

Programming model-checkers using a fixed-point calculus

Salvatore La Torre
Università degli Studi di Salerno



Topic of the tutorial

- 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

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

How model-checkers look in our formalism

```
mu bool Reachable( Module s_mod, PrCount s_pc, Local s_cl, s_ENTRY_C
s_mod < s_pc, s_pc < s_CL, s_CL ~+ s_ENTRY_C
( ( exists Module t_mod, PrCount t_pc, Local t_cl, t_ENTRY_C
( target(t_mod,t_pc) & Reachable(t_mod,t_pc, t_cl, t_ENTRY_C

| (enforce(s_mod, s_cl, s_cg) & (
(s_mod=0 & s_pc=0)
| ( s_pc=0 & CopyLocals(s_mod, s_ENTRY_C
& (exists Module t_mod, PrCount t_pc, Local t_cl, t_ENTRY_C
( (Reachable(t_mod,t_pc,t_cl,t_ENTRY_C
& 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_ENTRY_C
& ( programInt(s_mod,t_pc,s_pc,t_cl,s_CL,t_ENTRY_C

| (exists PrCount t_pc, Global t_cg, Module u_mod, PrCount u_pc, Local u_cl, u_ENTRY_C
( exists Local t_cl. ( (Reachable(s_mod,t_pc,t_cl,t_ENTRY_C
& SkipCall(s_mod,t_pc,s_pc) & programInt(s_mod,t_pc,s_pc,t_cl,s_CL,t_ENTRY_C
& SetReturnTS(s_mod,u_mod,t_pc,u_pc,t_ENTRY_C
& ( exists Local u_cl, Global u_cg.
( (Reachable(u_mod,u_pc,u_cl,u_ENTRY_C
& SetReturnUS(s_mod,u_mod,t_pc,u_pc,t_ENTRY_C
));

( exists Module s_mod, PrCount s_pc, Local s_cl, s_ENTRY_C
( target(s_mod,s_pc) & Reachable(s_mod,s_pc, s_cl, s_ENTRY_C
```

• 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

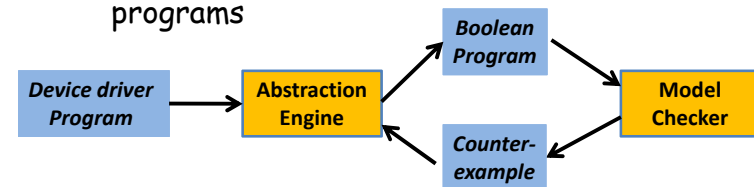
• Underlying solver GETAFIX will convert the program and run this algorithm efficiently

Why fixed-point calculus?

- Natural formalism in verification
 - Most symbolic model-checking algorithms essentially compute fixed-points
 - Ex: compute the least X s.t. $X = \text{Start} \vee X \vee \text{Succ}(X)$
(forward reachability)
- Right primitives
 - easy encoding of model-checking algorithms
 - sufficiently low-level to express algorithmic details

Why check Boolean programs?

- Programs can be *restricted* to finite domains giving Boolean programs
- Programs can be abstracted to Boolean pgms:
 - SLAM/SDV from Microsoft for verifying device drivers
 - *Predicate abstraction* to derive Boolean pgms
 - Bool pgm tracks predicates over the data domain
 - BEBOP / MOPED - Model-checkers for Boolean programs



Outline of the talk

- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

Quantified Boolean logic

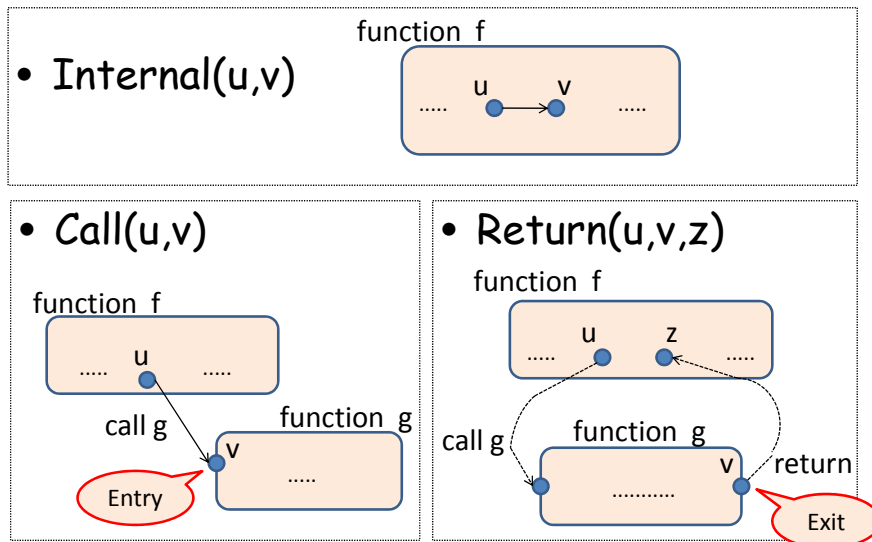
First-order logic over the Boolean domain

$$\text{BoolExp } B ::= T \mid F \mid R_k(x_1, \dots, x_k) \mid \neg B \mid B \wedge B \mid B \vee B \mid \exists x.(B) \mid \forall x.(B)$$

