Implementare model-checkers con fixed-point calculus (fondamenti tool Getafix)

Implementing a model-checker

- In standard programming languages (e.g., C or Java) it is a complex task
  - memory management
  - caching management
  - variable ordering (BDD)
  - .....
- A typical model checker spans over thousand lines of code
- Small changes in the algorithms may require redesigning large portions of code
  - usually hard to try new ideas

Topics

- A framework to implement model-checking algorithms for Boolean programs (tool GETAFIX)

- Efficient
  - competitive with mature model-checkers
- Easy
  - to implement and debug
  - to experiment with variants
- Amenable to “theory people”
  - high level formalism (fixed-point calculus)
  - hide details unrelated to the algorithmic aspects of solutions

How model-checkers look in our formalism

```
mu bool Reachable( Module s_mod, PrCount s_p
s_mod <  s_pc, s_pc  <  s_CL, s_CL  ~+ s_ENTR
(  ( exists Module  t_mod, PrCount t_pc, Local t_(
       (    target(t_mod,t_pc) &  Reachable(t_mod,

  |(enforce(s_mod, s_CL, s_CG)  &    (
    (s_mod=0 & s_pc=0 )
    | (        s_pc=0 & CopyLocals(s_mod,s_ENTRY
            & (exists Module t_mod, PrCount t_pc, L
              (   (Reachable(t_mod,t_pc,t_CL,t_CG,t_
                    & CopyGlobals(s_mod, t_CG, s_EN

    | (exists PrCount t_pc, Local t_CL, Global t_CG.
       ( (Reachable(s_mod,t_pc,t_CL,t_CG,s_ENTR
         &( programInt(s_mod,t_pc,s_pc,t_CL,s_CL

    | (exists PrCount t_pc, Global t_CG, Module u_
          ( exists  Local t_CL. (  (Reachable(s_mod
            & SkipCall(s_mod,t_pc,s_pc)) & prograr
            & SetReturnTS(s_mod,u_mod,t_pc,u_p
        & ( exists Local u_CL, Global u_CG.
          ( (Reachable(u_mod,u_pc,u_CL,u_CG,u_
            & SetReturnUS(s_mod,u_mod,t_pc,u_p
)));

(  exists Module  s_mod, PrCount s_pc, Local s_(
       (    target(s_mod,s_pc) &    Reachable(s_mo
```

- Checks whether an error state in Boolean program is reachable.
- Entire model checking algorithm in 1~2 pages !
- Symbolic algorithm that uses BDDs
- Competitive with mature model-checkers

- High-level declarative algorithm at one level; but also algorithmic aspects encoded!
- Highly readable ; easily debuggable
- Easy to experiment with variants
- Underlying solver GETAFIX will convert the program and run this algorithm efficiently

Why fixed-point calculus?

- Natural formalism in verification
  - Most symbolic model-checking algorithms essentially compute fixed-points
  - Ex: compute the least X s.t $X = Start \vee X \vee Succ(X)$

    (forward reachability)

- Right primitives
  - easy encoding of model-checking algorithms
  - sufficiently low-level to express algorithmic details

Fixed-point calculus

Quantified Boolean logic

First-order logic over the Boolean domain

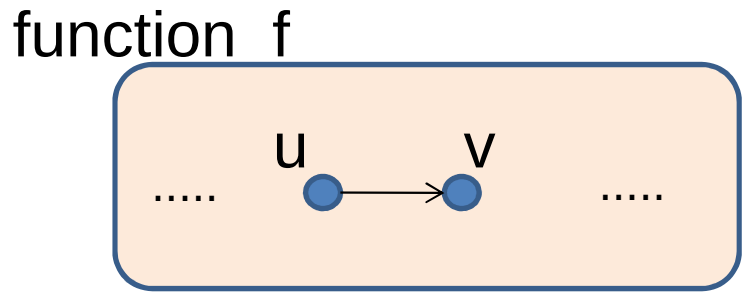
BoolExp B ::= T | F | $R_k(x_1, \ldots, x_k)$ | $\neg$ B |
B $\wedge$ B | B $\vee$ B | $\exists x.$(B) | $\forall x.$(B)

Variables interpreted over the Boolean domain {T, F}

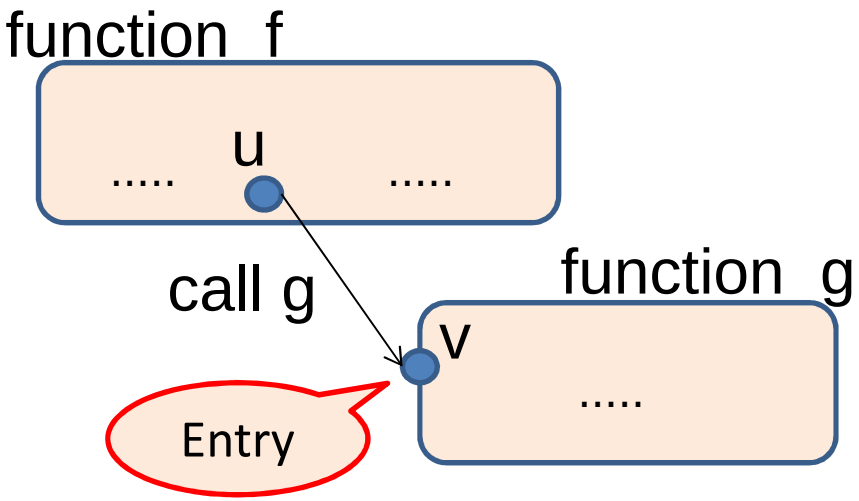
Relations interpreted as k-ary tuples of {T,F}

Some relations defined by pgms

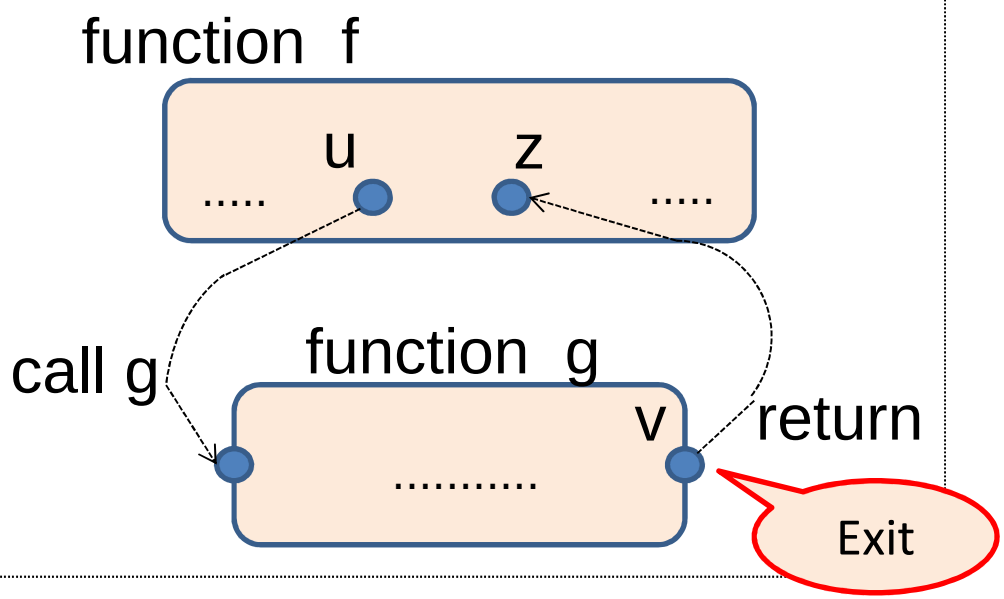
- Internal(u,v)



- Call(u,v)



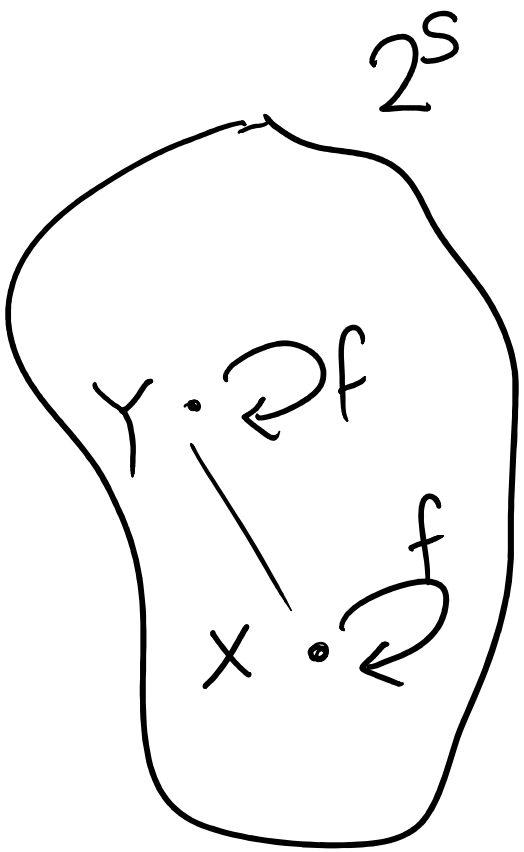
- Return(u,v,z)



Fixed-points

Consider the complete lattice $(2^S, \subseteq)$

Consider a function $f : 2^S \rightarrow 2^S$



- $X \in 2^S$ is a fixed point of f
  if $f(X)=X$

- A fixed point X of f is the *least fixed point*
  if for every fixed point Y, $Y \supseteq X$

Tarski-Knaster theorem

Thm.

