Constraints in Verification

Andreas Podelski

University of Freiburg

two kinds of constraints in verification

1. “upper bound for fixpoint” constraint over sets of states

$$\bot \subseteq X \;\wedge\; F(X) \subseteq X \;\wedge\; X \subseteq \text{bound}$$

verification ⇔ least-fixpoint check ⇔ constraint problem

2. constraint denoting a set of states

used in abstract fixpoint checking

- abstraction ⇔ entailment between constraints
- fixpoint test ⇔ entailment between constraints

constraint, no programming

- “just declare it!“
  – define the set of desired solutions
- “logic and control“
  – algorithmic meaning of logical connectives

constraint as a data structure
in constraint programming, CLP, ccp, ...

- relation (set of n-tuples, n≥1)
- formula (n free variables , n≥1)
- data structure with operations:
  - test satisfiability
  - compute solution
  - test entailment
  - add conjunct (... still satisfiable?)
  - add disjunct (... now entailed?)

compute set of solutions

- transform into an (equivalent) normal form
  - normal form may be: *false*

or:

- search for a solution
  - may find out that no solution exists

- finite-model checking:
  - constraint solving = search
  - solution = (“bad“) state
  - correctness = absence of solution

- program verification:
  - constraint solving = transformation
  - solution = set of (reachable) states
  - correctness = set contains no bad state
  - set ≈ Floyd-Hoare annotation, i.e.,
    control flow graph labeled by constraints (“assertions“)

we cannot verify a program through failure of search

finite-model checking for parallel systems

- finite-model checking is linear in size of model

but ...

- input = parallel composition of *n* components
  - model uses exponential space
  - model checking = search =>
    model need not be constructed explicitely =>

  finite-model checking is PSPACE

  (in *n*, the number of components)

verification of parallel programs

- proof requires Floyd-Hoare annotation,
  i.e., control flow graph labeled by constraints (“assertions“)

- control flow graph uses exponential space
  (product of *n* control flow graphs)
  even just the reachable part generally uses exponential space

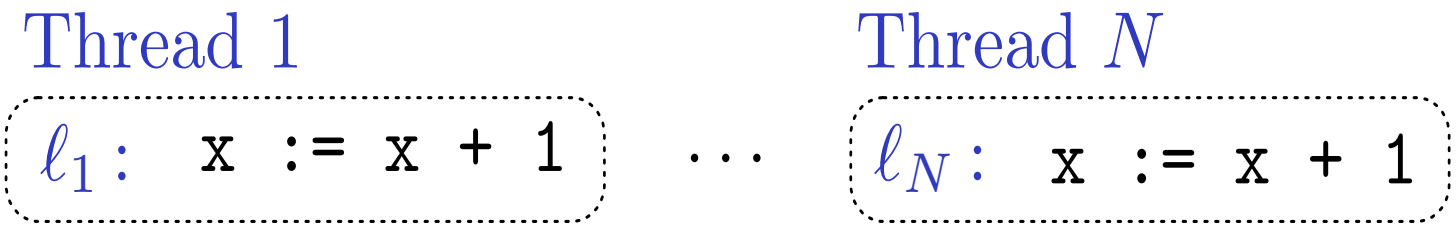
verification of parallel programs in PSPACE

two steps:

1. construct
   data flow graph with Floyd-Hoare annotation
   - denotes set of correct traces
     i.e., a regular linear temporal property ***P***
     represented by (alternating) finite automaton
2. model checking
   control flow graph = finite model ***M***

$$\boldsymbol{M} \models \boldsymbol{P}$$

{ x = 0 }



{ x ≤ N }

exponental-size control flow graph with
Floyd-Hoare annotation uses N assertions

- data flow graph with Floyd-Hoare annotation denotes set containing all traces of form below (of length ≤ N)

  x := x+1

  .

  .

