

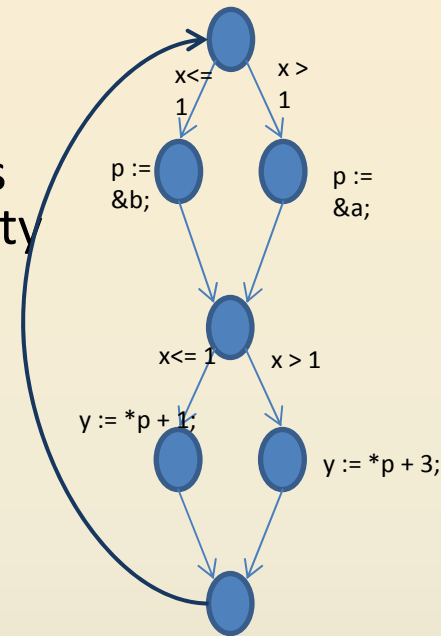
Symbolic Program Analysis using Term Rewriting and Generalization

Nishant Sinha

NEC Labs, Princeton, NJ

Symbolic Execution

- Symbolic Execution
 - Assign **symbolic values** to input variables
 - **Propagate** symbolic values through all program paths using a **decision procedure** for checking path feasibility
- Issues
 - **Path Explosion**
 - Sequence of If-Then-Else: $O(2^{\text{length of sequence}})$
 - Sequence of Loops: $O(\text{product of loop paths})$
 - **Complex Path Conditions**
 - Due to repeated substitutions
 - Handled by **Incremental SMT solvers** effectively
- **Path enumeration based** symbolic execution does not scale
 - Need a method to **merge** symbolic values at join points



Data-flow analysis

- Provides an **uniform way of merging data facts** at join points
 - Join operators for Abstract Domains
 - Work list based algorithms
 - However, most join operators **sacrifice precision** (path-sensitivity) to achieve scalability
- Can we devise an **efficient** symbolic analysis algorithm that **preserves path-sensitivity**, while **avoiding path-enumeration**?

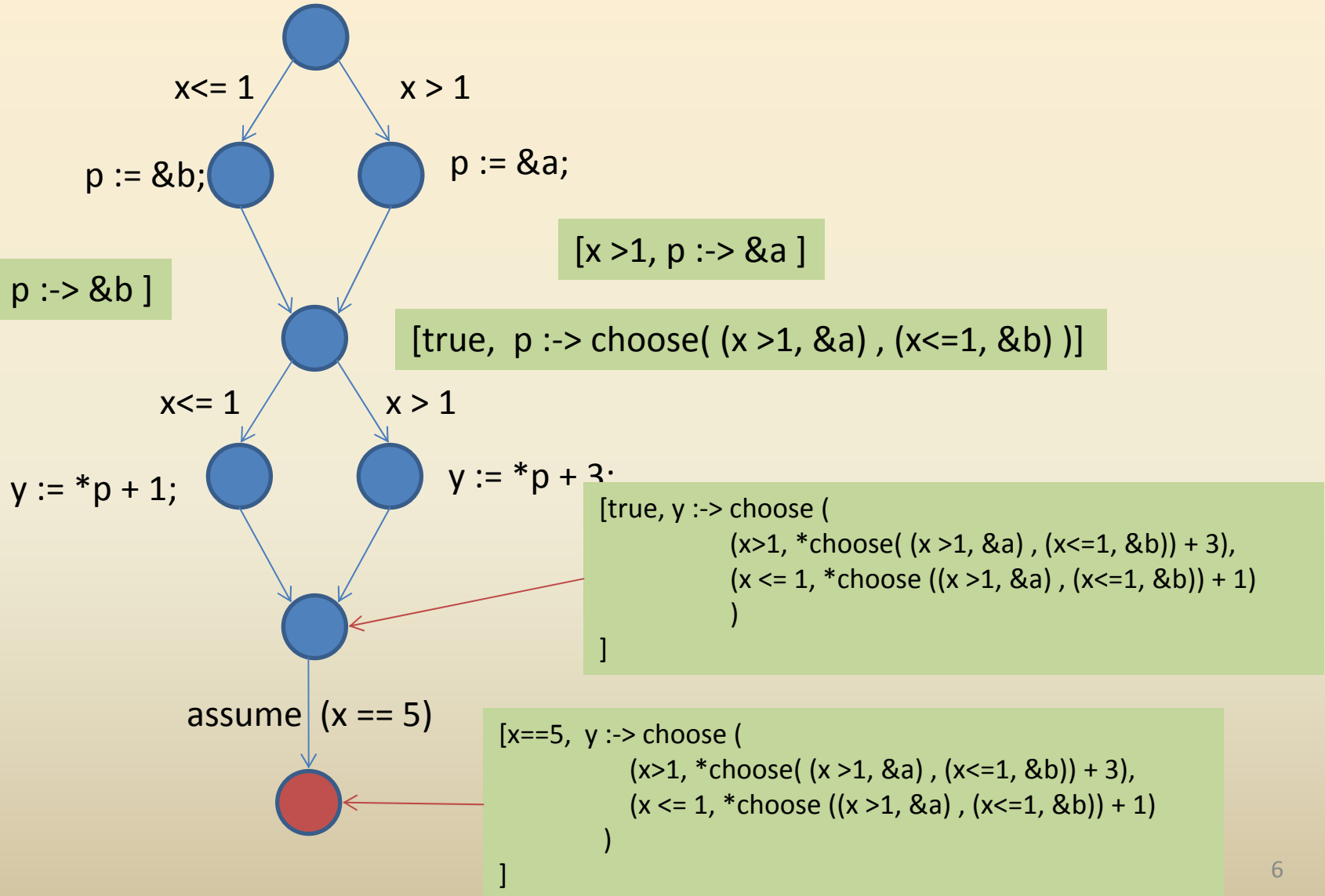
Symbolic Program Analysis (SPA)

