminals – program locations
  - Terminals – data test/operations

```
L0   bit=F;                   L0 -> <bit=F> L1

L1   if bit==T                L1 -> <bit==T> L2
                              L1 -> <bit==F> L3
L2     goto 4;                L2 -> L4

L3   return;                  L3 -> ε

L4   print “busted”           L4  print “busted”
```

Input – data definition

- Used to generate data grammar

- Data grammar
  - Non-terminals – data value
  - Terminals – data access
    - The same terminals are in control flow grammar
  - Regular rules, symbolic

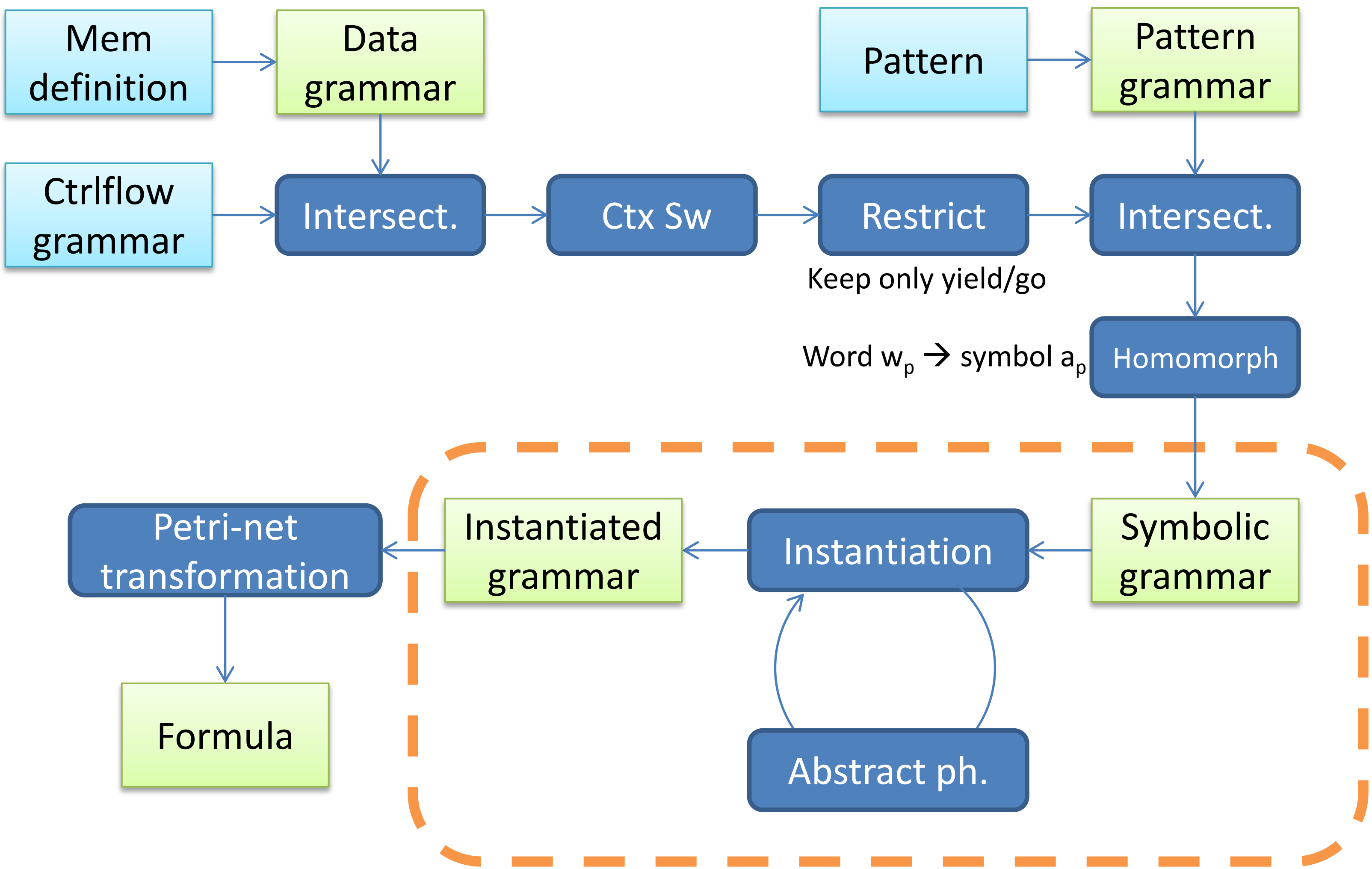
```
[x=C,y=D] -> <x==C>[x=C,y=D]
```

- Generated language ~ valid memory behavior
  - <x=4> can be followed by <x==4> but not <x==5>
  - Intersected with the control flow grammar to provide semantics