Variables interpreted over the Boolean domain $\{T, F\}$

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

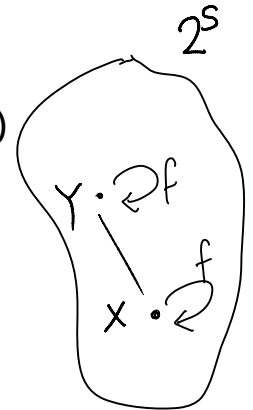
Some relations defined by pgms



Fixed-points

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

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



- $X \in 2^S$ if 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

Consider a function $f : 2^S \rightarrow 2^S$ that is monotonic
 $X \subseteq Y$ then $f(X) \subseteq f(Y)$

Consider the sequence

$$X_0 = \emptyset$$

$$X_1 = f(X_0)$$

$$X_2 = f(X_1) \dots$$

*This eventually converges to the least-fixed point
 (and hence always exists).*

Eg: Reachability in non-rec programs

$$\begin{aligned} \text{Reach}(pc, x) &= (pc=0 \wedge \text{Init}(x)) \\ &\vee \exists pc', y. (\text{Reach}(pc', y) \wedge \text{Internal}(pc', y, pc, x)) \end{aligned}$$

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

$f_1(x_1, \dots, x_r) = \text{PosBoolExp using } f_1, \dots, f_m$

$f_2(y_1, \dots, y_s) = \text{PosBoolExp using } f_1, \dots, f_m$

⋮

$f_m(z_1, \dots, z_t) = \text{PosBoolExp using } f_1, \dots, f_m$

- 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, \dots, f_m exists.

Outline of the talk

- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

Computation of the least fixed-points

- Least fixed-point of f_1, \dots, 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 $S := \emptyset$

while (S does not equal $f(S)$)

- $Eq' := Eq$

- $Eq'' := Eq'$

- Let f_1, \dots, f_m

- $S := \text{evaluate}(Eq'', f_1, \dots, f_m)$

}

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)