- Our Solution
 - **Data-flow analysis** on program expressions (**terms**)
 - State is a condition-value tuple $\langle C, \sigma \rangle$
 - C is the path condition, σ is a map (Var \rightarrow Term)
 - **Merge** states at join nodes
 - Using a **choose** operator
 - Use a decision procedure (an SMT solver) to check for satisfiability of Error path Conditions (ECs)
- Use **Term Rewriting** and **Generalization** to simplify **choose** terms

Background (F-Soft)

- **Integer programs (+bitvector arithmetic)**
 - Each variable assigned an integer address
 - Pointers have integer address values
 - $*p = 3$; translates to “ $v = \text{ite}(p == \&v, 3, v)$ ” for each v in the points-to set of p .
- **Control Flow Graph (CFG) Representation**
 - CFG nodes have statements; edges have constraints
 - Call/Return edges between functions in CFG
- **Approximations**
 - Bounded arrays, heap, recursion
- **Property Checks**
 - Pointer validity, Array bounds violation, String checker, ...
 - CFG instrumented with Error Blocks, **check Reachability**

SPA Example



SPA Algorithm Pseudo-code

SPA () {

//Q is a reverse postorder work list of CFG nodes

while Q is not empty

 n := Q.pop()

 //Handle error if n is an error node

 (C_n, σ_n) := **process_data** (n)

for each successor n' of n

 // edge (n,n') has constraint C

 C'_n := C σ_n ∧ C_n

if C'_n is satisfiable

 Data_{n'} := Data_n ∪ (C'_n,
σ_n)

 Q.insert (n')

}

process_data (n) {

 D := **join_data** (Data_n)

 clear (Data_n)

 //St_n is the list of statements at node n

 D' := **compute_post** (D, St_n)

 return D'

}

join_data (<(C_i, σ_i)>) {

 C := C₁ ∨ ... ∨ C_n

 σ := {x :-> choose (<(C_i, σ_i(x))>) }

 return (C, σ)