Let L be a complete lattice and let f : L → L be a monotonic function.

Then the set of fixed points of f in L is also a complete lattice.

(The theorem was shown just for the powerset lattice earlier by Tarski)

Tarski theorem applied to $(2^S, \subseteq)$

Consider a function $f : 2^S \longrightarrow 2^S$ that is monotonic, i.e.:

$X \subseteq Y$ then $f(X) \subseteq f(Y)$

Consider the sequence

$X_0 = \emptyset$

$X_1 = f(X_0)$

$X_2 = f(X_1)$ …

This eventually converges to the least-fixed point

(and hence always exists).

Es.: Reachability in non-rec programs

Reach(pc, x)

= (pc=0 ∧ Init(x))

∨ ∃pc',y. (Reach(pc', y) ∧ Internal(pc',y, pc,x))

Declarative: Reach is the smallest set that contains the initial state and is closed under internal image.

Operational/Algorithmic:

Start with the empty relation; keep applying Reach to the current definition till the least fixed-point is reached

Note: Init and Internal are relations that are defined by the program being checked. Algorithm uses these relations.

A fixed-point calculus

- System of equations

  f_1 (x_1, …, x_r) = PosBoolExp using f_1,… f_m

  f_2 (y_1, …, y_s) = PosBoolExp using f_1,… f_m

  ⋮

  f_m (z_1, …, z_t) = PosBoolExp *using* f_1,… f_m

- Positive Boolean expression: relations do not occur within an odd number of negations
- A positive Boolean expression defines a monotonic function.
- Least fixed-point of all these relations f_1,… f_m exists.

Computation of the least fixed-points

- Least fixed-point of $f_1,\ldots, f_m$ can be computed by an algorithm that starts with the empty relation, and computes iteratively till a fixed-point is reached

Let f=B be an equation of a system Eq

Evaluate(*f,Eq) :*

Set *S := ∅*

while (*S does not*

- *Eq' := Eq \ f =*
- *Eq'':=Eq' [f ←*
- *Let f_1, ...,f_r o*
- *S:= evaluatio*

}

return S

- Evaluate gives operational semantics to our algorithms
  - even when general (non positive) Boolean expressions are used (in this case we are in charge to ensure convergence)

Recursive Boolean programs

(sequential) Boolean programs

- Nondeterministic procedural programs
- Has conditionals, iteration, recursive function calls
- Variables range over the Boolean domain {True, False}
- And specific statements for:
  - function invariants (enforce)
  - assertions (assume)
  - constrained assignments (constrain)

A Boolean program

```
main () {
    bool a,b;
    a := T;
    b := *;
    while (b) {
          a:= f(b);
             b := * constrain (!(a && b'));
    }
    if (c) {
       ERR: skip
    }
}
```

```
bool   f( bool x ) {
        bool y;
        y := *;
        while (y) {
              y:= f(x)
        }
        return y;
}
```

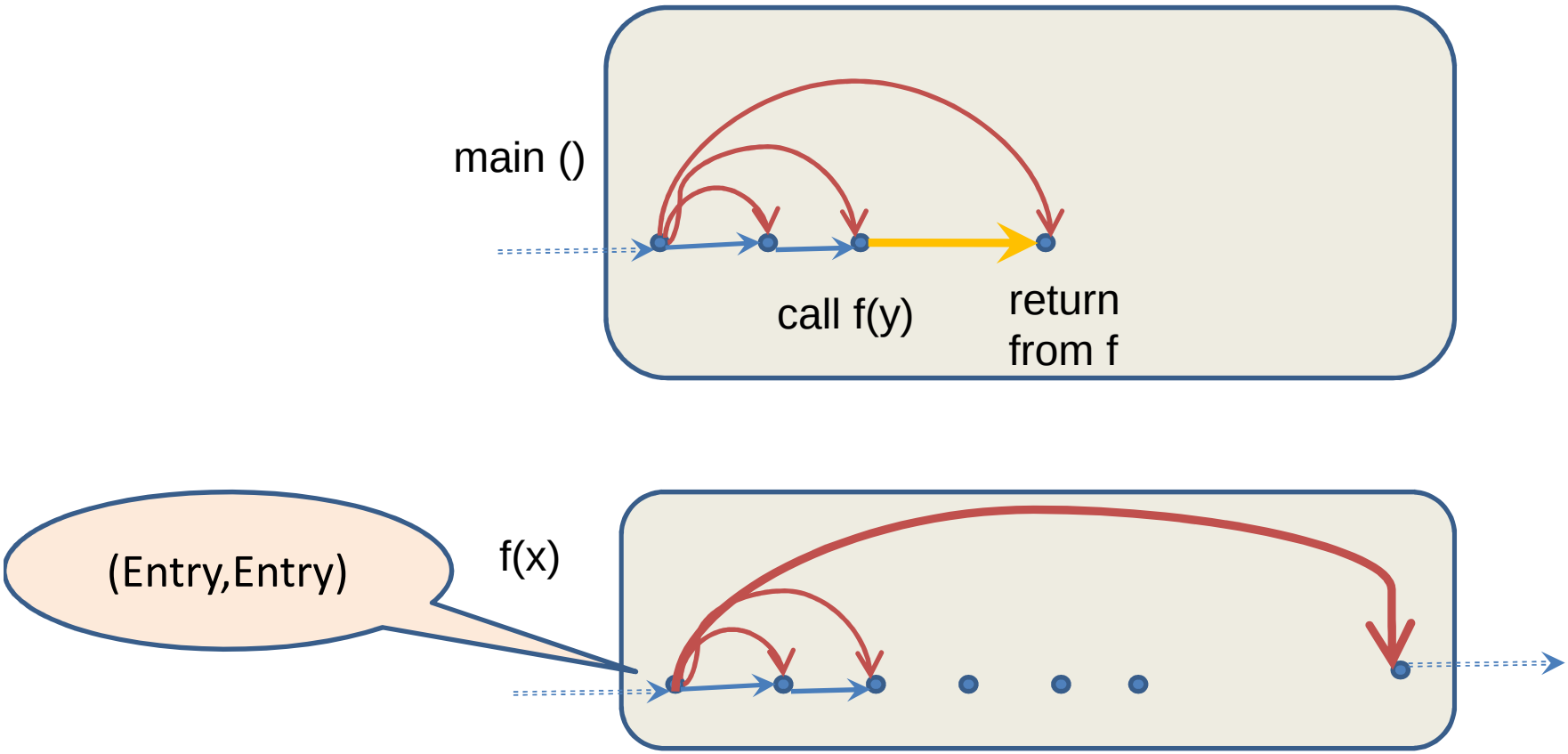
Nondeterminism:

"b is either T or F"

Constrained assignment

"b is F" or "a is F"

Checking recursive Boolean programs



Summaries: (Entry, State)

meaning the state is reachable from the entry of the function (perhaps using calls to other functions)

Writing the algorithm

$$Summary(u,v) =$$

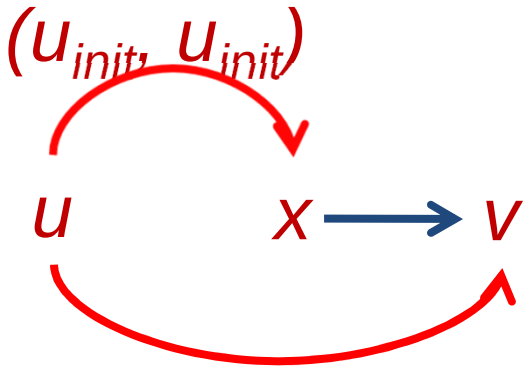
$$(Entry(u.pc) \wedge u=v \wedge Init(u.pc))$$

$$\vee \exists x.(Summary(u, x) \wedge Internal(x, v))$$

$$\vee \exists x, y.(Summary(x, y) \wedge Call(y, u) \wedge u=v)$$

$$\vee \exists x, y, z.(Summary(u, x) \wedge Call(x, y)$$

$$\wedge Summary(y, z) \wedge Exit(z.pc) \wedge Return(x, z, v) \quad)$$



Writing the algorithm

Summary(u,v) =

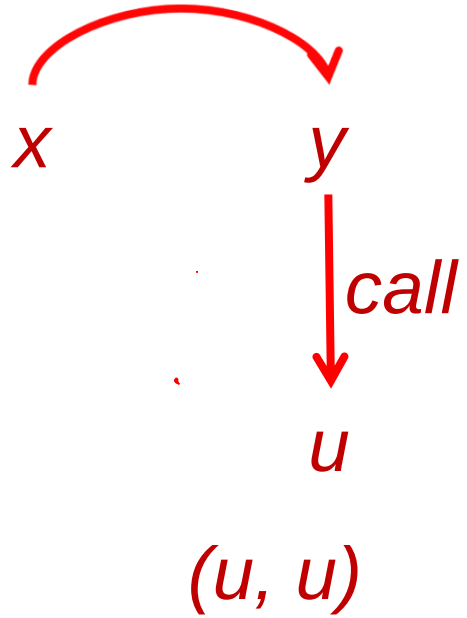
(Entry(u.pc) ∧ u=v ∧ Init(u.pc))

∨ ∃x.(Summary(u, x) ∧ Internal(x, v))

∨ ∃x, y.(Summary(x, y) ∧ Call(y, u) ∧ u=v)

∨ ∃x, y, z.(Summary(u, x)∧ Call(x, y)

