Exploiting Shared Structure in Software Verification Conditions

Domagoj Babic and Alan J. Hu
University of British Columbia

Outline

- Introduction
- Basic definitions
- Exploiting shared structure
- Preliminary experimental results
- Future work

Goal

- Software checking tools
  - Produce a long sequence of queries (tens, hundreds of thousands)
  - Frequently some sharing (common sub-expressions) among adjacent queries
- Exploit that sharing
  - Faster solving of a sequence of queries (verification conditions)

Verification conditions (VCs)

- Logical formulas
  - Constructed from a system and desired correctness properties
  - Validity of VCs corresponds to the correctness of the system (or its abstraction)

Trend towards automation...

- Proving validity
  of VCs automatically:

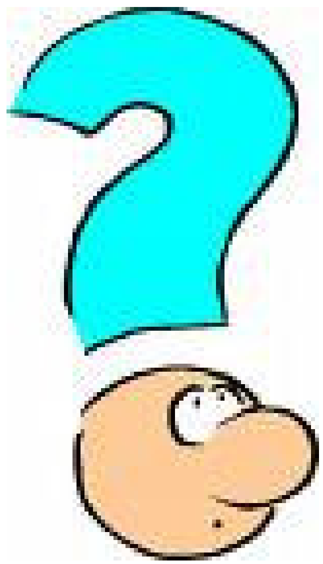
  - Avoids manual effort
  - Has its limitations
    - Computability
    - Performance often unacceptable
      (especially when checking large real-world
      software)

How do we improve performance of decision procedures?

- Algorithmical improvements
  - Faster algorithms
    (e.g. watched literals in SAT solvers)
- Better heuristics
  - Usually result of better understanding of the problem
- Learning techniques
  - Avoid redundant work
- Automated tuning [Hutter et al., FMCAD '07]
  - Automated finding of good combinations of search parameters
- Exploiting structure of problems

Exploiting structure in software checking

Coarse-grained



Fine-grained

- Libraries change less often than other code [Rountev et al. ’06]
  - Pre-analyze libraries
- Shared code among different versions [Conway et al. ’05]
  - Analyze only modified code and its cone of influence
- Structural abstraction [Babic, Hu ’07]
  - Abstract function calls
- What is next?

Inter-VC sharing

- Most software checking tools produce a large number of queries
  - Extended static checkers
  - Testing tools
- Generated queries often share some subexpressions (especially adjacent queries)
- How about exploiting this inter-VC sharing?

A naïve approach

- Construct a large disjunction
- If certain VC is not valid, add a clause that blocks it

Why is naïve approach a bad idea?

- Blocking clause does **not** stop the solver completely from analyzing at least part of the search space corresponding to blocked verification conditions
- All VCs don't fit in the memory
- Only a small percentage of learned facts can be kept around (e.g. learning in SAT solvers)
- Not all learned facts are re-usable (context-dependency)
- In our setting, future VCs are not known (constructed on-the-fly through structural abstraction)

What is needed

- A fast technique to identify
  - Context-independent facts
  - In online manner (future VCs not known)
  - Compatible with standard decision procedures
    (in our case bit-vector theorem prover, based on a SAT solver)

Outline

- Introduction
- Basic definitions
- Exploiting shared structure
- Preliminary experimental results
- Future work