Input - pattern

- Pattern expression
  - List of words (yield/go)
  - Transformed into regular grammar

Transformation chain for thread



Instantiation

- Non-terminals in symbolic rules use variables
  - One symbolic rule stands for number of ‘ground’ rules

[pc=1,x=?] -> [pc=2,x’=x+1]

[pc=1,x=0] -> [pc=2,x’=1]

[pc=1,x=1] -> [pc=2,x’=2]

⋮

[pc=1,x=254] -> [pc=2,x’=255]

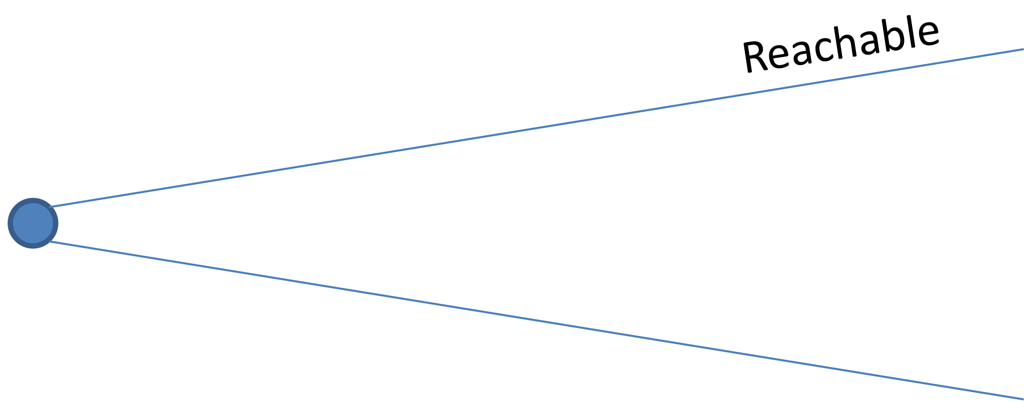
- Petri-net transformation can not process symbolic rules
  - Formula needs to have fixed number of variables (~nonterminals,rules)

Instantiation

- Goal – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- Algorithm
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewriten to sequence of terminals)
  - Transitive closure

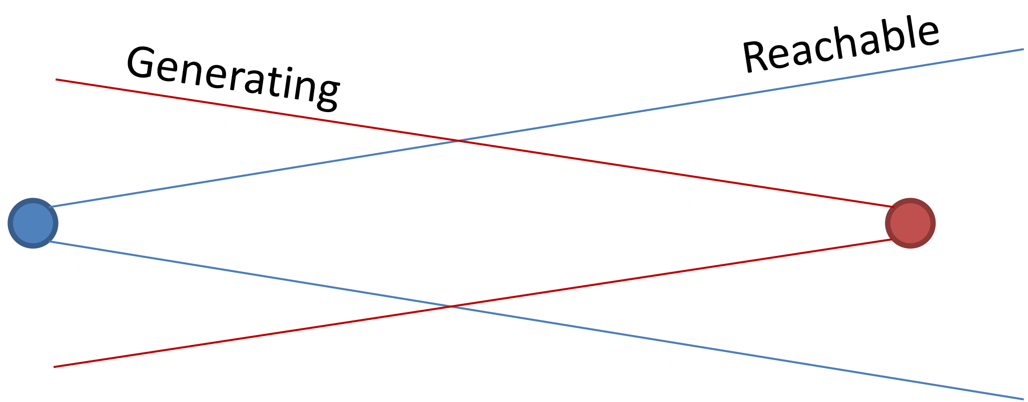
Instantiation

- Goal – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- Algorithm
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewriten to sequence of terminals)
  - Transitive closure