(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

```

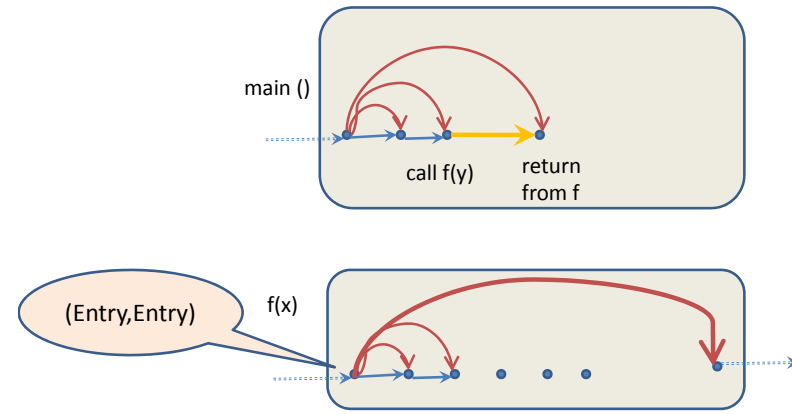
main () {
  bool a,b;
  a := T;
  b := *;
  while (b) {
    a:= f(b);
    b := * constrain (!(a && b'));
  }
  if (c) {
    ERR: skip
  }
}

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"

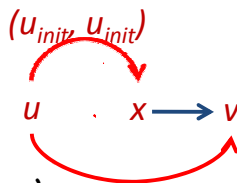
Checking recursive Boolean programs



Summaries: (Entry, State)
meaning the state is reachable from the entry of the function (perhaps using calls to other functions)

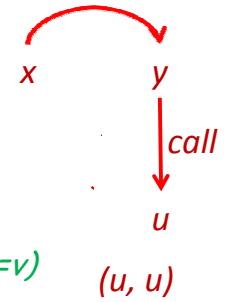
Writing the algorithm

$$\begin{aligned}
 \text{Summary}(u,v) = & \\
 & (\text{Entry}(u.pc) \wedge u=v \wedge \text{Init}(u.pc)) \\
 \vee \exists x. & (\text{Summary}(u, x) \wedge \text{Internal}(x, v)) \\
 \vee \exists x, y. & (\text{Summary}(x, y) \wedge \text{Call}(y, u) \wedge u=v) \\
 \vee \exists x, y, z. & (\text{Summary}(u, x) \wedge \text{Call}(x, y) \\
 & \wedge \text{Summary}(y, z) \wedge \text{Exit}(z.pc) \wedge \text{Return}(x, z, v))
 \end{aligned}$$



Writing the algorithm

$$\begin{aligned}
 \text{Summary}(u,v) = & \\
 & (\text{Entry}(u.pc) \wedge u=v \wedge \text{Init}(u.pc)) \\
 \vee \exists x. & (\text{Summary}(u, x) \wedge \text{Internal}(x, v)) \\
 \vee \exists x, y. & (\text{Summary}(x, y) \wedge \text{Call}(y, u) \wedge u=v) \\
 \vee \exists x, y, z. & (\text{Summary}(u, x) \wedge \text{Call}(x, y) \\
 & \wedge \text{Summary}(y, z) \wedge \text{Exit}(z.pc) \wedge \text{Return}(x, z, v))
 \end{aligned}$$



Writing the algorithm

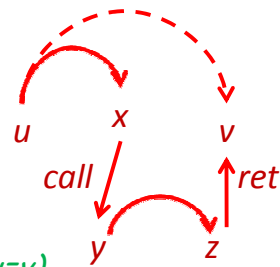
$Summary(u,v) =$

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

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

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

$\vee \exists x, y, z.(Summary(u, x) \wedge Call(x, y)$
 $\wedge Summary(y, z) \wedge Exit(z.pc) \wedge Return(x, z, v)$
 $)$



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_CL,t_ENTRY_CG) & CopyGlobals(s_mod, t_CG, s_ENTRY_CG) &
        CopyLocals(s_mod, t_ENTRY_CL, s_CL) )
    | (exists PrCount t_pc, Local t_CL, Global t_CG.
      ( (Reachable(s_mod,t_pc,t_CL,t_CG,s_ENTRY_CL,s_ENTRY_CG) & programInt(s_mod,t_pc,s_pc,t_CL,s_CL,t_CG,s_CG)
        & CopyGlobals(s_mod, t_CG, s_ENTRY_CG) & CopyLocals(s_mod, t_ENTRY_CL, s_CL) )
      | (exists PrCount t_pc, Global t_CG, Module u_mod, PrCount u_pc, Local u_CL, Global u_CG.
        ( (exists Local t_CL. ( Reachable(s_mod,t_pc,t_CL,t_CG,s_ENTRY_CL,s_ENTRY_CG) & SkipCall(s_mod,t_pc,s_pc)
          & programCall(s_mod,t_pc,u_mod,u_pc,t_CL,s_CL,t_CG,s_CG) & SetReturnTS(s_mod,u_mod,t_pc,u_pc,t_CL,s_CL,t_CG,s_CG)
          & (exists Local u_CL, Global u_CG.
            ( (Reachable(u_mod,u_pc,u_CL,u_CG,u_ENTRY_CL,u_ENTRY_CG) & SetReturnUS(s_mod,u_mod,t_pc,u_pc,u_CL,s_CL,t_CG,s_CG)
              & CopyLocals(u_mod,u_ENTRY_CL,u_CL) )
          )))
      ( exists Module s_mod, PrCount s_pc, Local s_CL, Global s_CG, Local s_ENTRY_CL, Global s_ENTRY_CG.
        ( target(s_mod,s_pc) & Reachable(s_mod,s_pc,s_CL,s_CG,s_ENTRY_CL,s_ENTRY_CG) ) )
    )
  )
)

```

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

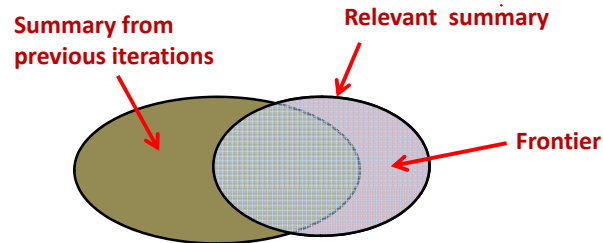
Correctness of the algorithm

- 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

$$New2(u, v) = (\exists x.(Relevant(x.pc) \wedge Summary(1, u, x) \wedge Call(x, v))) \vee (\exists x, y, z.(Summary(u, x) \wedge IntoCall(x, y) \wedge Summary(y, z) \wedge Exit(z.pc) \wedge Return(x, z, v) \wedge (Relevant(x.pc) \vee Relevant(z.pc))))$$