  .

  x := x+1

correct: satisfy Hoare triple {x=o} ... {x≤ N}

bakery algorithm

Thread A

a_1 : `e1 := true`
a_2 : `tmp := n2`
a_3 : `n1 := tmp + 1`
a_4 : `e1 := false`
a_5 : `[¬e2]`
a_6 : `[¬(n2 ≠ 0 ∧ n2 < n1)]`
`// critical section`
a_7 : `n1 := 0`

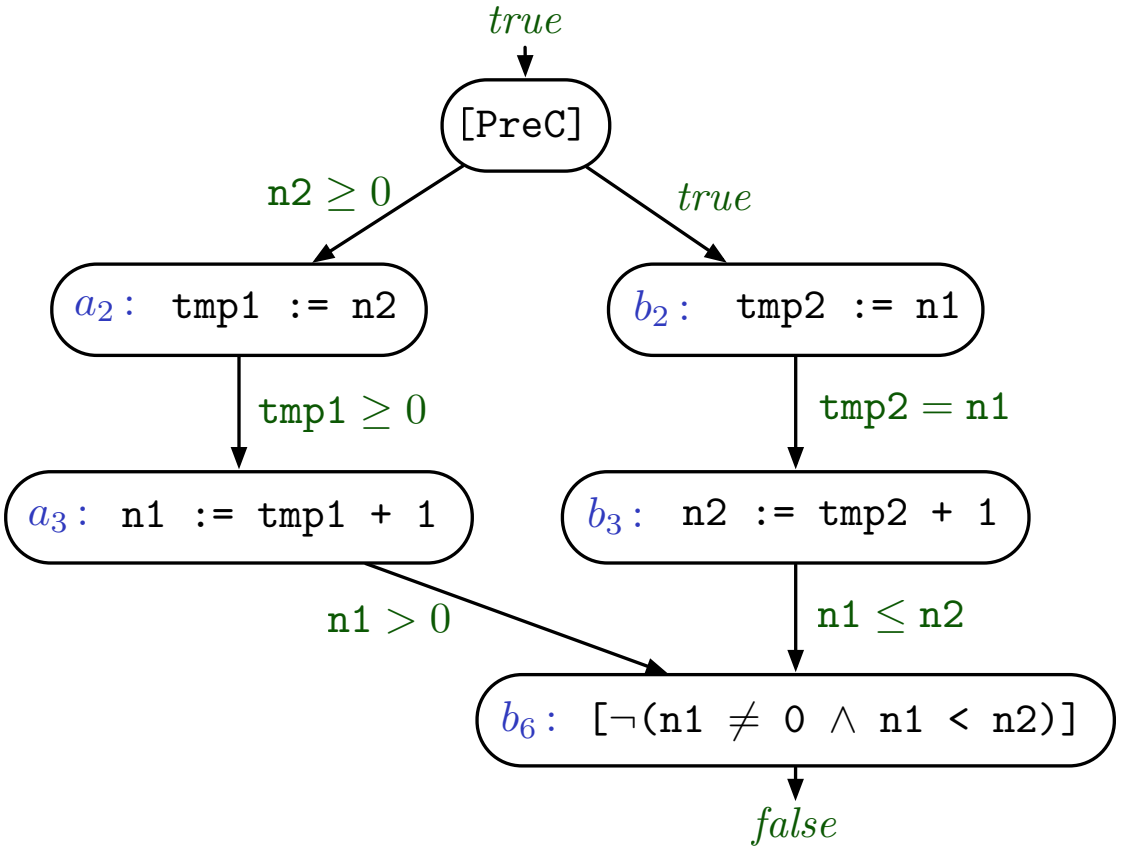
Thread B

b_1 : `e2 := true`
b_2 : `tmp := n1`
b_3 : `n2 := tmp + 1`
b_4 : `e2 := false`
b_5 : `[¬e1]`
b_6 : `[¬(n1 ≠ 0 ∧ n1 < n2)]`
`// critical section`
b_7 : `n2 := 0`

$$\texttt{PreC} \equiv \texttt{n1} = 0 \wedge \texttt{n2} = 0 \wedge \texttt{e1} = \texttt{false} \wedge \texttt{e2} = \texttt{false}$$

Trace 1

	[PreC]
a_1 :	e1 := true
a_2 :	tmp1 := n2
a_3 :	n1 := tmp1 + 1
a_4 :	e1 := false
a_5 :	[¬e2]
a_6 :	[¬(n2 ≠ 0 ∧ n2 < n1)]
b_1 :	e2 := true
b_2 :	tmp2 := n1
b_3 :	n2 := tmp2 + 1
b_4 :	e2 := false
b_5 :	[¬e1]
b_6 :	[¬(n1 ≠ 0 ∧ n1 < n2)]