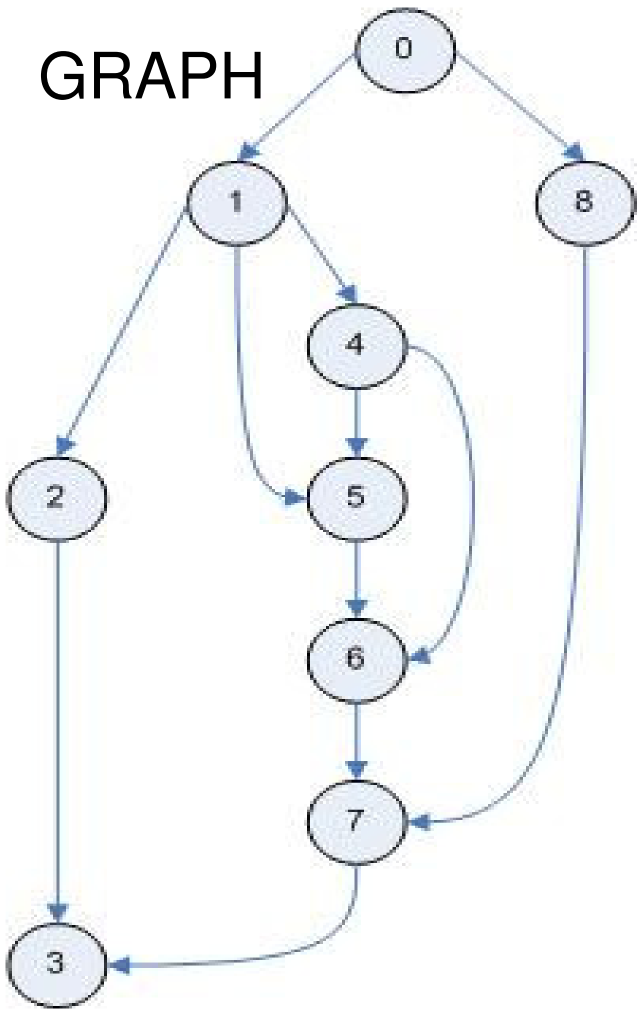
Dominance

Definition [Dominance relation]

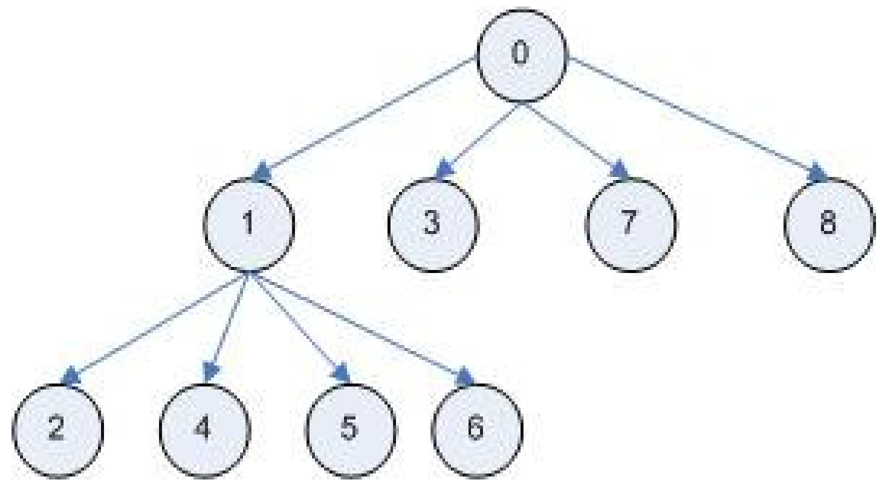
A node *n* dominates node *m* if and only if all the paths from the root of the graph to *m* go through *n*, written as $n \geq\geq m$.