- 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

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

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

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

Outline of the talk

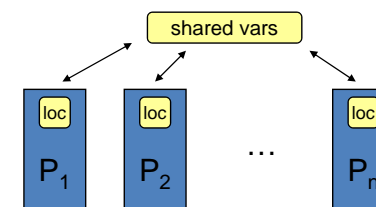
- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

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

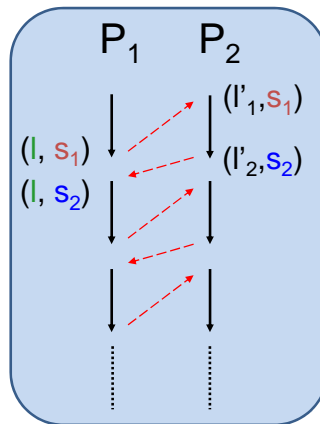
Concurrent Boolean programs

- A fixed number of recursive Boolean programs P_1, \dots, 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)

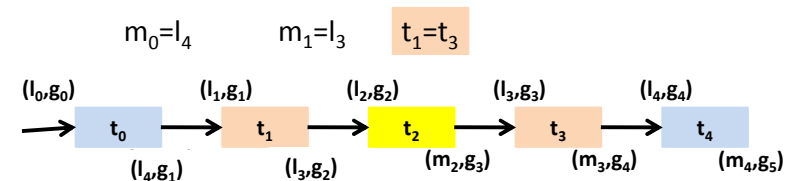
Bounded context-switching

Fix k .

Is an error reachable within k context-switches?

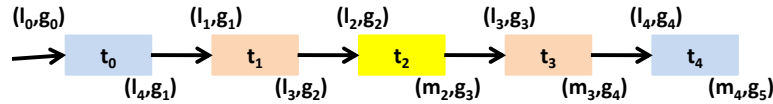
- [Qadeer-Rehof, TACAS'05]: Decidable.
 - Proof uses tuples of automata that capture stack contents of processes
 - Each such automaton is grown as for sequential programs [Schwoon, PhdThesis 2000][Esparza-Schoown, CAV'01]
- Most concurrency-related bugs often show up within few contexts [Musuvathi-Qadeer, PLDI'07]

A computation up to 4 context-switches



- t_0, \dots, t_4 = active component in each context (colors identify components)
- l_0, \dots, l_4 = local states at the beginning of each context
- m_0, \dots, 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, \dots, 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, \dots, g_5 and t_0, \dots, t_5
 - (u, v) was added in context 2 and
 - (u', v') , $v'=(l_3, g_2)$, was added in context 1

Fixed-point formulation

- Let $G=\{g_i\}_{i=1, \dots, k}$, $T=\{t_i\}_{i=0, \dots, k}$
- $Reach(u, v, ecs, cs, G, T) = \phi_{init} \vee \phi_{int} \vee \phi_{call} \vee \phi_{ret} \vee \phi_{1st-switch} \vee \phi_{switch}$
- $\phi_{init} = (cs = ecs = 0 \wedge Entry(u.pc) \wedge u=v \wedge Init(t_0, u.pc))$
- $\phi_{int} = \exists x. (Reach(u, x, ecs, cs, G, T) \wedge ProgramInt(x, v))$
- $\phi_{call} = \exists x, y, ecs. (Reach(x, y, ecs, cs, G, T) \wedge Call(y, u) \wedge ecs = cs \wedge u=v)$

A summary relation

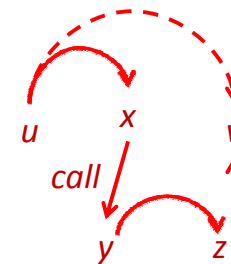
- Summary tuples are of the form: $(u, v, ecs, cs, \{g_i\}_{i=1, \dots, k}, \{t_i\}_{i=0, \dots, k})$

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

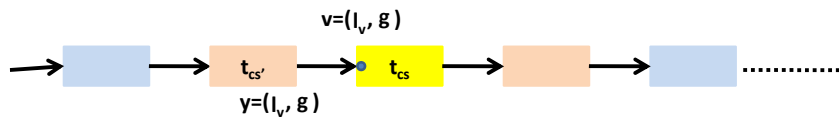
$$\phi_{ret} = \exists x, y, z, cs'. (Reach(u, x, ecs, cs', G, T) \wedge Call(x, y) \wedge Reach(y, z, cs', cs, G, T) \wedge Exit(z.pc) \wedge Return(x, z, v) \wedge cs' \leq cs)$$