}

Queue Priority Ordering

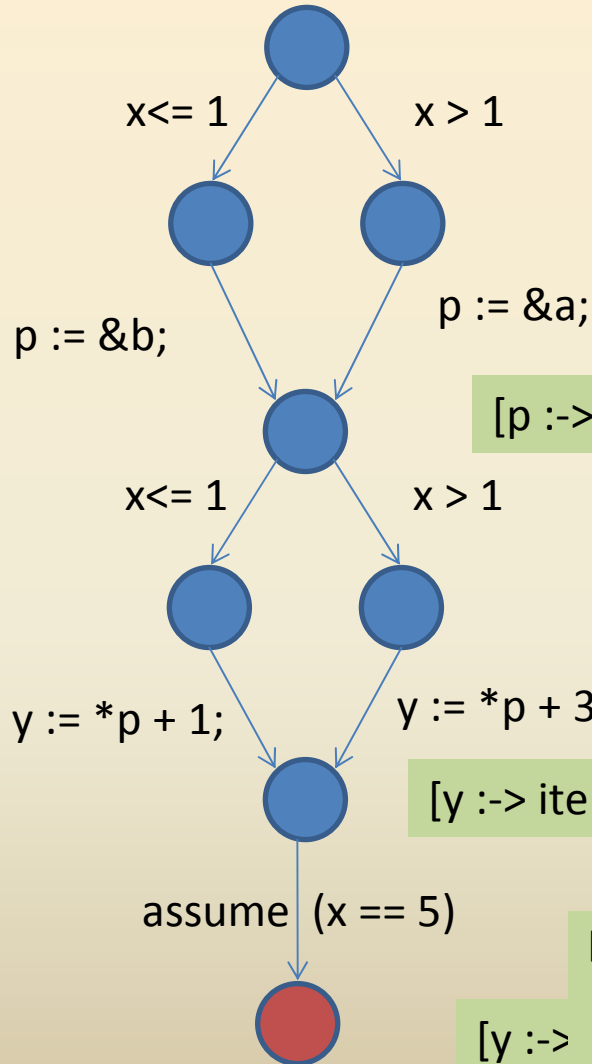
- **Priority order** is essential for avoiding path explosion
- **Reverse post order**
 - All predecessors processed before the current node
 - Problem with loops
- **Weak total order** (Bourdoncle)
 - SCC-based priority ordering
 - All nodes *following* a loop have lower priority than loop nodes

Join leads to Complex Terms

- **choose/ite** terms **blowup** after several joins
- **Nested ite** terms are **problematic** for SMT solvers
 - Exponential search space in worst case
- However, in many cases, one can exploit the **ite** term structure to **simplify** it

Example (contd.)

Rewrite



$[p \text{ :-> ite } (x > 1, \&a, \&b)]$

$\text{simplify}(\text{ite } (C, E, E)) = \text{simplify}(E)$
 $\text{simplify}(\text{ite } (C, E1, E2), P) = E1 \text{ if } P \Rightarrow C$
 $\text{simplify}(\text{ite } (C, E1, E2), P) = E2 \text{ if } P \Rightarrow !C$

$[y \text{ :-> ite } (x > 1, * \text{ite } (x > 1, \&a, \&b)) + 3,$
 $\quad * \text{ite}((x > 1, \&a, \&b)) + 1$
 $]$

$[y \text{ :-> ite } (x > 1,$

$[(x == 5, y \text{ :-> ite } (x > 1, * \text{ite } (x > 1, \&a, \&b)) + 3,$
 $\quad * \text{ite}((x > 1, \&a, \&b)) + 1$

$[y \text{ :-> } \quad)$
 $]$

Example with Arrays (due to M. Musuvathi)

```
a[0] = 0;  
if (pi) a[i] = 0 else a[i] = 1; [ i = 1, 2, 3, 4 ... ]  
assert(a[0] == 0);
```

- $a_i \rightarrow \mathbf{ite}(p_i, \mathbf{store}(a_{i-1}, i, 1), \mathbf{store}(a_{i-1}, i, 0))$
- To prove $a[0] = 0$ ($\mathbf{select}(a_n, 0) = 0$), the SMT solver must consider the exponential search space for all p_i
- Now, consider a rewrite rule provided by user
 - $\mathbf{ite}(p, \mathbf{store}(a, j, x), \mathbf{store}(a, j, y)) = \mathbf{store}(a, j, \mathbf{ite}(p, x, y))$

Efficient Simplification

- How do we **simplify effectively**?
 - Large set of rules for simplification (**ite** + theory rules)
 - May need to **add/remove rules** particular to a problem
 - Rules may interact in unexpected ways
- The theory of **Term Rewriting** offers a systematic way to simplify