∧ Summary(y, z) ∧ Exit(z.pc) ∧ Return(x, z, v))



Writing the algorithm

Summary(u,v) =

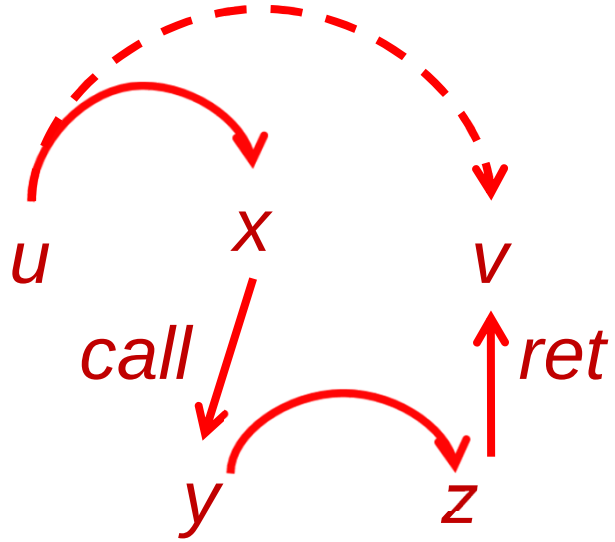
(Entry(u.pc) ∧ u=v ∧ Init(u.pc))

∨ ∃x.(Summary(u, x) ∧ Internal(x, v))

∨ ∃x, y.(Summary(x, y) ∧ Call(y, u) ∧ u=v)

∨ ∃x, y, z.(Summary(u, x)∧ Call(x, y)

∧ Summary(y, z) ∧ Exit(z.pc) ∧ Return(x, z, v))



Actual code

```
mu bool Reachable( Module s_mod, PrCount s_pc, Local   s_CL, Global  s_CG, Local   s_ENTRY_CL, Global
s_ENTRY_CG)
s_mod <  s_pc, s_pc  <  s_CL, s_CL  ~+ s_ENTRY_CL, s_CL  <  s_CG, s_CG  ~+ s_ENTRY_CG  /* BDD ordering */
(  ( exists Module  t_mod, PrCount t_pc, Local t_CL, Global t_CG, Local t_ENTRY_CL, Global  t_ENTRY_CG.
       (    target(t_mod,t_pc) &  Reachable(t_mod,t_pc,t_CL,t_CG,t_ENTRY_CL,t_ENTRY_CG)    )   )

  |(enforce(s_mod, s_CL, s_CG)  &    (
    (s_mod=0 & s_pc=0 )
    | (       s_pc=0 & CopyLocals(s_mod,s_ENTRY_CL,s_CL)
            & (exists Module t_mod, PrCount t_pc, Local t_CL, Global t_CG, Local t_ENTRY_CL, Global t_ENTRY_CG
              (   (Reachable(t_mod,t_pc,t_CL,t_CG,t_ENTRY
                    & CopyGlobals(s_mod, t_CG, s_ENTRY_C

    | (exists PrCount t_pc, Local t_CL, Global t_CG.
       ( (Reachable(s_mod,t_pc,t_CL,t_CG,s_ENTRY_CL
         &( programInt(s_mod,t_pc,s_pc,t_CL,s_CL,t_CG,

    | (exists PrCount t_pc, Global t_CG, Module u_mod, I
          ( exists  Local t_CL. (  (Reachable(s_mod,t_pc,t_
            & SkipCall(s_mod,t_pc,s_pc)) & programCall(s
            & SetReturnTS(s_mod,u_mod,t_pc,u_pc,t_CL,
        & ( exists Local u_CL, Global u_CG.
          ( (Reachable(u_mod,u_pc,u_CL,u_CG,u_ENTR
            & SetReturnUS(s_mod,u_mod,t_pc,u_pc,u_CL
)));

(  exists Module  s_mod, PrCount s_pc, Local s_CL, Glo
```

Code is executed according to the fixed-point algorithm

- *bracketing is respected in the evaluations*

Proper bracketing can reduce the number of variables in the intermediate BDDs

Correctness of the algorithm

- Let P be a recursive Boolean program
- For each pair (u,v) which is added to Summary
  - u is either an initial state or a reachable entry
  - v is either u or (u,x) is in Summary and v is reachable from x by an internal move or by a call which returns

  (only reachable states are added to Summary)
- Each reachable state is eventually added to Summary
- Theorem.

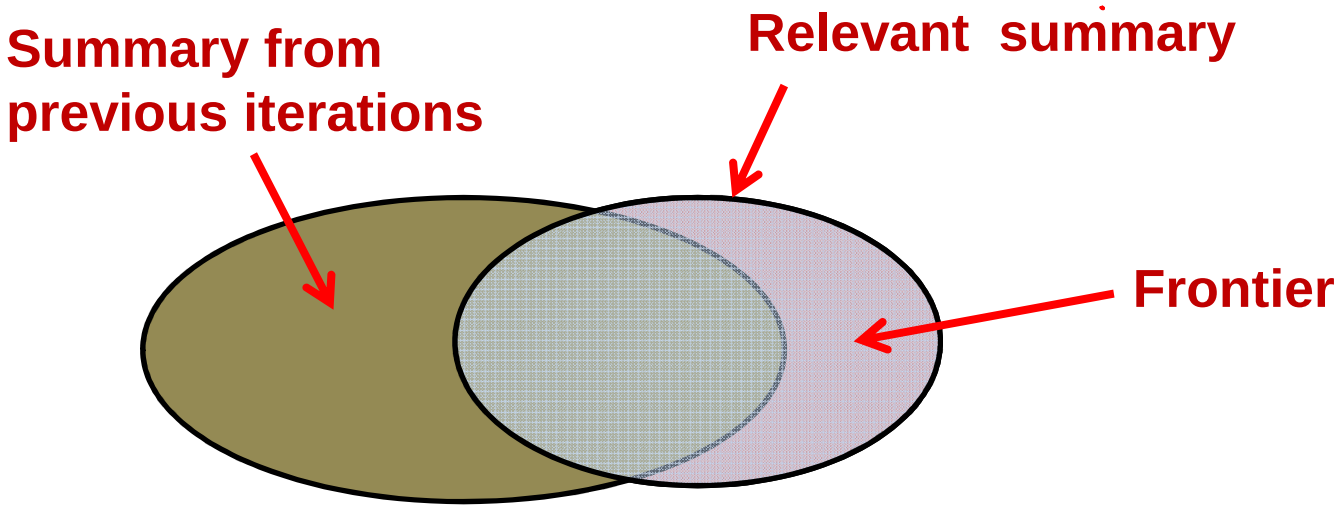
  A state v is reachable in P if and only if

  $\exists$ an entry u such that Summary(u,v)

A simpler summary-based algorithm

- Modify Summary such that
  - clause $\exists x, y.(Summary(x, y) \wedge Call(y, u) \wedge u=v)$ is deleted
  - clause $(Entry(u.pc) \wedge u=v \wedge Init(u.pc))$ is replaced by $(Entry(u.pc) \wedge u=v)$
- Still answers reachability
- May also compute unreachable states

An optimized algorithm



Frontier: Newly discovered summaries in the last round

Relevant: Summaries whose program counter value is involved in the frontier

Idea: Compute only on the relevant summaries.

An optimized algorithm

- Why not computing simply on Frontier?
  - BDDs for Frontier can be larger than reachable set
  - Relevant is a restriction of the reachable set to a particular set of program counters
- New Frontier is computed into two steps:
  - New1: image-closure of Relevant on internal transitions
  - New2: image of Relevant on calling a module or skipping the call using a summary
- Handling call and return transitions is expensive compared to internal transitions
- Programs contain many more internal than other transitions

Writing the optimized algorithm

- Summary(1,u,v) denote the computed summaries (u,v)
- Summary(0,u,v) denote the summaries (u,v) computed before the last round
  - (u,v) is in Frontier if Summary(1,u,v) and not Summary(0,u,v)

- Transitive closure on internal transitions:

$$New1(u, v) = (Summary(1, u, v) \wedge Relevant(v.pc)) \vee (\exists x.(New1(u, x) \wedge ProgramInt(x, v)))$$

Writing the optimized algorithm

$$
\begin{aligned}
New2(u, v) = \; & (\exists x.(Relevant(x.pc) \wedge Summary(1, u, x) \wedge Call(x, v))) \\
& \vee (\exists x, y, z.(Summary(u, x) \\
& \quad \wedge IntoCall(x, y) \wedge Summary(y, z) \\
& \quad \wedge Exit(z.pc) \wedge Return(x, z, v) \\
& \quad \wedge (Relevant(x.pc) \vee Relevant(z.pc))))
\end{aligned}
$$

- Compute calls and returns on relevant program counters
- Note: either the caller or the exit are required to be relevant to jump from a caller to a matching return

Writing the optimized algorithm

$$Summary(fr, u, v) = (fr=1 \wedge Entry(u.pc) \wedge u=v \wedge Init(u.pc)) \vee Summary(1, u, v) \vee (fr=1 \wedge (New1(u, v) \vee New2(u, v)))$$

- Updates the already computed summaries with the current Frontier
- Adds new summaries as Frontier

$$Relevant(pc)=\exists u, v.(Summary(1, u, v) \wedge \neg Summary(0, u, v) \wedge v.pc=pc)$$