Dominance Example

GRAPH



DOMINATOR TREE

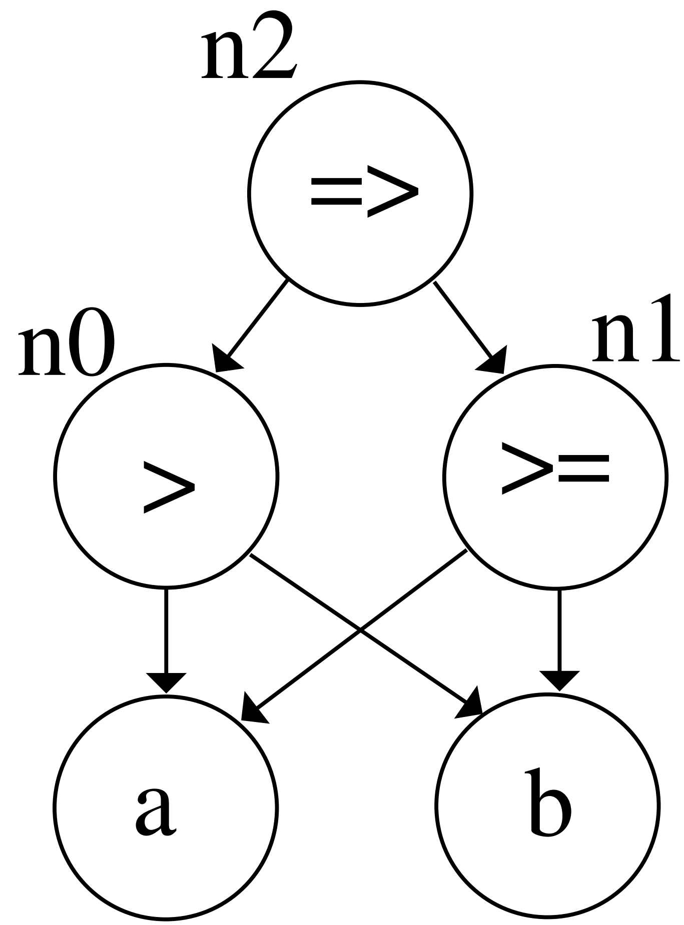


Maximally shared graph

- An acyclic graph
- Nodes represent constants, variables, and operators
- Common subexpressions eliminated
- A non-canonical representation
  (can be close if a solid term rewriting is used)

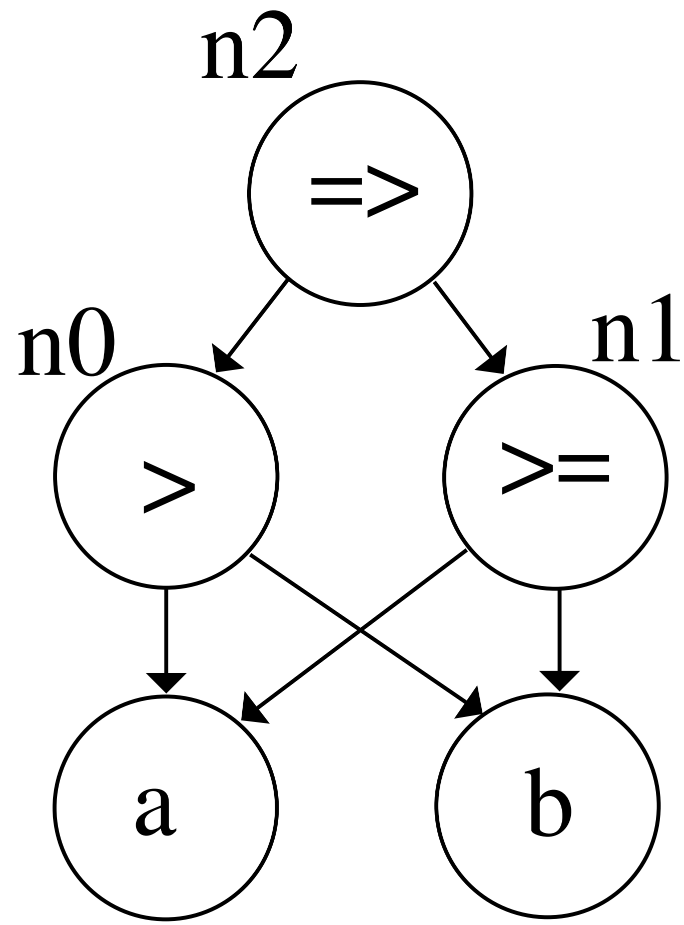
Maximally shared graph Example

Expression:
(a > b) => (a >= b)



Logical consistency of max.-shared graphs

- Max. shared graphs
  - Represent circuits
- Circuits
  - For any input produce output
  - Always logically consistent
- Validity proven by:
  - Forcing output (=>) to false
  - Proving the expression UNSAT
- Forcing an output to certain value
  - Can cause inconsistency



Context-independence (last def!)

- A node *n* in max. shared graph is fixed by the decision procedure
  - If the decision procedure derives invariant *n==constant*
  - Written:
    - $fix_{DP}(n) = true$
    - $FixVal_{DP}(n) = constant$

- An invariant (derived by a decision procedure) is context-independent
  - If it is uniquely implied by its sub-expressions
  - Otherwise, invariant is context dependant

Assumptions required for the presented technique

1) VCs are maximally shared graphs (acyclic)
   - Routinely satisfied in practice, if not satisfiable with some pre processing (common subexpression elimination)
2) Decision procedure must be able to identify invariants of the form *var == constant*
   - E.g. learned unit literals are such facts
3) Complete propagation of equalities
   - E.g. *a=7,b=7,c=7* instead of *a=7, b=a, c=b*
   - Trivial to satisfy with some amount of post processing
4) Proper subexpressions of a VC are logically consistent
   - Ensures that the implicants derived from a subexpression are meaningful (anything can be derived from false)