Term Rewriting

- Terms $T(S, X)$ over **signature** S and set of **variables** X
- Rewrite system is a set of
 - **equations** : $r = s$
 - **conditional equations**: $(u_1 = v_1 \wedge \dots \wedge u_n = v_n) \Rightarrow r = s$
 - **rewrite rules** : $l \rightarrow r$
 - **conditional rewrite rules**
- Rewriting a term “ t ” with a Rewrite system
 - **Orient** all equations into $l \rightarrow r$
 - **Match** : find subterm t' of t , so that $t = l\sigma$
 - **Substitute**: $t [t'/r\sigma]$
 - **Goal**: Obtain a normal form for t , where can't apply any rewrite rule
- Desirable properties: **Termination, Confluence**
- Given an Equational theory, a **Decision Procedure** can be obtained by constructing a terminating and confluent rewrite system

Rewriting Logic

- **Logic for Rewrite systems**
 - List of **operators** (uninterpreted symbols), **sorts** (types) with ordering
 - (Conditional) Equations and Rewrite rules on the symbols
 - Generic specification language for Rewrite systems
- **Examples**
 - $\text{select}(\text{store}(a, j, x), i) = \text{if } i = j \text{ then } x \text{ else } \text{select}(a, i)$
 - $\text{ite}(P, E, E) = E$
- Rewrite engines **compile** the set of rules into automata for **efficient matching**
 - Maude, ELAN, CafeOBJ, ASF+SDF engine, Stratego/XT, Tom
 - Flexibility to add/remove rules
- **Maude tool** allows modular specification of Rewrite systems in Rewriting Logic

SPA with Rewriting

- Rewrite at join points (**join_data**)
 - Invoke Maude engine on-the-fly; rules specified in text
- Rewrite rules
 - For **choose**, **ite**, Presburger arithmetic
 - Rewriting logic allows **seamless combination** of rules for multiple **theories**
 - We show that the Rewrite system is terminating
- Rewrite rules for **ite**
 - Equational axiomatization: McCarthy, Bloom-Tindell, Nelson-Oppen
 - Semantic: Sethi (implied, useless tests), Burch-Dill
 - **ite**+EUF: Ackermann, Bryant et al., Pnueli et al. , Groote-Pol

A Sample of Rewrite Rules

```
(I1) ite (true, E, E') = E
(I2) ite (false, E, E') = E'
(I3) ite (not N, E, E') = ite (N, E', E)
(I4) ite (N, E, E) = E
(I5) ite (N, ite(N, E, E'), F) = ite (N, E, F)
(I6) ite (N, F, ite(N, E, E')) = ite (N, F, E')
(I7) ite (N, ite(N1, E, E'), E)
    = ite (N and not N1, E', E)
(I8) ite (N, ite(N1, E, E'), E')
    = ite (N and N1, E, E')
(I9) ite (N, E, ite(N1, E, E'))
    = ite (not N and not N1, E', E)
(I10) ite (N, E', ite(N1, E, E'))
    = ite (not N and N1, E, E')

(I11) ite (N, E, E') ⊙ F = ite (N, E ⊙ F, E' ⊙ F)
(I12) - ite (N, E, E') = ite (N, -E, -E')

(S1) simplEnv (V, V EQ N and C) = N
(S2) simplEnv (ite(C and P, Et, Ee), C and Q)
    = simplEnv(ite (P, Et, Ee), C and Q)
(S3) simplEnv (ite(C, Et, Ee), P)
    = simplEnv(Ee, P) if P implies (not C)
(S4) simplEnv (ite(C, Et, Ee), P)
    = simplEnv(Et, P) if P implies C
(S5) simplEnv (ite(C, X, Y), P) = Y
    if (P and C) implies (X EQ Y)
(S6) simplEnv (ite(C, X, Y), P) = X
    if (P and not C) implies (X EQ Y)
(S7) simplEnv (ite(C, Et, Ee), C and P)
    = simplEnv(Et, C and P)
(S8) simplEnv (ite(C, Et, Ee), not C and P)
    = simplEnv(Ee, not C and P)
(S9) simplEnv (ite(C, Et, Ee), P)
    = ite (C, simplEnv(Et, P and C),
        simplEnv(Ee, P and not C) ) [owise]
(S10) simplEnv (E, P) = E [owise]
```

```
*** Presburger arithmetic rules
(A1) (E NEQ E') = not (E EQ E')
(A2) ((E EQ P) and (E EQ Q)) = false if P NEQ Q .
(A3) (not(E EQ P) and (E EQ Q)) = E EQ Q if P NEQ Q .
(A4) ((E + P) EQ Q) = (E EQ (Q - P))
(A5) (-E EQ P) = (E EQ -P) if E is not an integer
*** normalize to <=
(A6) (E > E') = (E' <= E - 1)
(A7) (E < E') = (E <= E' - 1)
(A8) (E >= E') = (E' <= E)
(A9) (not (E <= E')) = (E' <= E - 1)
*** orient constants to right and other terms to left
(A10) (E + P <= E') = (E <= E' - P)
(A11) (P <= E') = (0 <= E' - P)
(A12) (E <= F) = (E - F <= 0) if F is not an integer
(A13) (E <= F + G) = (E - F <= G) if F is not an integer

*** constant propagation
(A14) ((E <= F) and (E EQ P)) = (P <= F) and (E EQ P)
(A15) (E' + E <= F) and (E EQ P)
    = (E' + P <= F) and (E EQ P)
(A16) (-E <= F) and (E EQ P)
    = (-P <= F) and (E EQ P)
(A17) ((E <= P) and (E <= Q)) = E <= P if P < Q
(A18) ((E <= P) and (E <= Q)) = E <= Q [owise]

*** difference logic rules
(A19) ((E + (- F) <= P and F + (- E) <= Q))
    = false if P + Q < 0
(A20) ((E <= P and - E <= Q))
    = false if P + Q < 0
```