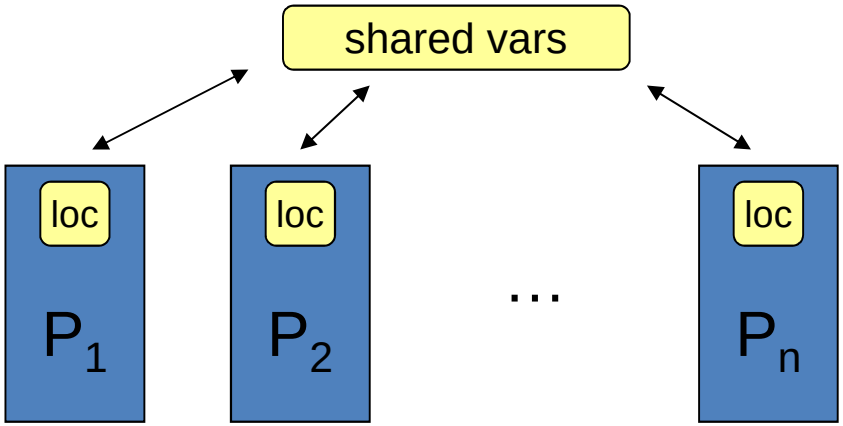
Problem

- **Relevant does not grow monotonically!**
- Tarski-Knaster theorem does not apply
- Convergence of the algorithm is up to you
  - assume the algorithmic semantics to compute the least fixed-point

Concurrent Boolean programs

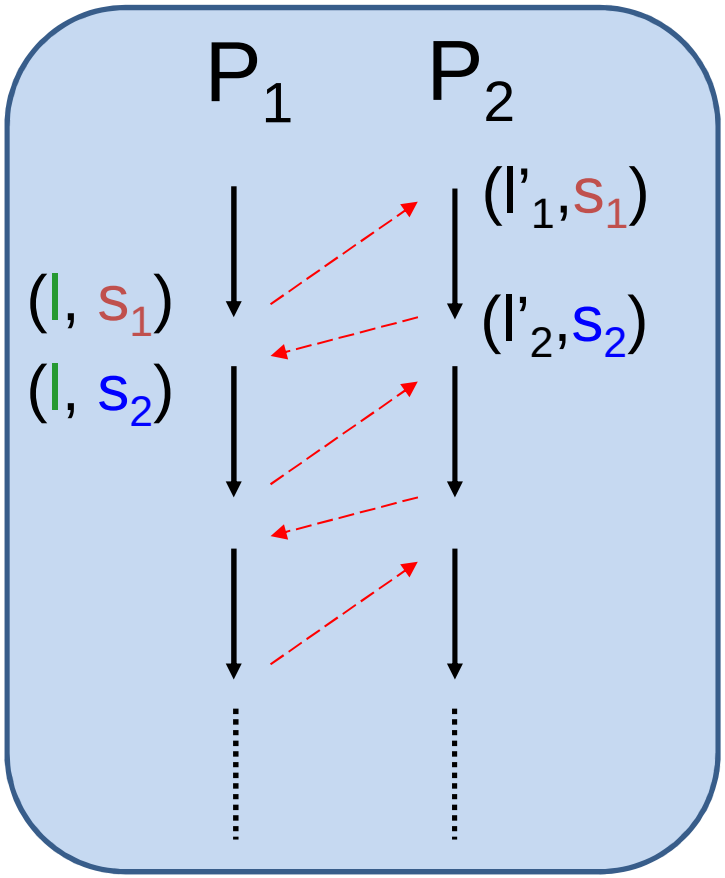
Concurrent Boolean programs

- A fixed number of recursive Boolean programs $P_1, \ldots, P_n$ (running in parallel)
- each program has its own local variables
  - local to the program
- communication is through shared variables



Concurrent Boolean programs

- Global states (l,g)
  - g is a valuation of shared variables (*shared state*)
  - l is *local state*
- Semantics by interleaving:
  - computations as sequences of execution contexts
  - only one P_i is active in each context
  - either the active P_i moves (*local behavior*)
  - or control switches to another component (*context-switch*)



Reachability of concurrent Boolean prgms

Given a concurrent Boolean program and a particular position of a component,

is that position reachable?

Problem is **UNDECIDABLE !**

(two recursive Boolean programs sharing finitely many Boolean variables can simulate a Turing machine)

Bounded context-switching

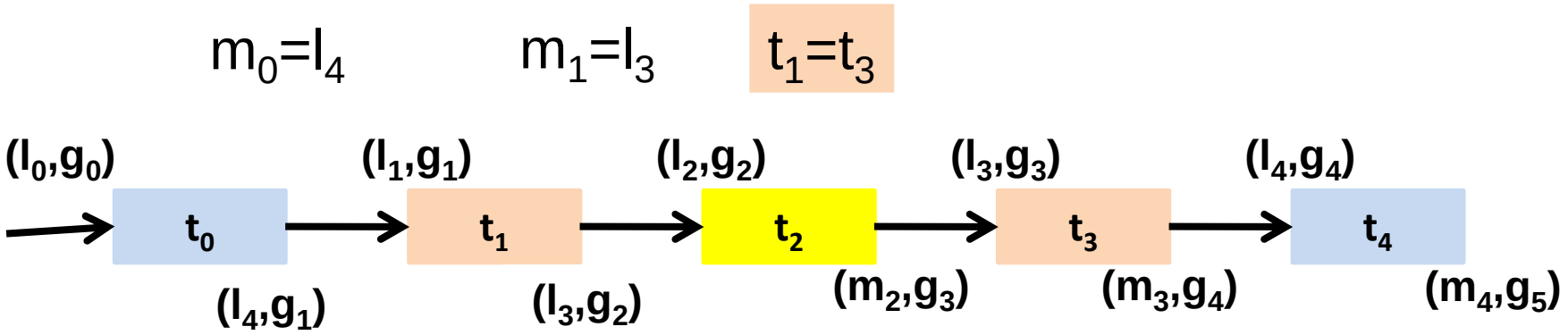
Fix k.

Is an error reachable within k context-switches?

- [Qadeer-Rehof, TACAS'05]: Decidable.
  - Proof uses tuples of automata that capture stack contents of processes
  - Each such automaton is grown as for sequential programs [Schwoon, PhdThesis 2000][Esparza-Schoown, CAV'01]

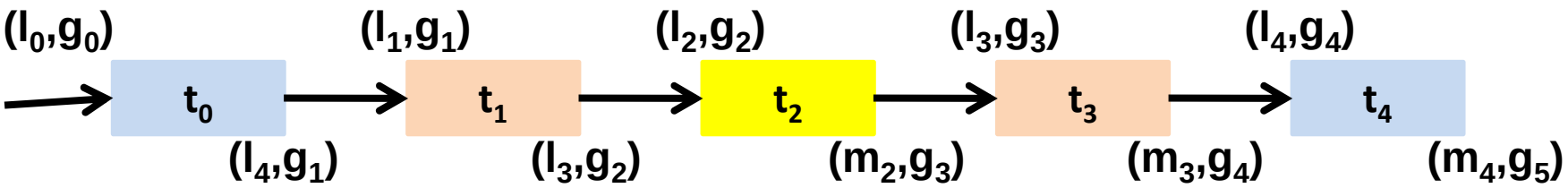
- Most concurrency-related bugs often show up within few contexts [Musuvathi-Qadeer, PLDI'07]

A computation up to 4 context-switches



- $t_0,\ldots,t_4$ = active component in each context (colors identify components)
- $l_0,\ldots,l_4$ = local states at the beginning of each context
- $m_0,\ldots,m_4$ = local states at the end of each context
- a re-activated component restarts from the last visited local state (when last context-switched out)
- $g_1,\ldots,g_4$ = shared states at context-switches

Towards a summary relation



- Consider a summary (u,v) as for sequential prgms
  - add summaries within the same context is fine
- Suppose $v=(m_2,g_3)$, we want to add a *consistent* summary (u',v') where $v'=(l_3,g_3)$
- We can add (u',v') if we know that according to a global run matching $g_0,\ldots,g_5$ and $t_0,\ldots,t_5$
  - (u,v) was added in context 2 and
  - (u',v''), $v''=(l_3,g_2)$, was added in contex 1

A summary relation

- Summary tuples are of the form:

  $(u, v, ecs, cs, \{g_i\}_{i=1,..k}, \{t_i\}_{i=0,..k})$

meaning that there is a global computation which

- visits an entry state u in context ecs and then v in context cs
- u and v are in the same function of program $t_{cs}=t_{ecs}$ (and (u,v) is a *summary* edge)
- uses cs context-switches
- executes t_i at each context i
- shared state at the i-th context-switch is g_i

Fixed-point formulation

- Let $G=\{g_i\}_{i=1,..k}$, $T= \{t_i\}_{i=0,..k}$

- $Reach(u, v, ecs, cs, G, T) = \phi_{init} \vee \phi_{int} \vee \phi_{call} \vee \phi_{ret} \vee \phi_{1st-switch} \vee \phi_{switch}$

$\phi_{init} = (cs= ecs=0 \wedge Entry(u.pc) \wedge u=v \wedge Init(t_0, u.pc))$

$\phi_{int} = \exists x. (Reach(u, x, ecs, cs, G, T) \wedge ProgramInt(x, v))$

$\phi_{call} = \exists x, y, ecs. (Reach(x, y, ecs, cs, G, T) \wedge Call(y, u) \wedge ecs = cs \wedge u=v)$