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

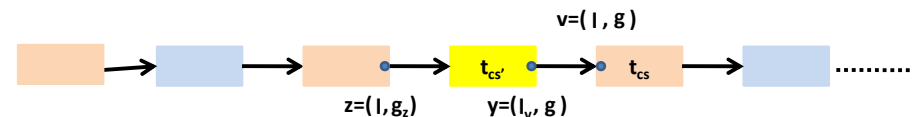
Fixed-point formulation

$$\begin{aligned} \phi_{1st-switch} = & \exists x, y, cs', ecs'. \\ & (Reach(x, y, ecs', cs', G, T) \\ & \wedge (cs = cs' + 1) \wedge First(t_{cs}, cs, T) \\ & \wedge (v.Global = g_{cs} = y.Global) \\ & \wedge (u = v) \wedge (ecs = cs) \wedge Init(t_{cs}, v.pc) \\ &) \end{aligned}$$



Fixed-point formulation

$$\begin{aligned} \phi_{switch} = & \exists x, y, cs, ecs. \\ & (Reach(x, y, ecs, cs, G, T) \wedge (cs = cs + 1) \\ & \wedge \neg First(t_{cs}, cs, T) \\ & \wedge (v.Global = g_{cs} = y.Global)) \\ & \wedge (\exists z, cs''. (Reach(u, z, ecs, cs'', G, T) \wedge (cs'' < cs) \\ & \wedge Consecutive(cs'', cs, T) \wedge z.Local = v.Local) \\ &) \end{aligned}$$



Outline of the talk

- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

Parameterized Boolean programs

- A fixed number of recursive Boolean programs P_1, \dots, 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)

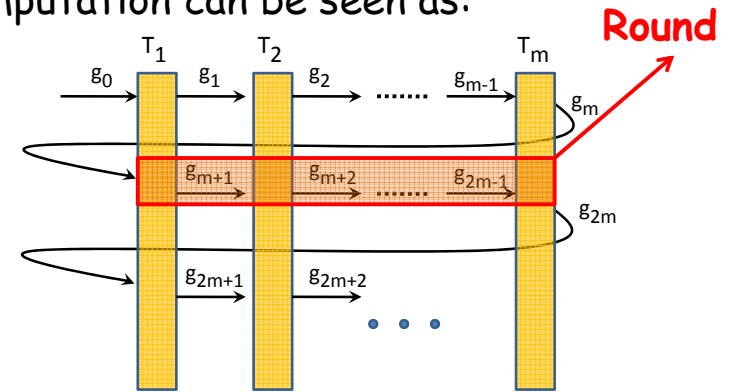
Parameterized Boolean programs

- 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!

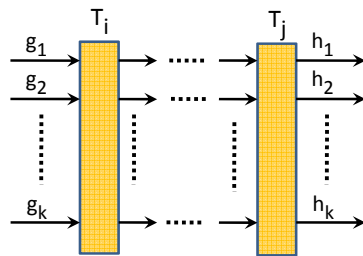
Round-robin scheduling of threads

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



Linear Interface

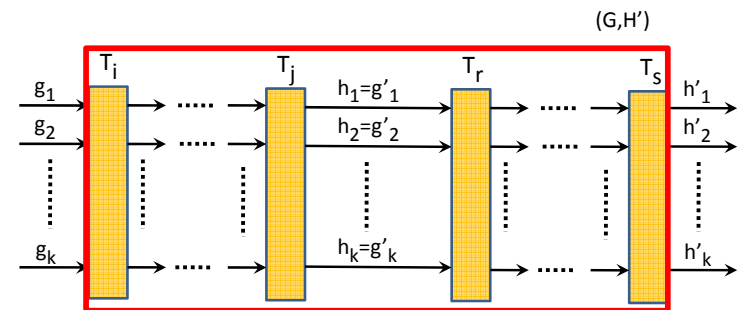
- Let $G=(g_1, \dots, g_k)$ and $H=(h_1, \dots, h_k)$
- (G,H) is a k-linear interface if:



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

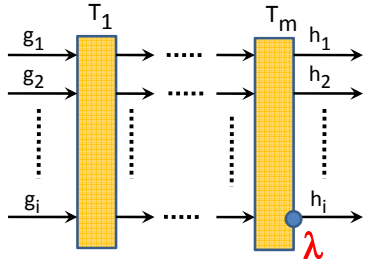
Linear Interfaces compose

- 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 $G=(g_1,\dots,g_k)$ and $H=(h_1,\dots,h_k)$
- $EagerLI(i, \lambda, G, H)$ means that
 - (G, H) is a linear interface over threads T_1, \dots, T_m