Simplification for Loops

- **Loop heads/exits** allow another opportunity for simplification
 - Can capture/exploit loop structure in **more compact way** than nested **ite** terms
- Loops give rise to similar **parametric term values**
- `for(i=0; i<n; i++)`
 - $i \mapsto \mathbf{choose} ((\text{true}, 0), (0 < n, 1), (0 < n \wedge 1 < n, 2), \dots)$
 - Generalize as: $\mathbf{choose} ((0 \leq k-1 < n, k-1)), k = 1, 2, \dots$
 - Term Generalization via **Anti-unification**

(Parametric) Anti-Unification of Terms

- **Anti-unifier (AU)** of t_1 and t_2 preserves common structure and introduces new variables for the difference
 - $AU (f(a, g(b, c)) , f (b, g(x,c))) = f (z, g(x, c))$
with $\sigma_1 = \{z \rightarrow a, x \rightarrow b\}$ and $\sigma_2 = \{z \rightarrow b\}$
- **Parametric Anti-unification (P-AU)**
 - Given $\langle t_1, \dots, t_n \rangle$, find $\mathbf{t}(\mathbf{k})$, so that, $t_i = \mathbf{t}(\mathbf{k}) [k \rightarrow i]$, $1 \leq i \leq n$
 - e.g., $P\text{-AU}(\langle 0, 1, 2, \dots, 10 \rangle) = (0 \leq j \leq 10, j)$
- Intuitively, **summarizing N iterations** of the loop by a **closed form** solution
 - Method: **Generalize** from individual solutions

A Simple example

- Bounded-Parametric term (bp-term): $\int_{i=lo}^{hi} \mathbf{t(i)}$
 - \int is an associative operator, **lo** and **hi** are bounds on **i**
 - A list is represented as $\langle l_1, l_2, l_3, \dots \rangle$

Parametric-AU ($\langle (a \neq 1), (a \neq 1 \wedge a \neq 2), (a \neq 1 \wedge a \neq 2 \wedge a \neq 3) \rangle$)

MATCH : $\bigwedge^* (v)$, where

$$\sigma_1(v) = \langle a \neq 1 \rangle$$

$$\sigma_2(v) = \langle a \neq 1, a \neq 2 \rangle$$

$$\sigma_3(v) = \langle a \neq 1, a \neq 2, a \neq 3 \rangle$$

PARAMETERIZE ($\langle \sigma_1, \sigma_2, \sigma_3 \rangle$) :