Fixed-point formulation

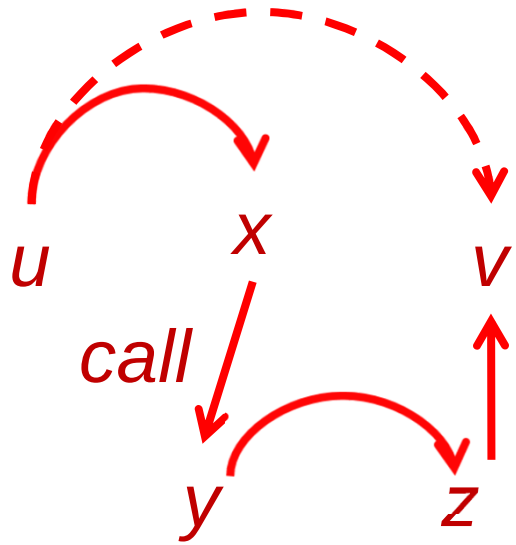
$\phi_{ret} = \exists x, y, z, cs'.$

$(Reach(u, x, ecs, cs', G, T)$

$\wedge Call(x, y) \wedge Reach(y, z, cs', cs, G, T)$

$\wedge Exit(z.pc) \wedge Return(x, z, v) \wedge cs' \leq cs$

$)$



Caller's **cs** must be less than the **cs** at return

Fixed-point formulation

$\phi_{1st-switch} = \exists x, y, cs', ecs'.$

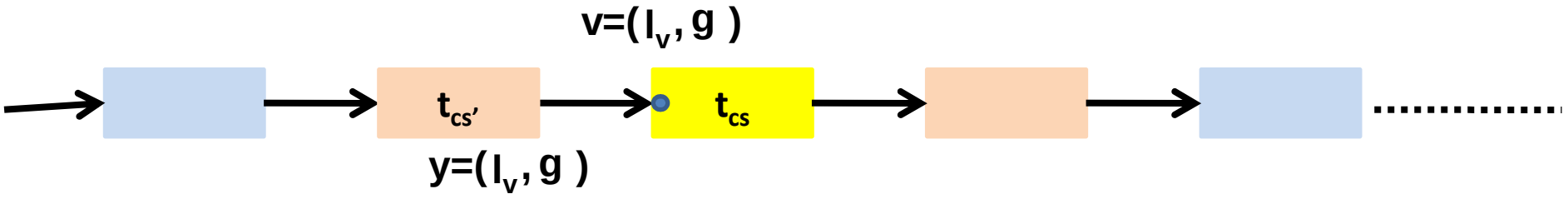
$(Reach(x, y, ecs', cs', G, T)$

$\wedge (cs=cs'+1) \wedge First(t_{cs}, cs, T)$

$\wedge (v.Global = g_{cs} = y.Global)$

$\wedge (u = v) \wedge (ecs = cs) \wedge Init(t_{cs}, v.pc)$

$)$



Fixed-point formulation

ϕ_{switch} =

$(\exists x, y, cs, ecs.$

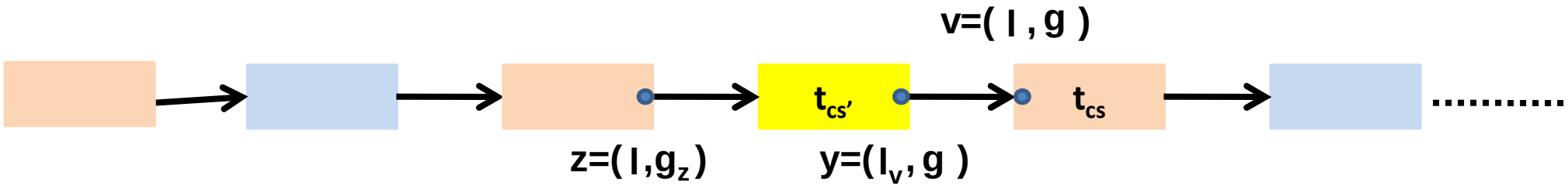
$(Reach(x, y, ecs, cs, G, T) \wedge (cs=cs+1)$

$\wedge \neg First(t_{cs}, cs, t)$

$\wedge (v.Global = g_{cs} = y.Global)))$

$\wedge (\exists z, cs''.(Reach(u, z, ecs, cs'', G, T) \wedge (cs''<cs)$

$\wedge Consecutive(cs'', cs,T) \wedge z.Local = v.Local))$



Parameterized Boolean programs

Parameterized Boolean programs

- A fixed number of recursive Boolean programs $P_1, \ldots, P_n$
- A fixed number of shared variables
- Each P_i can be executed on possibly unboundedly many threads
- Each computation has a fixed number of threads
  - threads are not created dynamically
- Interesting class of programs (e.g., device drivers)

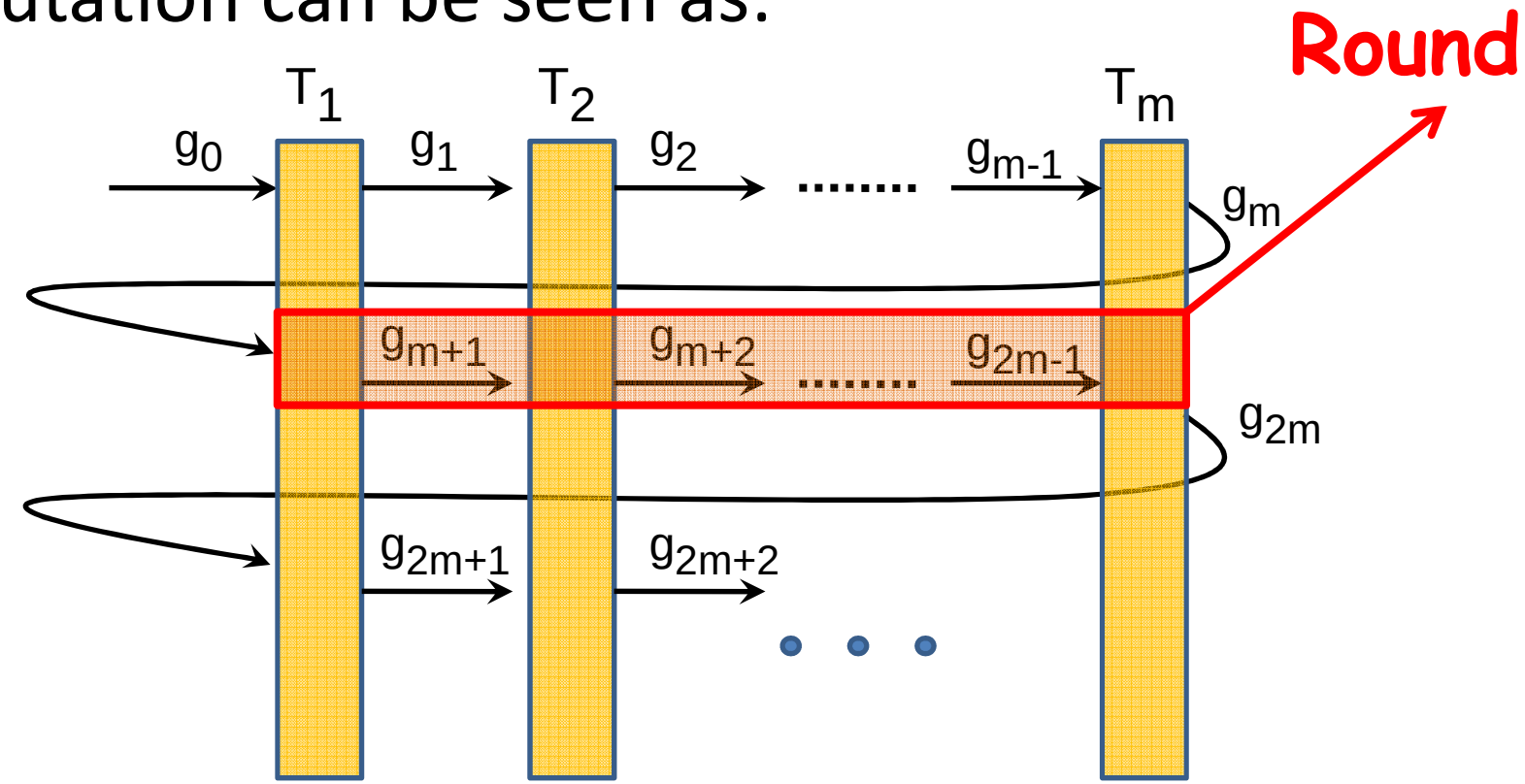
Parameterized Boolean programs

- Infinite number of states:
  - each thread is possibly recursive
  - number of threads is unbounded
- Reachability is clearly undecidable
- What about reachability within a bounded number of context-switches?

  Decidable!