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

A fixed point-algorithm

$$\begin{aligned}
 EagerLI(i, \lambda, G, H) = & \\
 & (i = 1 \wedge LocalInit(\lambda) \wedge g_1 = h_1) \\
 & \vee (i > 1 \wedge g_i = h_i \wedge EagerLI(i - 1, \lambda, G, H)) \\
 & \vee \langle \text{local reachability} \rangle \\
 & \vee (\exists G', \lambda'. EagerLI(i, \lambda', G, G') \wedge EagerLI(i, \lambda, G', H))
 \end{aligned}$$

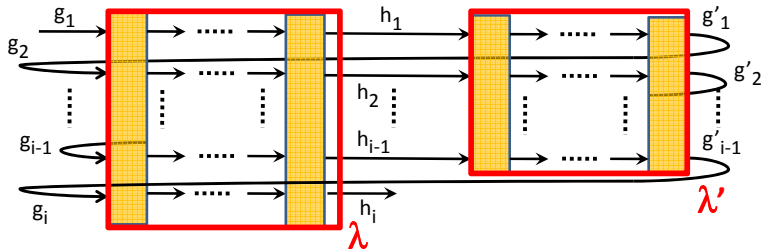
starts each context
 explores each context
 composes linear interfaces

Theorem.

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

A fixed point-algorithm

$$\begin{aligned}
 Reach(\lambda, g) = & \exists G, H, i. (SharedInit(g_1) \wedge EagerLI(i, \lambda, G, H) \wedge g = h_i \\
 & \wedge (i=1 \vee \exists G', \lambda'. (i > 1 \wedge EagerLI(i-1, \lambda', H, G') \wedge Wrap(G, G'))))
 \end{aligned}$$



Theorem.

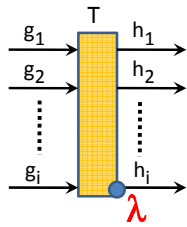
(λ, g) is reachable in P if and only if $Reach(\lambda, g)$

A lazy fixed-point algorithm

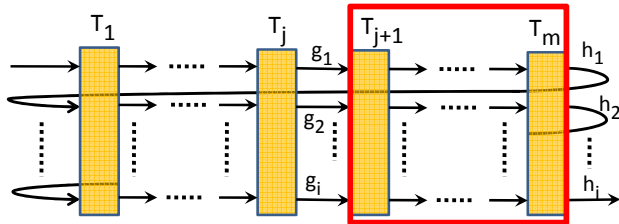
- $Reach(\lambda, g)$ explores also unreachable states
 - May be unefficient in practice
- We want to compute linear interfaces by exploring only reachable states
- Formula is more involved, we need more relations

Relations

- $\text{ThrdLI}(i, \lambda, G, H)$:



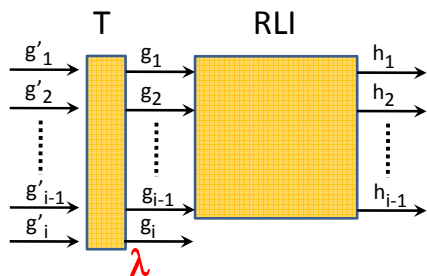
- $\text{RightLI}(i, G, H)$: (G, H) is a right linear interface of size i



Fixed-point formulation

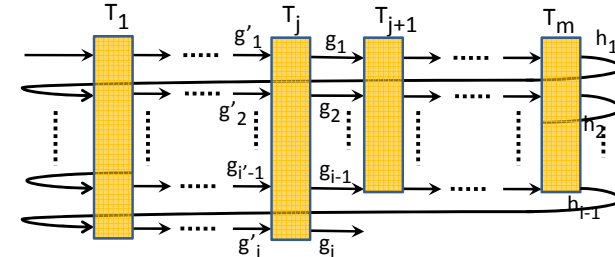
$\text{WantRightLI}(i, G, H) =$

$$\begin{aligned} & \exists G', \lambda. (\text{ThrdLI}(i, \lambda, G', G) \wedge \text{RightLI}(i-1, G, H) \\ & \quad \wedge \text{existence of a global run as before} \\ & (\text{WantRightLI}(i, G', H) \\ & \quad \vee (\text{SharedInit}(g'_1) \wedge \text{Wrap}(G', H)))) \end{aligned}$$



Relations

- $\text{WantRightLI}(i, G, H)$ means that there exists a run

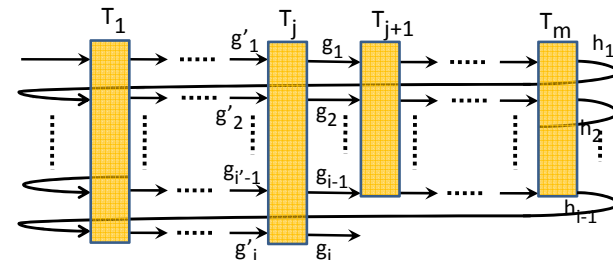


- (G, H) is a right linear interface of size $i-1$
- (G', G) is a linear interface of size i over a single thread