$$\sigma_1(v) = \int_{i=1}^1 (a \neq i)$$

$$\sigma_2(v) = \int_{i=1}^2 (a \neq i)$$

$$\sigma_3(v) = \int_{i=1}^3 (a \neq i)$$

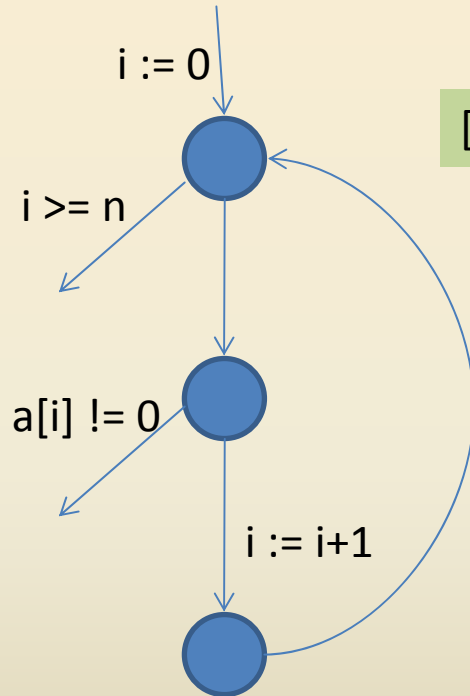
[e.g., P-AU (1, 2, 3) \rightarrow $\int_{i=1}^3 (i)$]

$$\mathbf{P-AU} (\langle \sigma_1(v), \sigma_2(v), \sigma_3(v) \rangle) : \sigma = \int_{i=1}^k (a \neq i)$$

SUBSTITUTE : $[\bigwedge^*(v)] \sigma = (\bigwedge_{i=1}^k (a \neq i))$

RETURN $\int_{k=1}^3 (\bigwedge_{i=1}^k (a \neq i)) \rightarrow (1 \leq k \leq 3 \wedge 1 \leq i \leq k \wedge a \neq i)$

Loop/Array P-AU example



$[i \text{ :-> choose } [(\text{true}, 0), (c_1, 1), (c_2, 2), (c_3, 3) \dots]]$

where $c_k = (a[0] \neq 0 \wedge \dots \wedge a[k-1] \neq 0) \wedge (0 < n \wedge \dots \wedge k-1 < n)$
 $= \bigwedge_{j=0}^{k-1} (a[j] \neq 0) \wedge j < n$

so, on generalization:

$[i \text{ :-> choose } ((c_k \wedge n_k, k))]$

where $n_k = (k \geq 0 \wedge k \leq \text{loop unrolls})$

Approximation for loops

- SPA **may not** terminate, e.g., for reactive loops
- Must introduce **approximation** to guarantee termination
- **Under-approximation**
 - Iterate each loop until cannot enter or k-loop exiting values found
 - Not unrolling the loops before-hand
 - Only Errors detected soundly
- **Over-approximation**
 - Iterate once; Set all variables written in loop to NON_DET
 - Better: **combine with invariants** from other domains, e.g., octagons
 - Only Proofs (unreachability of error locations) are sound

Alternative Methods

- **Bounded Model Checking** using Program Transition Relation
 - Formula size blows up with depth, Loops unrolled up to fixed depth
 - Difficult to exploit program structure
- **Verification Condition (VC) Generation**
 - Needs program annotations, e.g., loop invariants
 - SPA may be viewed as an operational style of VC Generation
- **Abstract Interpretation** based Analysis
 - Join in common abstract domains leads to loss of precision
 - Eager abstraction for scalability, termination

Related Work

- **Merge-based symbolic analysis algorithms** for Hardware
 - Symbolic Execution for Hardware: Koebl, Pixley'05
 - Lightweight simplification of terms; Adhoc treatment of loops
 - SAT query caching (Arons et al'08): complementary optimization
- **SSA based WP computation** (Flanagan, Saxe'01, Leino'05)
 - **Eager flattening to clauses** does not allow simplification of terms
 - Becomes a burden on the decision procedure
- Similar to **Calysto** (Babic, Hu'07, '08)
 - Data-flow algorithm, Term Rewriting, Generalization are our contributions
 - Need to incorporate function summaries

Implementation

- Implemented in **F-Soft framework** for C programs
 - **Yices** SMT Solver
 - **Maude** Rewrite Engine (rules in text ~300 lines)
- Compared with **DFS-based symbolic execution**
 - Optimized based on **Incremental SMT solving** (+IncSMT)
 - Fixed **Depth and Loop bounds** (minimum to find errors)
- Compared with and without Rewriting (+/- Rew)