Step 1. Construction of DFG with Floyd-Hoare annotation

repeat until G is *“large enough“*

pick trace of program, say: a_1, ..., a_m

construct Hoare triples
{Pre} a_1 {P_1} ... {P_m-1} a_m {Post}

create node for each action
label edge between nodes by
“local“ conjuncts of assertions

merge resulting DFG with G

verification of parallel programs in PSPACE

two steps:

1. construct
   data flow graph with Floyd-Hoare annotation
   - denoteing ***set of correct traces***
     i.e., a regular linear temporal property ***P***
     represented by finite automaton
2. model checking
   control flow graph = finite model ***M***

$$M \models P$$

construct automaton that accepts traces

$a_1 \ldots a_n$

sucht that

$\{Pre\}\ a_1 \ldots a_n\ \{Post\}$

- state for each assertion $P_1, \ldots, P_m$
- transition from state P_k to P_j for letter a_i if

$$\{P_k\}\ a_i\ \{P_j\}$$

end of excursion

back to theme of this talk

constraints in verification

two notions of constraint solving,
search and transformation

- finite-model checking:
  - constraint solving = search
  - solution = (“bad“) state
  - correctness = absence of solution

- program verification:
  - ***constraint solving = transformation***
  - solution = set of (reachable) states
  - correctness = set contains no bad state
  - set ≈ Floyd-Hoare annotation, i.e.,
    control flow graph labeled by constraints (“assertions“)

compute set of solutions

- transform into an (equivalent) normal form
  - normal form may be: *false*

or:

- search for a solution
  - may find out that no solution exists

constraints over sets of strings

$$X = a.b.X + a.b$$

$(a.b)^*a.b$ is smallest solution for X

automata = constraints over sets of strings

$q1(x) \leftarrow x=a.y, q2(y)$

≈ transition of automaton
from state q1 to state q2, reading letter a

automata are constraints in normal form

pushdown systems are constraints over sets of strings

pop: $q1(a.y) \leftarrow q2(y)$

push: $q1(x) \leftarrow q2(a.x)$

model checking pushdown system =
transforming constraint into normal form

tree automata are constraints over sets of trees

- $q(x) \leftarrow x = f(x1, x2),\ q1(x1), q2(x2)$

  equivalent notation:

  $q(f(x1, x2)) \leftarrow q1(x1), q2(x2)$

- set constraints
  - $q \supseteq f(q1, q2)$
  - $q(x) \leftarrow q1(f(x, \_))$

  set-based analysis:
  transform set constraint into normal form

Constraint-based Model Checking

- CLP program = constraint over set of states
- model checking = constraint solving via CLP engine

- manipulation of sets of states (over, say, integers)
  = operations on constraints (over integers)
  i.e., on data structure of CLP engine

Verification Algorithm

input:

- program
- correctness property
  - non-reachability, termination

output:

- yes, no, don't know
- no output (verification algorithm does not terminate)
- necessary or sufficient pre-condition (P.,Rybalchenko,Wies-CAV'08)
- quantitative information ("how far from correct is the program?")

Program Correctness

- Non-reachability
  - validity of invariant
  - safeness: “assert” does not fail
  - partial correctness {..} P {..}
  - safety properties

- Termination
  - validity of “intermittent assertions”
  - total correctness
  - liveness properties

Least-Fixpoint Checking

- program semantics ⇔ least fixpoint of operator F
- correctness property ⇔ bound
- check:

  least fixpoint of F ⊆ bound ?

- solve constraint in variable X over sets of states:
-

$$\bot \subseteq \mathbf{X} \quad \wedge \quad \mathbf{F(X) \subseteq X} \quad \wedge \quad \mathbf{X \subseteq bound}$$

from now on: “upper bound on fixpoint” constraint

Non-Reachability = Least-Fixpoint Checking

- set of reachable states = lfp(post)
  - least fixpoint of post operator
  - lattice of sets of states
  - order " $\subseteq$" = set inclusion
  - bottom = set of initial states
- non-reachability of bad states $\Leftrightarrow$ lfp(post) $\subseteq$ {good states}
- "upper bound on fixpoint" constraint:

$$\text{lfp(post)} \subseteq X \quad \wedge \quad X \subseteq \{\text{good states}\}$$

constraint solving via iteration of abstract fixpoint checking

Fixpoint Checking ⇔ Constraint Solving

- "upper bound on fixpoint" constraint:

$$\text{lfp}(\text{post}) \subseteq X \quad \wedge \quad X \subseteq \{\text{good states}\}$$