Fixed-point formulation

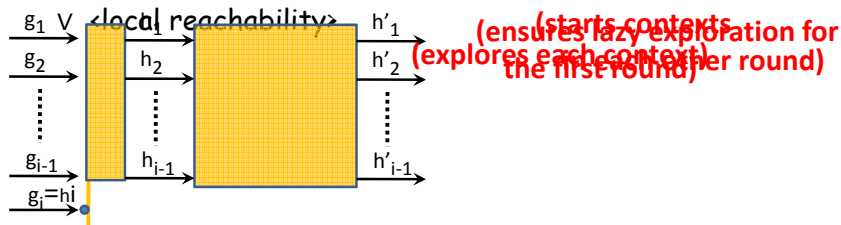
$\text{RightLI}(i, G, H) =$

$$\begin{aligned} & \exists \lambda. (\text{ThrdLI}(1, \lambda, G, H) \\ & \quad \vee \exists H'. (\text{ThrdLI}(1, \lambda, G, H') \wedge \text{RightLI}(i, H', H))) \\ & \wedge (i = 1 \vee \text{WantRightLI}(i, G, H)) \end{aligned}$$



Fixed-point formulation

$\text{ThrdLI}(i, \lambda, G, H) =$
 $(i = 1 \wedge \text{LocalInit}(\lambda) \wedge g_1 = h_1$ **(starts each context in first round)**
 $\wedge (\text{SharedInit}(g_1) \vee \exists G', \lambda'. \text{ThrdLI}(1, \lambda', G', G))$
 $\vee (i > 1 \wedge g_i = h_i \wedge \text{ThrdLI}(i - 1, \lambda, G, H)$ **(ensures lazy exploration for the first round)**
 $\wedge \exists H'. (\text{RightLI}(i - 1, H, H')$ **(starts context)**
 $\wedge ((\text{SharedInit}(g_i) \wedge \text{Wrap}(G, H'))$ **(ensures lazy exploration for the first round)**
 $\vee \text{WantRightLI}(i, G, H'))$)



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

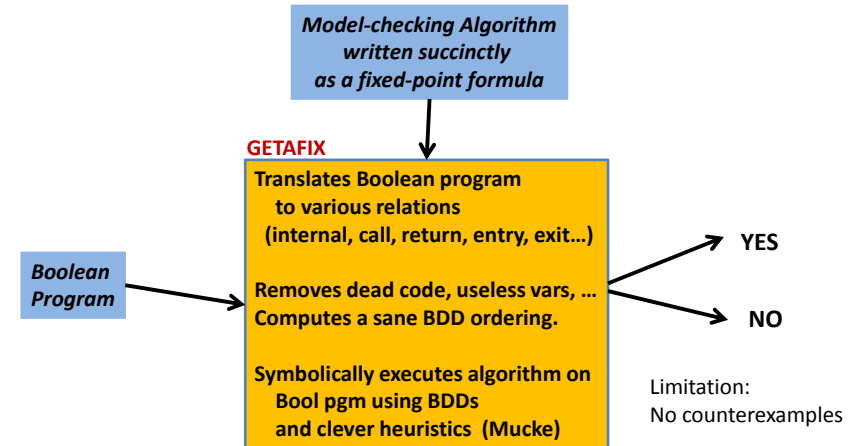
$$\exists i, \lambda, G, H. (1 \leq i \leq k) \wedge \text{ThrdLI}(i, \lambda, G, H) \wedge \text{Target}(\lambda)$$

Outline of the talk

- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

Getafix

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



Experiments

	#LOC	G	L	Ans	Getafix			Moped Ver 1	Moped Ver2	Bebop (SLAM / SDV)
					TIME (s)			TIME(s)	TIME(s)	TIME(s)
					BDD	Alg	Opt			
Reg-Pos (99)	368	0.8	4.8	Yes	240	1	1	1	1	1
Reg-Neg (79)	224	2.0	8.2	No	625	1	1	1	1	1
Iscscprt (15)	10K	3.5	211	Yes	8K	2	2	1	1	1
floppy-pos (12)	17K	5.3	172.5	Yes	9K	2	2	1	1	1
floppy-neg (4)	10K	10.7	154.2	No	17K	2	2	1	1	1
iscsi pos (16)	12K	17.1	358.8	Yes	81K	5	5	2	4	6