Outline

- Introduction
- Basic definitions
- Exploiting shared structure
- Preliminary experimental results
- Future work

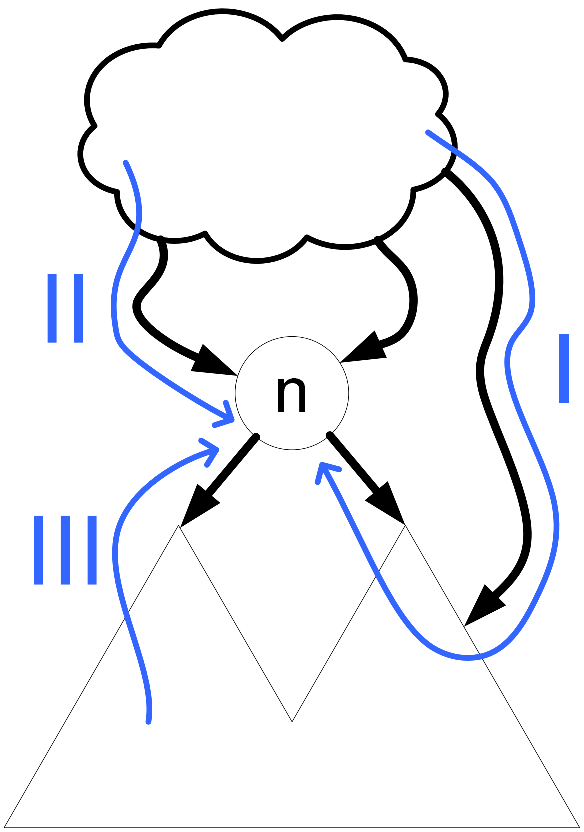
Computing context-insensitive invariants

- Precise:
  - Recording proofs
  - For SAT solvers, that means implication graphs
  - Too expensive (computationally)

- Approximated:
  - Reconstruction based
  - From the implied invariants *var==constant*
  - Relatively cheap

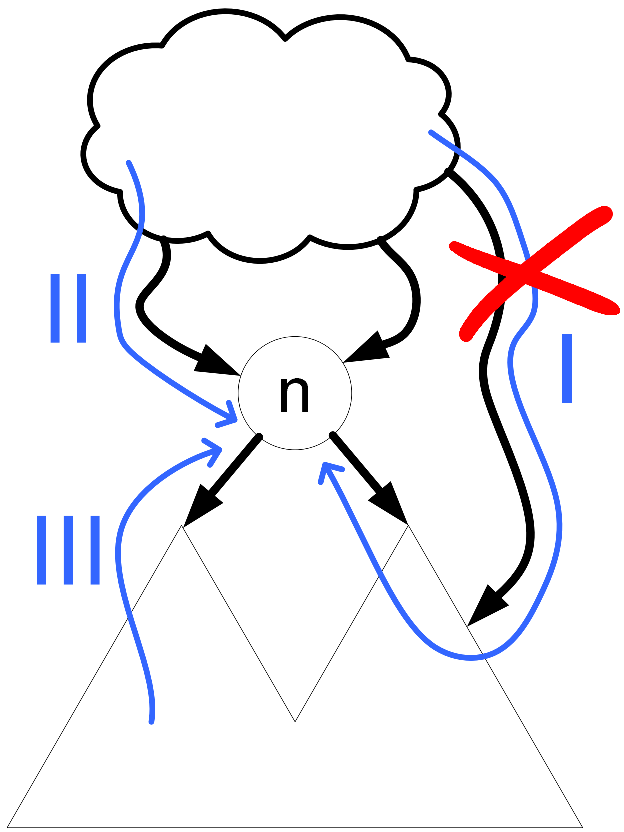
3 types of invariant *n==constant* propagation