- constraint solving via iteration of abstract fixpoint checking

- "lower bound on fixpoint" constraint:

$$X \subseteq \text{lfp}(\text{post}) \quad \wedge \quad \text{not}(X \subseteq \{\text{good states}\})$$

- constraint solving via bounded model checking

Verification

- construct X such that lfp(post) $\subseteq$ X
- check X $\subseteq$ {good states}
- semi-test: definite Yes answers, don't know No answers

- solve "upper bound for fixpoint" constraint by co-semi-algorithm
- construct sequence $X_1 > X_2 > ... > X_n$ iterating semi-test
  - lfp(post) $\subseteq X_i$
  - $X_i \subseteq$ {good states} ?
  - $X_n \subseteq$ {good states} (X_n being the first with this property)

Next in this Talk: Co-Semi-Test

- construct $X \subseteq \text{lfp(post)}$
- check $X \subseteq \{\text{good states}\}$
- co-semi-test: definite No answers, don't know Yes answers

- solve "lower bound for fixpoint" constraint by co-semi-algorithm
- construct sequence $X_1 \subseteq X_2 \subseteq \ldots \subseteq X_n$ iterating co-semi-test
  - $X_i \subseteq \text{lfp(post)}$
  - $X_i \subseteq \{\text{good states}\}$)
  - not($X_n \subseteq \{\text{good states}\}$) ($X_n$ being the first with this property)

Co-Semi-Test: Bounded Model Checking

- X $\subseteq$ lfp(post) $\wedge$ not(X $\subseteq$ {good states})
  constraint in set variable X

- X := post^k({initial states}) ... $\subseteq$ lfp(post)
  = {states reachable in 0, 1, ..., k steps})

- s $\in$ post^k({initial states}) $\wedge$ s $\in$ {bad states}
  constraint in state variable s
  state = valuation of program variables x, y, z

- post^k(init) $\wedge$ bad
  constraint in (renamings of) program variables x, y, z

Constraint = Set of States

- state = valuation of program variables x, y, z

- constraint denotes set of its solutions
- constraint in variables x, y, z denotes {states}
- constraints init, good, bad
  denoting: {initial states}, {good states}, {bad states}

- post = operator over sets of states
  = operator over constraints

Transition Constraint = Set of Transitions

- pair of states = valuation of variables x, y, z, x', y', z'
- transition = (pre-state, post-state)
- program statement = transition relation
  = transition constraint
  if x>0 then x:=x+1 = x>0 ∧ x'=x+1
- post(x>10) = ∃x. x>10 ∧ x>0 ∧ x'=x+1
  = x'>11

Falsification = Constraint Solving with Search

- $post^k(init)$ = “big” disjunction of constraints

- if constraint (in program variables):

  $post^k(init) \wedge bad$

  is satisfiable

  then constraint (in set variable):

  $X \subseteq lfp(post) \wedge not(X \subseteq \{good\ states\})$

  is satisfiable

  (since $X = post^k(init)$ is a solution)

  ... and we have a definite No answer

That’s the best what constraint solving with search can do for programs
(as opposed to: for finite models)

Done for this Talk: Co-Semi-Test

- construct $X \subseteq \text{lfp(post)}$
- check $X \subseteq \{\text{good states}\}$
- co-semi-test: definite No answers, don't know Yes answers

- solve "lower bound for fixpoint" constraint by co-semi-algorithm
- construct sequence $X_1 \subseteq X_2 \subseteq \ldots \subseteq X_n$ iterating co-semi-test
  - $X_i \subseteq \text{lfp(post)}$ ... simply set $X_i = \text{post}^k(\text{init})$
  - not($X_i \subseteq \{\text{good states}\}$) ?
  - not($X_n \subseteq \{\text{good states}\}$) ($X_n$ being the first with this property)

Verification = Constraint Solving via Search

- construct X such that lfp(post) $\subseteq$ X
- check X $\subseteq$ {good states}