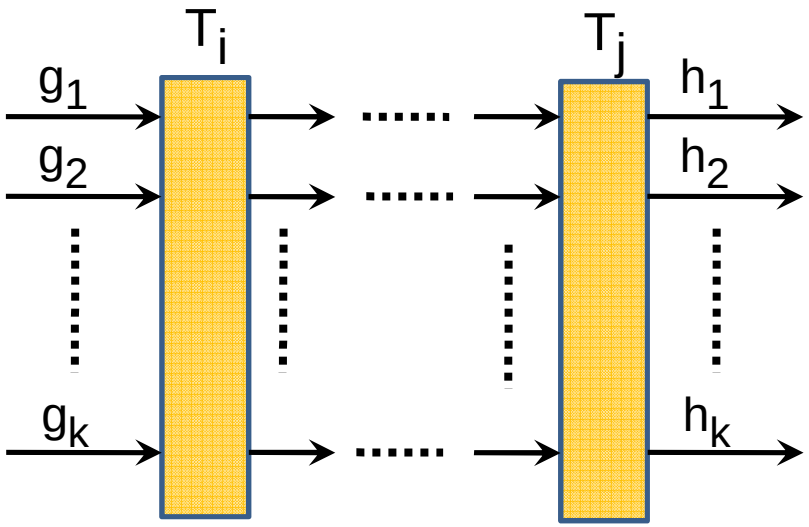
Round-robin scheduling of threads

- Fix a round-robin scheduling
- A computation can be seen as:



Linear Interface

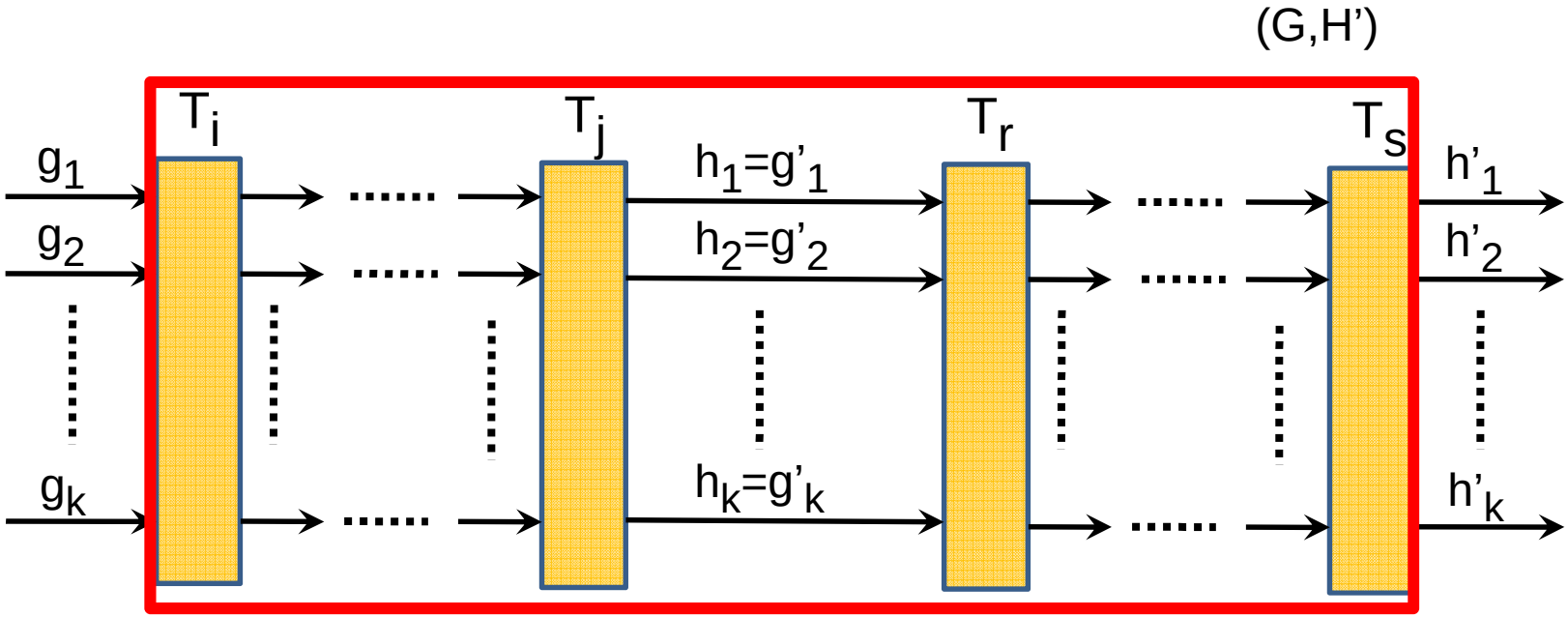
- Let $G=(g_1,\ldots,g_k)$ and $H=(h_1,\ldots,h_k)$
- (G,H) is a k-linear interface if:



- *G is the input and H is the output*

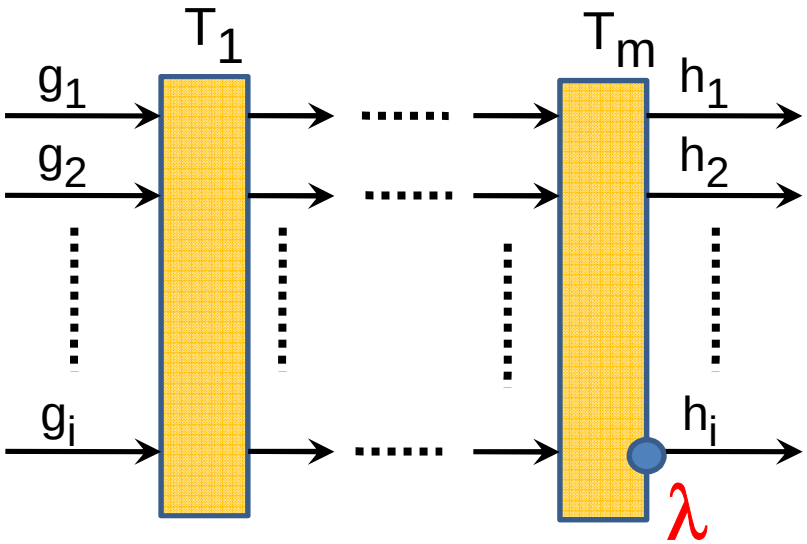
Linear Interfaces compose

- Let (G,H) and (G',H') be linear interfaces s.t. H=G'.
  Then, (G,H') is a linear interface.



A summary relation for parametrized prgms

- Let G=(g_1,...,g_k) and H=(h_1,...,h_k)
- EagerLI(i, λ, G,H) means that
  - (G,H) is a linear interface over threads T_1,..,T_m



- λ is a local state
- i is current round
- **(λ, h_i)** current state

A fixed point-algorithm

EagerLI(i, λ, G,H) =

(i = 1 ∧ LocalInit(λ) ∧ $g_1 = h_1$)

∨ (i > 1 ∧ $g_i = h_i$ ∧ EagerLI(i − 1, λ, G, H))

∨ <local reachability>

∨ (∃G', λ'. EagerLI(i, λ' ,G,G') ∧ EagerLI(i, λ ,G',H))

starts each context

explores each context

composes linear interfaces

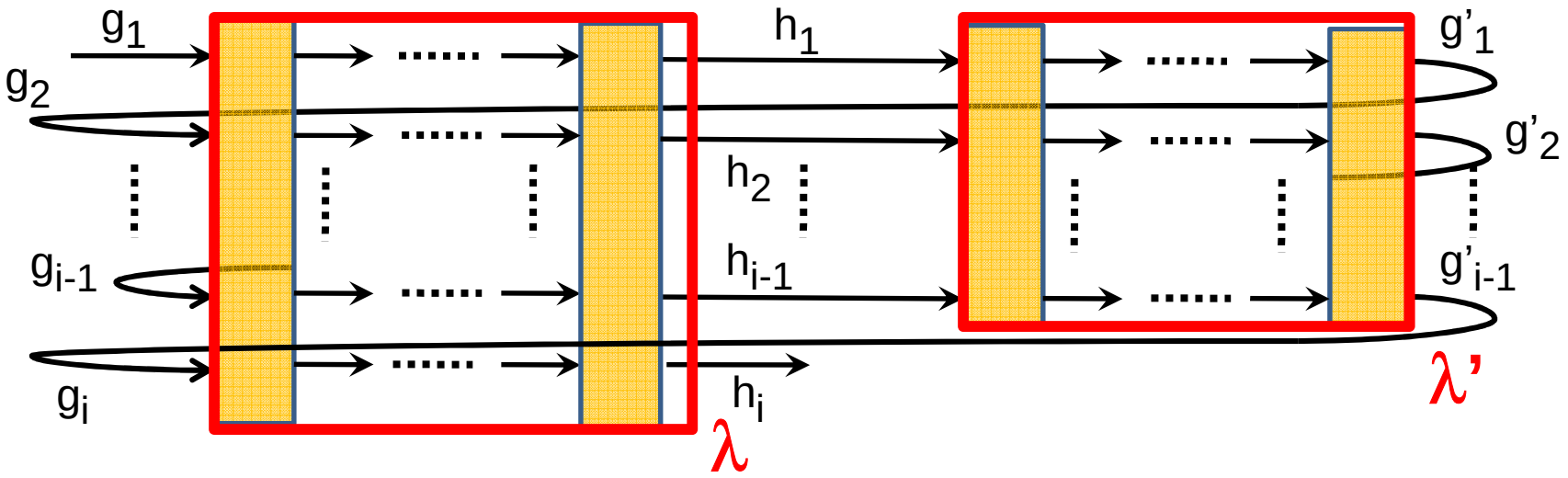
Theorem.

EagerLI is the set of all the linear interfaces of lenght k for the considered parameterized Boolean program

A fixed point-algorithm

$$\text{Reach}(\lambda,g) = \exists G,H,\ i.\ (\ \text{SharedInit}(g_1) \wedge \text{EagerLI}(i,\lambda,G,H) \wedge g=h_i$$

$$\wedge\ (i=1\ \vee \exists G',\lambda'.\ (\ i>1 \wedge \text{EagerLI}(i-1,\ \lambda'\ ,H,G') \wedge \text{Wrap}(G,G')\ )\ ))$$



Theorem.