Instantiation

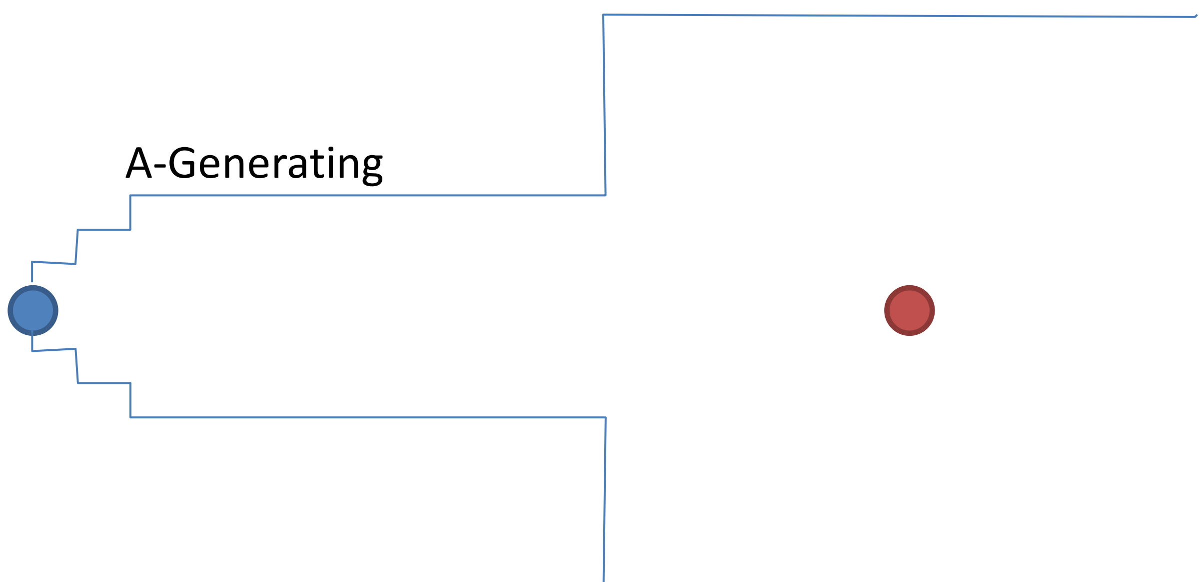
- Goal – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- Algorithm
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewriten to sequence of terminals)
  - Transitive closure



Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

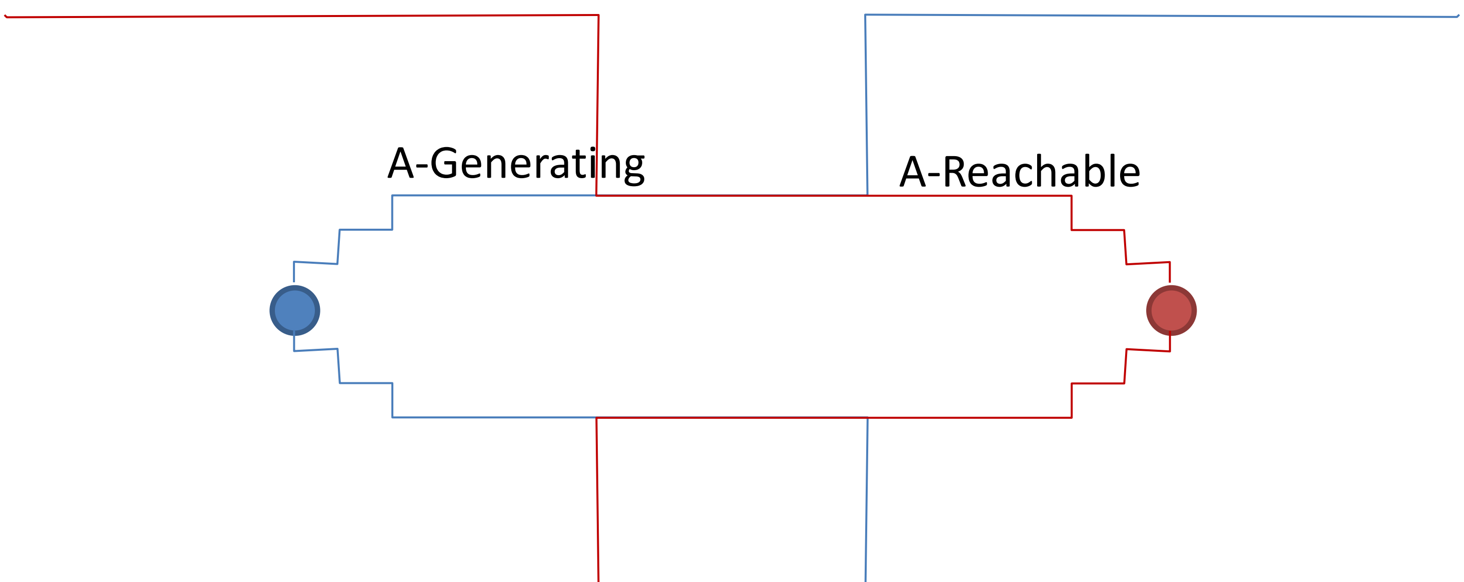
- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result



Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

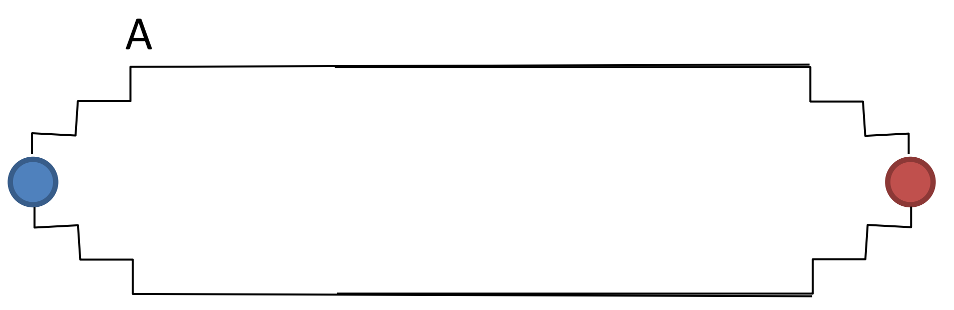
- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result



Instantiation

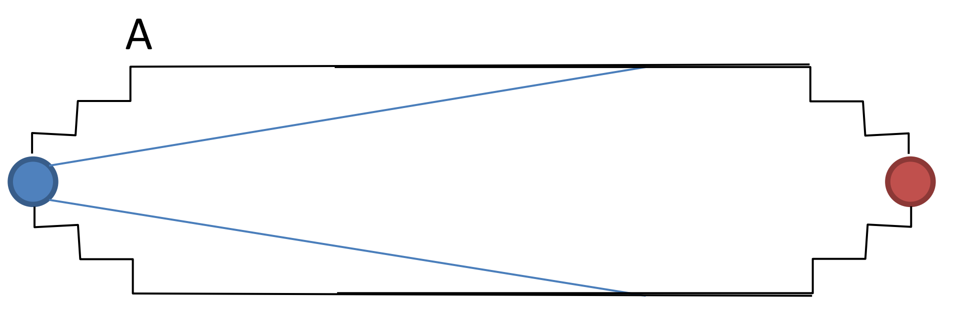
- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result



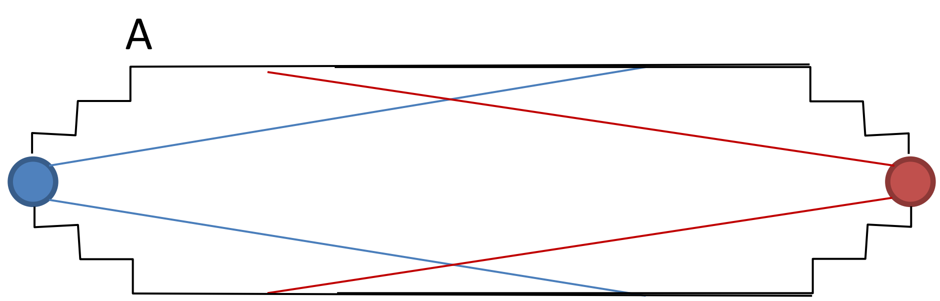
Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small
- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result