- semi-test: definite Yes answers, don't know No answers

- solve "upper bound for fixpoint" constraint by search
- construct sequence $X_1 > X_2 > ... > X_n$ iterating semi-test
  - $X_i >$ lfp(post)
  - $X_i \subseteq$ {good states} ?
  - $X_n \subseteq$ {good states} (X_n being the first with this property)

Abstract Fixpoint Check ⇒ Constraint Solving

- "upper bound on least fixpoint" constraint in set variable X:

$$\text{lfp}(X) \subseteq X \ \wedge \ X \subseteq \{\text{good states}\}$$

- semi-test: try any fixpoint X of post:

$$\text{post}(X) \subseteq X \ \wedge \ \{\text{initial states}\} \subseteq X$$

  and check $X \subseteq \{\text{good states}\}$

- "upper bound on least fixpoint" constraint in set variable X becomes:

$$\text{post}(X) \subseteq X \ \wedge \ \{\text{initial states}\} \subseteq X \ \wedge \ X \subseteq \{\text{good states}\}$$

methods to solve above constraint

1. abstraction to simpler constraint problem
2. abstract fixpoint checking

Abstraction to Set Constraint Problem

to solve "Upper Bound on Fixpoint" Constraint

$post(X) \subseteq X \wedge \{initial\ states\} \subseteq X \wedge X \subseteq \{good\ states\}$

- n = number of program variables $\Rightarrow$ X ranges over sets of n-tuples
- set constraint: X ranges over Cartesian products (of n sets)
- set-based analysis for programs over lists, stacks and trees
  = solving set constraints (Reynolds, Jones, Gallagher, ...)
  = abstract fixpoint iteration (Cousot'92)

Abstraction to Linear Constraint Problem

to solve “Upper Bound on Fixpoint” Constraint

$post(X) \subseteq X \wedge \{\text{initial states}\} \subseteq X \wedge X \subseteq \{\text{good states}\}$

- solution for X = set of states
- set denoted by linear constraint over program variables with coefficients as parameters
- “Upper Bound on Fixpoint” Constraint translates to linear constraint over coefficients

- Bradley, Colon, Manna, Sipma, Sankaranarayanan, Tiwari, Rybalchenko, ...
- does not work well with features of realistic programs, until now
- does not scale well, until now

Fixpoint Iteration

to solve “Upper Bound on Fixpoint” Constraint

$post(X) \subseteq X \wedge \{initial\ states\} \subseteq X \wedge X \subseteq \{good\ states\}$

- construct solution for X in $post(X) \subseteq X \wedge \{initial\ states\} \subseteq X$
- generate sequence of constraints init, c_1, c_2, c_3, ..., c_n
  - $init \subseteq c_1 \subseteq c_2 \subseteq c_3 \subseteq ... \subseteq c_n$
  - $post(c_n) \subseteq c_n$ $\quad$ c_n is fixpoint

two issues with naive fixpoint iteration:

- if $c_{i+1} = post(c_i)$ then in general no convergence
- fixpoint test “$post(c_n) \subseteq c_n$” = entailment test : too expensive

Abstraction

in model checking vs. abstract interpretation

- abstraction to finite-state system (“partitioning”)
  works only for finite-state systems

- finite-state abstraction does not preserve termination
  of program with executions of unbounded length

- instead: abstract the functional in the fixpoint iteration
  => abstract least fixpoint checking

Abstract Fixpoint Iteration

- "accelerated" sequence of constraints init, c_1, c_2, c_3, ..., c_n
  - init $\subseteq c_1 \subseteq c_2 \subseteq c_3 \subseteq \ldots \subseteq c_n$
  - post(c_n) $\subseteq c_n$
- after each application of post operator, extrapolation " $\triangleright$ " of result
  - init , post(init) $\triangleright$ c_1 , post(c_1) $\triangleright$ c_2 , post(c_2) $\triangleright$ c_3 , ...
- fixpoint test ("post(c_n) $\subseteq c_n$") in new ordering between constraints e.g.,
  - local entailment: each disjunct entailed by one of disjuncts
  - ordering in free lattice, i.e., ordering between sets of bitvectors (bitvector presents conjunction of n possibly negated base constraints)
- formalized in abstract interpretation (Cousot,Cousot'77)