(λ,g) is reachable in P if and only if Reach(λ,g)

A lazy fixed-point algorithm

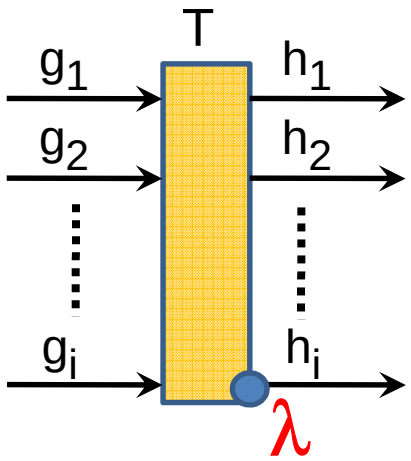
- Reach(λ,g) explores also unreachable states
  - May be inefficient in practice

- We want to compute linear interfaces by exploring only reachable states

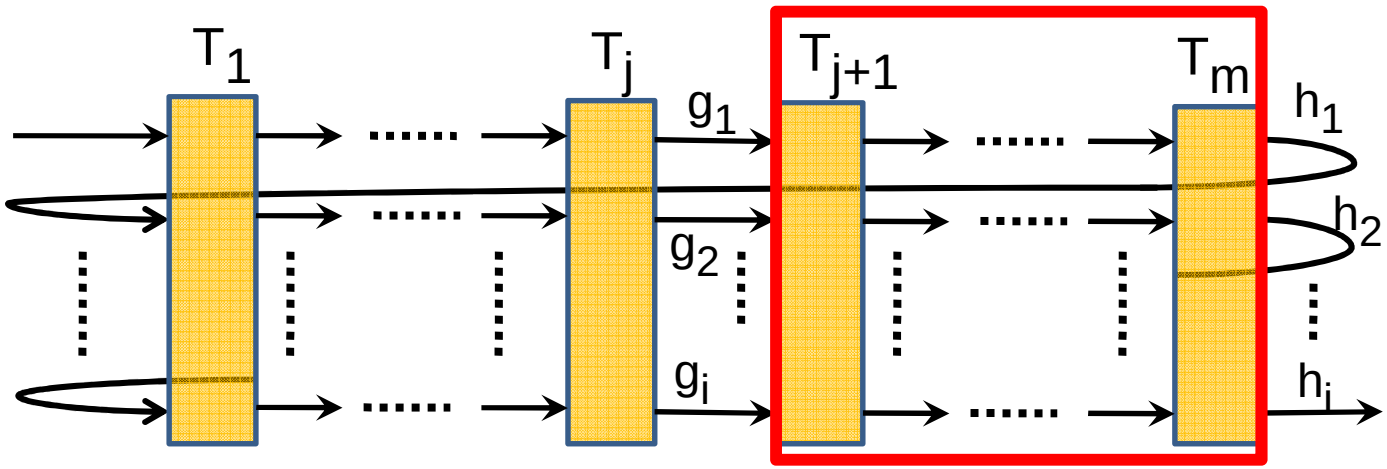
- Formula is more involved, we need more relations

Relations

- ThrdLI(i, λ, G,H):

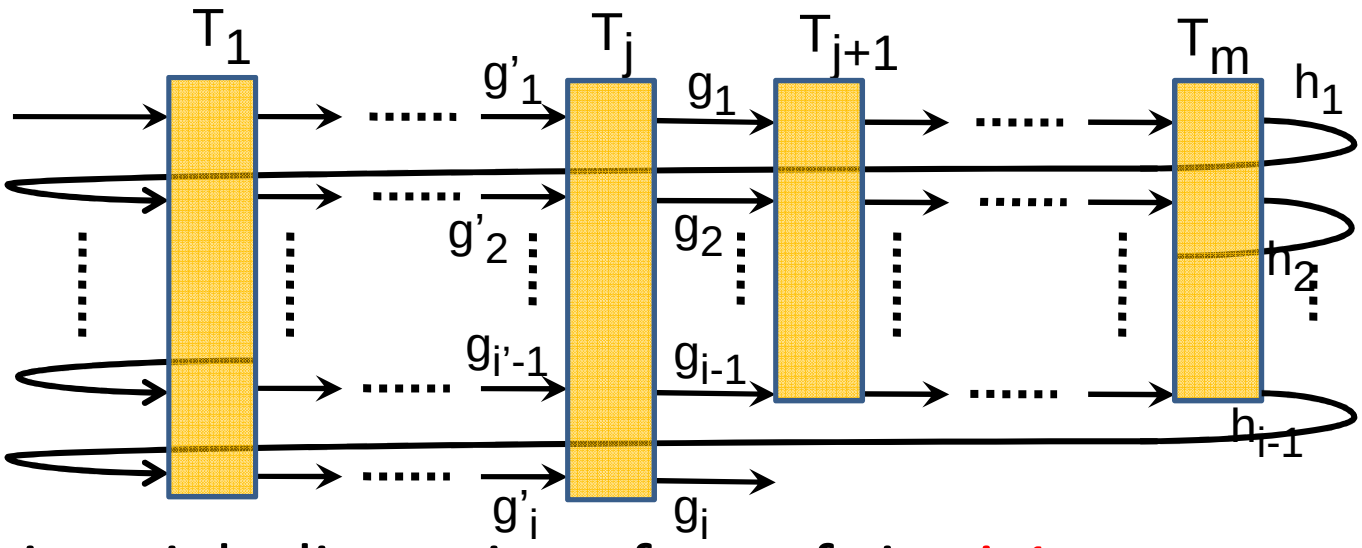


- RightLI(i,G,H): (G,H) is a right linear interface of size i



Relations

- WantRightLI(i, G,H) means that there exists a run



- (G,H) is a right linear interface of size i-1
- (G',G) is a linear interface of size i over a single thread

Fixed-point formulation

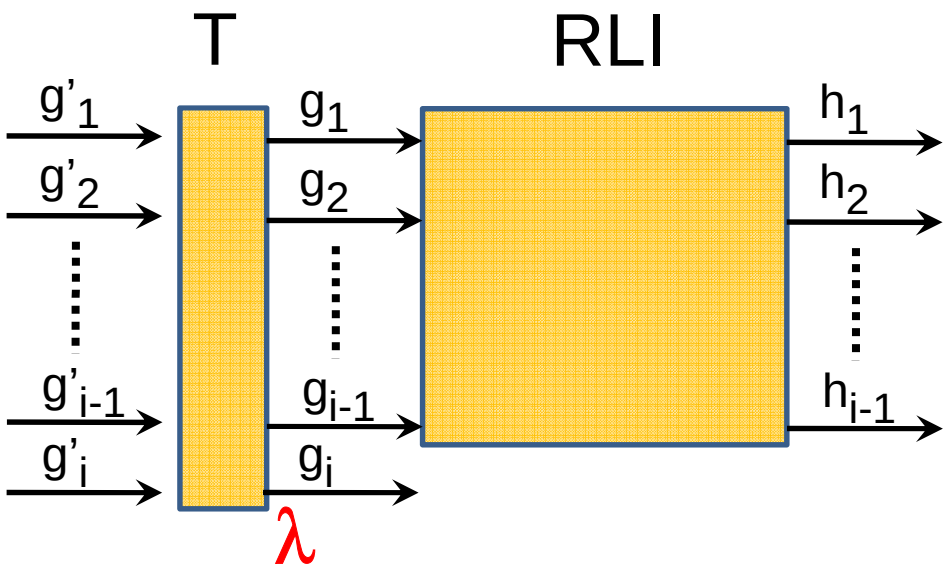
WantRightLI(i, G, H) =

$\exists$ G′, λ. (ThrdLI(i, λ, G′, G) ∧ RightLI(i − 1, G, H)

∧ existence of a global run as before

(WantRightLI(i, G′, H)

∨ (SharedInit(g'_1)∧Wrap(G′, H))))



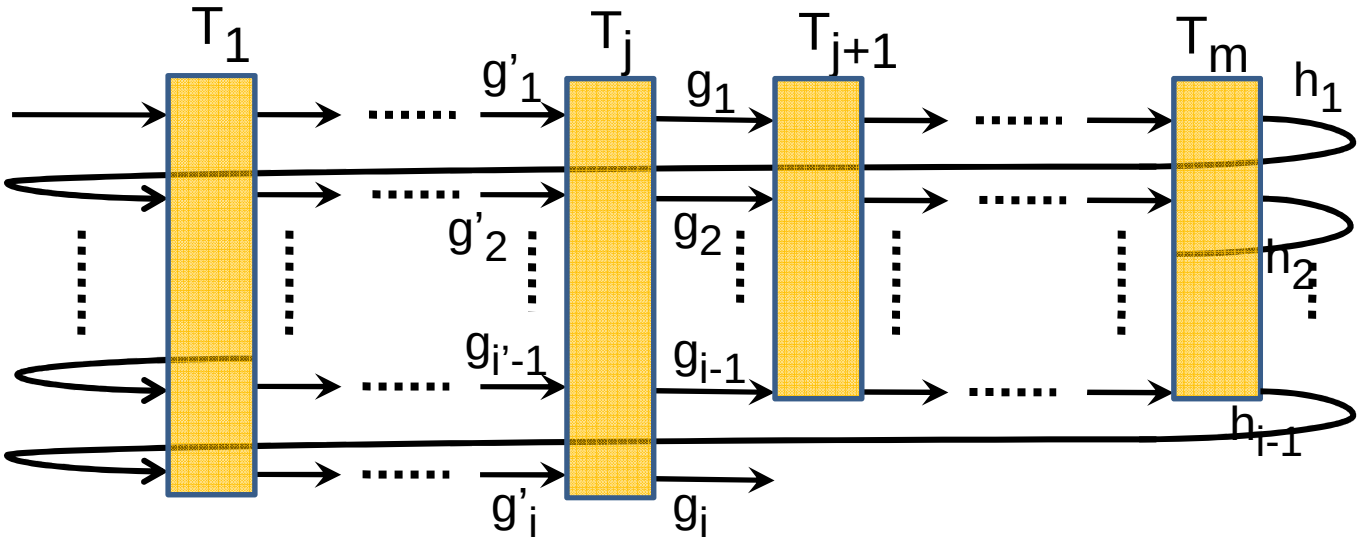
Fixed-point formulation

RightLI(i, G, H) =

$\exists \lambda$. (ThrdLI(1, λ, G, H)