Experiments - Terminator examples

	#LOC	G	L	Ans	Getafix			Moped Ver 1	Moped Ver2	Bebop (SLAM / SDV)
					TIME (s)			TIME(s)	TIME(s)	TIME(s)
					BDD	Alg	Opt			
Term-A iterative	284	9	97	Yes	141K	4	3	----	1	4
schoose					162K	3	3	14	1	3
Term-B iterative	471	14	77	No	997K	72	12	----	----	----
schoose					660K	32	9	24	490	953
Term-C iterative	908	10.7	154.2	No	48K	2	3	1	1	711
schoose					48K	2	3	2	4	709

Windows NT Bluetooth example

Two processes (one adder; one stopper) 12 local vars; 8 global vars

# context switches	Reachable?	Max BDD size	Time (sec)
1	No	0.6K	5.4
2	No	1.5K	5.5
3	No	5.5K	5.7
4	No	7.0K	5.8
5	No	13.3K	6.3
6	No	23.2K	7.3

Two processes (two adders; one stopper) 18 local vars; 8 global vars

# context switches	Reachable?	Max BDD size	Time (sec)
1	No	0.7K	5.5
2	No	2.6K	5.6
3	No	10.4K	6.0
4	Yes	41.1K	7.6
5	Yes	90.9K	9.1
6	Yes	250.1K	9.1

Experiments - DDVERIFY benchmark (1 copy of kernel and unboundedly many copies of driver)

(4 rounds)	# checked prop.	Avg. # LOC per pgm	Avg. # shared vars per pgm	Avg. Time (s) on successful pgm	# of pgm on which tool terminated	# of pgm that timed-out
vdt_pci (1195 pgm)	217	9K	30	4	541 (45,3%)	644 (54.7%)
wdt (1967 pgm)	351	7K	29	3	1084 (55,1%)	883 (44.9%)
machzwd (683 pgm)	255	8K	18	9	533 (78%)	150 (22%)
sbc60xxwdt (375 pgm)	106	5K	31	6	375 (100%)	
i8xx tco (766 pgm)	338	11K	22	4	645 (84.2%)	121 (15.8%)
Total (4986 pgm)					(72.5%)	(27.5%)

Outline of the talk

Think using fixed-points.
Try *Getafix*

www.cs.uiuc.edu/~madhu/getafix

- Fixed-point calculus
- Reachability
 - recursive Boolean programs
 - concurrent Boolean programs
 - parameterized Boolean programs
- Tool GETAFIX
- Conclusions

Conclusion

- Experience:
 - Succinctness
 - Easy to check correctness
 - No hidden heuristics; all ideas explicitly expressed
 - Extremely easy to program
- Open problem:
 - Can we do efficient context-sensitive analysis using *Getafix*?

Related research

- bddb [Lam et al, PODS'05]
 - Datalog to express program analysis properties
 - BDD-based implementation of analysis
 - No algorithmic control possible; purely declarative
 - Uses many complex heuristics... hard to know what's happening
 - We tried encoding Boolean pgs into bddb (it's possible)
 - Complete failure... even simple programs don't finish.
- Jedd [Lhotak -Hendren, PLDI'04]
 - Extension of Java with bdd-based relations
 - Very attractive; however, no fixed-points

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-Schwoon, 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 (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, CAV'09]: 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]

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 also a variant of the original algorithm
- [Lal-Reps, CAV'08]: symbolic algorithm computing procedure summaries lazily (not implemented)
- Algorithm presented today [La Torre-Madhusudan-Parlato, PLDI'09]

Linear interfaces

- [Cobleigh-Giannakopoulou-Pasareanu, TACAS'03]:
 - compositional verification with interfaces
 - interfaces are finite transition systems
 - do not scale for verification of parametric programs
- compositional approach based on linear interfaces [La Torre-Madhusudan-Parlato]

Parameterized programs

- [Atig-Bouajjani-Qadeer, TACAS'09]: dynamic creation of threads, number of activations of each thread is bounded
- exploit symmetry in model-checking [Emerson-Sistla, CAV'93]
- counter abstraction [Pnueli-Xu-Zuck, CAV'02]
- counter abstraction with Cartesian representation of local and global state for verifying Linux device drivers [Basler-Mazzucchi-Wahl-Kroening, CAV'09]

Parameterized programs

verification of network invariants (rich literature):

- local analysis assuming global invariants for the environment [Kesten-Maler-Marcus-Pnueli-Shahar, CAV'97]
- techniques based on regular model-checking [Kesten-Pnueli-Shahar-Zuck, CONCUR'02]
- adaptation of Owicki-Gries technique with particular form of invariants followed by abstraction and model-checking [Cohen-Namjoshi, CAV'08]
- in some classes it is sufficient to look at finitely many threads [Emerson-Kahlon, CSL'04]
- abstraction of network invariants [Clarke-Grumberg-Jha, CONCUR'95]