Blue lines represent constant propagation chains



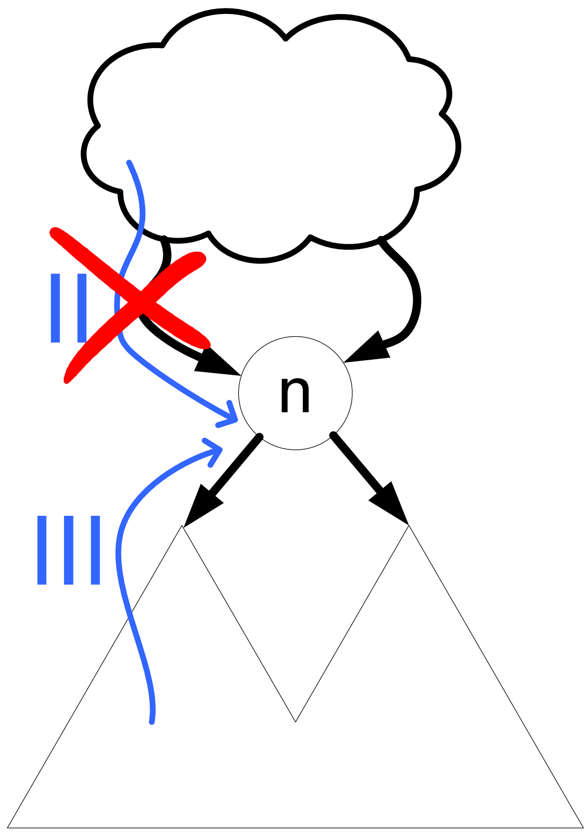
- *I. From above, circumventing the node* (context-dependent)
- *II. From above* (context-dependent)
- *III. From below* (context-independent)

Eliminating context-sensitive invariants of type I



- Check weather *n* dominates all its descendants
- Test: dominance – eliminates (I)

Eliminating context-sensitive invariants of type II



- Check that the chain of implications did not come from *above*
(from its predecessor)
- Test: *n* is fixed, but none of its predecessors is fixed – eliminates (II)
- After eliminating context-sensitive invariants, we are left only with context-insensitive ones

Algorithm – finds a subset of all context-insensitive facts

procedure Fix(*n*, *Fixed*) { // *Fixed* is a table with fixed nodes
    for each successor *s* **do**
        Fix(*s*, *Fixed*)

    if !isRoot(*n*) && isOperator(*n*) && $fix_{DP}(n)$ **then**
        for each descendant *d* **do**
            if !isConstant(*d*) || !($n \underline{\gg} d$) **then**
                return
        for each predecessor *p* **do**
            if $fix_{DP}(p)$ **then**
                return
        $Fixed[n] = FixVal_{DP}(n)$

Complexity

- $O(n^2)$ in the worst case, very pessimistic
- Implementation uses Tarjan-Lengauer ('79) $O(n \log(n))$ algorithm for dominance computation
- Dominance check – constant time (ancestry relation on trees can be established in amortized constant time)

High level algorithm

```
clear table Fixed
for each VC_i do
    C = Translate(VC_i) && VC_i == false
    for each descendant d of VC_i do
        if n exists in table Fixed then
            C = C && n==Fixed[n]
    if Solve(C) == satisfiable then
        report bug
    Fix(VC_i, Fixed) // Learn what you can
```

Outline

- Introduction
- Basic definitions
- Exploiting shared structure
- Preliminary experimental results
- Future work

Preliminary experimental results [obtained with Calysto ext. static checker]

Timeout=300 [s], dual-processor AMD X2 4600+, 2 GB RAM

Benchmark	KLOC	#VCs	Base approach		New approach	
			Time [s]	Timeouts	Time [s]	Timeouts
Bftpd v1.6	4	1130	725.8	0	582.5	0
HyperSAT v1.7	9	1363	5.3	0	5.1	0
Licq v1.3.4	20	2009	199.6	0	214.5	0
Dspam v3.6.5	37	8627	3478.6	8	3157.6	6
Xchat v2.6.8	76	8090	368.5	0	365.8	0
Wine 0.9.27	126	9000	1881.4	2	1266.7	0

Discussion

- Fewer timeouts, somewhat better runtime
- Method is (implementation-wise) complex
- More research needed

Outline

- Introduction
- Basic definitions
- Exploiting shared structure
- Preliminary experimental results
- Future work

Future work

- Better algorithm that discovers complete set of context-independent facts

- Semi-eager expansion that checks k (where k is small) VCs at once using classical disjunction and blocking clauses