$\vee \exists$ H'. (ThrdLI(1, λ, G, H') $\wedge$ RightLI(i,H',H)))

$\wedge$ (i = 1 $\vee$ WantRightLI(i, G, H))



Fixed-point formulation

ThrdLI(i, λ, G,H) =

(i = 1 ∧ LocalInit(λ) ∧ $g_1 = h_1$

∧ (SharedInit(g_1) ∨ ∃G′, λ′.ThrdLI(1, λ′, G′, G)))

∨ (i > 1 ∧ $g_i = h_i$ ∧ ThrdLI(i − 1, λ, G, H)

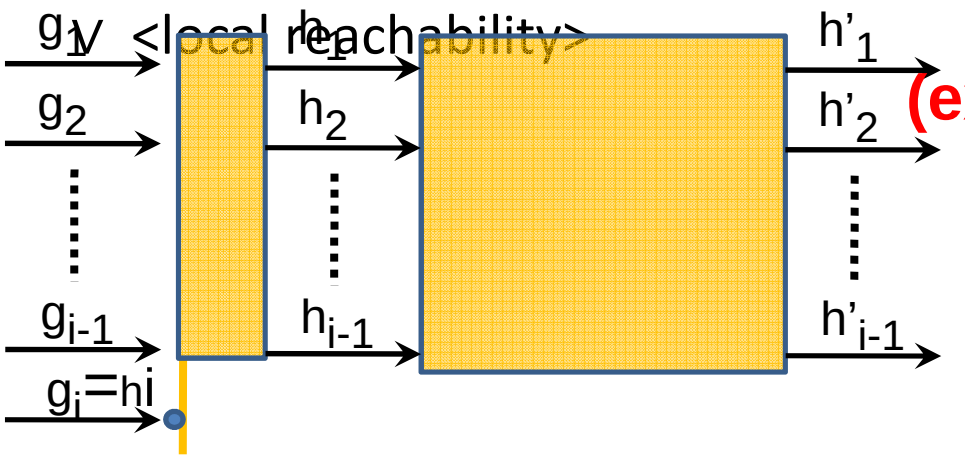
∧ ∃H′. (RightLI(i − 1, H,H′)

∧ ((SharedInit(g_1) ∧ Wrap(G,H′))

∨ WantRightLI(i, G, H′))))

(starts each context in first round)

(ensures lazy exploration for the first round)



(starts contexts (explores each context)

(ensures lazy exploration for the first round)

Correctness

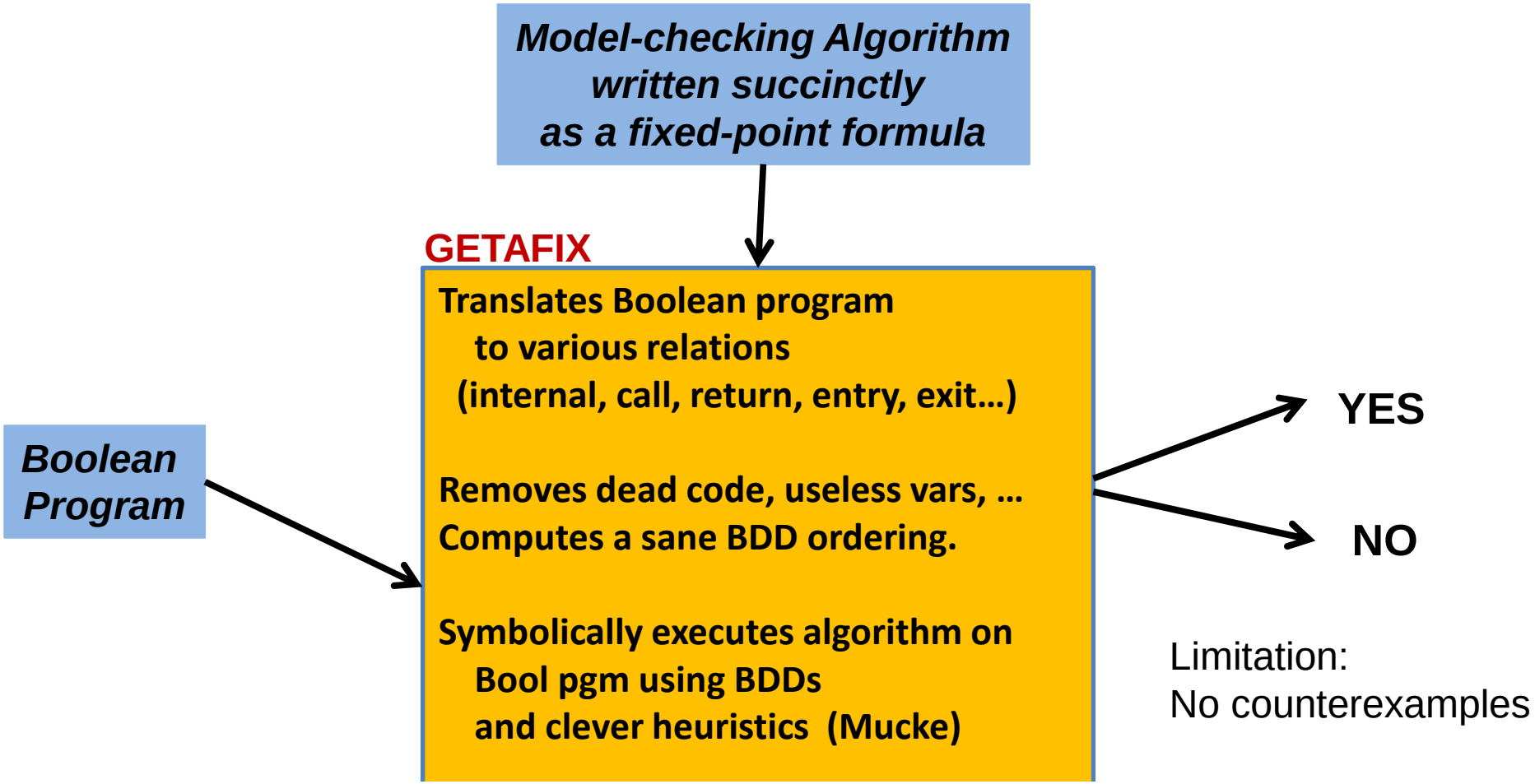
- Observe:
  - ThrdLI captures the actual exploration of program states
  - RightLI and WantRightLI: “service relations” for ensuring laziness and guide context-switching
- Context-bounded reachability is answered by checking

$$\exists i, \lambda, G, H.\ (1 \le i \le k)$$
$$\wedge\ \mathrm{ThrdLI}(i, \lambda, G, H) \wedge \mathrm{Target}(\lambda)$$

Concluding……

Getafix

A framework for writing model-checking algorithms using fixed-point formulae.



Summary based approach

- BEBOP computes procedure summaries for Boolean programs [Ball-Rajamani, SPIN'00]
- First version of MOPED, an automaton accepting reachable stack configurations is constructed [Esparza-Schoown, CAV'01]
  - incrementally adds summary edges to the automaton until saturation
- Summaries for recursive state machines can be computed by DATALOG rules [Alur et al., TOPLAS'05]

Bounded context-switching

- [Quadeer,Wu, PLDI'04]: introduce bounded-context switch reachability (only 2 context-switches addressed)
- Algorithm [Qadeer-Rehof, TACAS'05] is complex and was implemented in [Suwimonteerabuth-Esparza-Schwoon,SPIN'08]
- [Suwimonteerabuth-Esparza-Schwoon,SPIN'08] implement algorithm of [Qadeer-Rehof, TACAS'05] and a new (simpler) version
- Algorithms presented in this lecture:
  - concurrent programs [La Torre-Madhusudan-Parlato, PLDI'09]
  - parameterized programs [La Torre-Madhusudan-Parlato, CAV'10]

Bounded context-switching (concurrent-to-sequential)

- Only the local state of one component program is kept at each time
  - multiple copies of shared variables
- [Lal-Reps,CAV'08]: translation to sequential reachability which yields eager state exploration
- [Lahiri-Qadeer-Rakamaric, CAV09]: use this translation for deductive verification of C programs
- [La Torre-Madhusudan-Parlato, CAV'09]: translation achieving lazy state exploration
  - preserves invariants of the concurrent program
- We also have eager and lazy translations for parameterized programs [La Torre-Madhusudan-Parlato, 2010 unpublished]