Abstraction

- widening
  - syntactic criteria to obtain “some” weaker constraint
  - fixpoint test uses entailment ordering between constraints: c ⇒ c’
- best abstraction in abstract domain
  - abstract domain = given (finite) set of constraints
  - c ➪ conjunction of all c’ in abstract domain that are entailed by c
  - thus, to extrapolate c, we need to go through all c’ in abstract domain and test entailment c ⇒ c’
  - fixpoint test cheap:
    c1 smaller than c2 if every conjunct of c2 occurs in c1
    ordering not too restrictive if taken between “best abstractions”
- constraint solving effort: pay now or pay later!
  either in extrapolation or in fixpoint test

State-Predicate Abstraction

- abstract domain = finite set
  - ... of disjunctions of conjunctions of predicates
  - conjunction of predicates = abstract state
  - predicate = base constraint

- Cartesian abstraction
  - post(conjunction) = smallest conjunction above disjunction
  - ... $= \bigwedge \{ \text{predicate} \mid \text{conjunction} \Rightarrow \text{wp(predicate)} \}$
  - avoids exponential explosion
  - uses wp (weakest precondition) instead of post

Termination = Least-Fixpoint Checking

- transitive closure of transition relation = lfp(o)
  - operator “o” = composition of two relations
  - lattice of sets of pairs of states
  - order “ $\subseteq$” = set inclusion
  - bottom = transition relation

- termination

  $\Leftrightarrow$

  lfp(o) $\subseteq$ finite union of well-founded relations

Termination

$\Leftrightarrow$

lfp(o) $\subseteq$ finite union of well-founded relations

- assume: exists infinite computation s_1, s_2, ...
- each (s_i, s_j) where $i<j$ belongs to lfp(o)
- ... hence to one of the relations in finite union
- one of the relations contains infinitely many pairs
- even: infinitely many consecutive pairs (Ramsey)

- contradiction: all relations in union are well-founded

Well-foundedness of Transition Constraints

- transition constraint (no disjunction!)
  = conjunction of guard and action

  $x>0 \ \wedge \ x'=x+1$

- simple while loops

  while(x>0){ x := x+1 }

- decidable/efficient termination check (Tiwari,P., Rybalchenko)
  Farkas' Lemma + linear arithmetic constraint solving

From Trace Semantics to Relational Remantics

- state of recursive program
  = valuation of program variables + stack value
- trace defined by states with stack
- no good abstraction for stack as data structure
- no good abstract fixpoint construction

- circumvent issue:
  switch from trace semantics to relational semantics
- procedure summary:
  relation between entry and exit states (Sharir,Pnueli'81)
- refined procedure summary:
- relation between reachable entry and exit states (Reps/Horwitz/Sagiv'95)

Summary = Least Fixpoint

- transitive closure of transition relation including:
  - call (pass actual to local variables)
  - return (new value of globals, old value of locals)
- restrict transitive closure relation to domain of reachable entry states

- summary = lfp(o)
  - operator “o” = composition with transition relation + seeding
    add pair of identical entry state when it appears in new pair
    (_ , s) ∈ summary, s entry state ⇒ (s, s) ∈ summary
  - lattice of relations
  - ⊥ = identity relation on initial states

- non-reachability of bad states

  ⇔

  lfp(o) ⊆ {initial states} × {good states}

Verification of Recursive Programs = Solving Set Constraints

- fixpoint equation for post = set constraint
  - post operator on sets of stack states
  - stack state = unary tree
  - push = application of function symbol
  - pop = application of projection

- set constraint solving ≈ computing summaries
  - canonical rewrite systems (Buchi)
  - interprocedural analysis (Knoop, Steffen, Reps, Horwitz, Sagiv)
  - pushdown systems (Bouajjani, Esparza, Maler, ...)
  - cryptographic protocols (Dolev/Yao)
  - empirical evaluation (Kodumal, Aiken)

Conclusion

1. “upper bound for fixpoint” constraint over sets of states

$$\bot \subseteq X \quad \wedge \quad F(X) \subseteq X \quad \wedge \quad X \subseteq \text{bound}$$

verification ⇔ least-fixpoint check ⇔ constraint problem

2. constraints over integers etc. denote sets of states
   used in abstract fixpoint checking
   - abstraction ⇔ entailment between constraints
   - fixpoint test ⇔ entailment between constraints