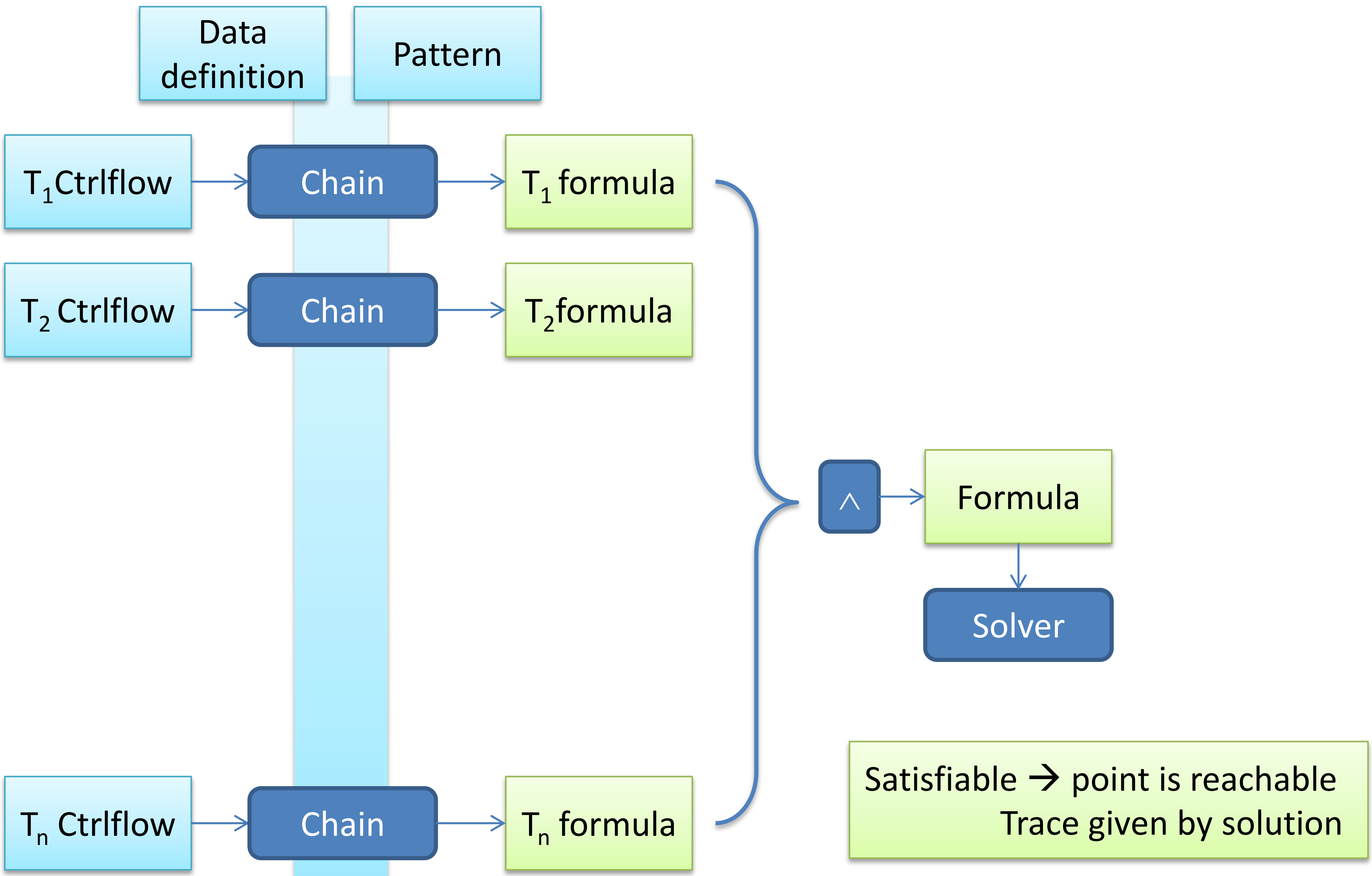


Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small
- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result



Transformation chain



Validation

- Windows NT bluetooth driver example
  - Several variants, race conditions reported in

    Suwimonteerabuth, Esparza, Schwoon: Symbolic Context-Bounded Analysis of Multithreaded Java Programs, SPIN '08
  - We can detect them all, given the proper pattern
- Still toy example
  - Input simplified to preserve essence of the bug

Task	Formula clauses	Transform Time[s]	Yices Time[s]
bt1	1074	62	1
bt2	6664	1003	1
bt2	2924	816	1
cavpp	5840	12	236

The size of data bothers the transformation, length of trace bothers yices

Abstraction phase helps

Thread	A-Gener [abst. NT]	A [abst. NT]	Gener [NT]	Reach [NT]	Total Time [s]
bt1/add	skip	skip	465822	326	438
bt1/add	46005	251	2394	326	26
bt3/add	150022	922	7445	787	280
bt3/stop	294773	520	3557	48	257

- Two variants of BT example runs out of memory if the abstraction phase is off

Conclusion

- Theory works
  - Bluetooth example is small, but real
- Tool runs
  - Lots of technical details solved
  - Provides result for all toy examples we have
- Future directions
  - More, larger, examples
  - Instantiation phase
    - Skip it – push the instantiation phase into formula
    - Smarter approximation (e.g. abstract interpretation)

Thank you