Experimental Results

Benchmark	DFS-based Symbolic Execution						SPA			
	-Rew, -IncSMT		+Rew, -IncSMT		+IncSMT, -Rew		-Rew		+Rew	
	T(s)	DP T(s)	T(s)	DP/Rew T(s)	T(s)	DP T(s)	T(s)	DP T(s)	T(s)	DP/Rew T(s)
tcas(p1)	TO	>1782	200	131/40	3	2	TO	> 1720	88	20/67
tcas(p2)	TO	>1782	200	131/40	3	2	TO	>1720	88	20/67
tcas(p3)	104	103	3	2.4/0.5	0	0	TO	>1720	27	7/19
tcas(p4)	112	112	3	2.4/0.5	0	0	TO	>1720	34	10/24
tcas(p5)	TO	>1782	200	131/40	3	2	TO	>1720	88	20/67
M(p1-30)	1433	1389	313	214/65	9	5	TO	>1761	15	8/6

TO = 1800s, MO = 2GB

New Experiments (work with G. Li)

- Native interface to Maude
- Improved specification of conditional rewrite rules
- Improved Term Library

Benchmark	loc	DFS-based Symbolic Execution		SPA	
		T(s)	+IncSMT DP T(s)/Depth	T(s)	+Rew DP/Rew T(s)
tcas(p5)	550	3	2/110	6.39	0.14/6.08
M(p1-30)	3500	9	5/30	0.93	0.14/0.32
H (p1-17)	6775	55	20/90	5	0.16/3.59

Observations

- **Average time** for each call to Rewrite engine is **similar** to that for SMT solver
- While **Incremental SMT** is a powerful technology, it alone **cannot deal with path explosion**
- Conditional Rewrite rules are most expensive
- **ite** terms of predicate sort are difficult to simplify with rewriting

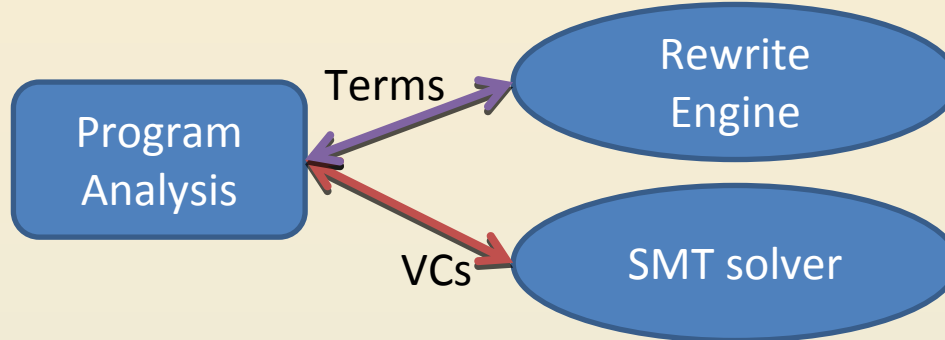
Rewriting for Program Analysis

- Generic **front-end for simplification** of terms systematically **before clausifying** in SMT solvers
- **Decouple** term-based simplification from clause-based simplifications
- **Avoids re-doing** simplification for each call to SMT solver
- **Tight integration** of Rewriting engines with Program analyses is **feasible** and improves performance

Program Analysis Frameworks

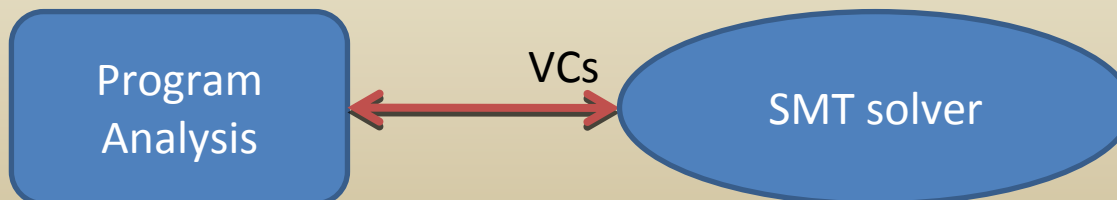
- SPA

- Task of proving VC shared by Rewriting engine and SMT solver; rewriting exploits structure before classification



- Most Program Analyses : Generate VC -> SMT solver

- Whole burden of proving classified VC on SMT solver: loss of structural information



Summary

- SPA **exploits program structure** to give rise to structured terms while generating error conditions (ECs)
 - Avoids path explosion by doing data-flow analysis on terms
 - Join operator over terms
- Term rewriting and generalization **exploits term structure** to obtain simplified ECs
- Simpler ECs are discharged by a Decision Procedure (DP); **flattening to clauses is done lazily**
- Term rewriting as a **generic front-end** to a DP
 - However, rewriting can be invoked independently also.

Extensions/Future Work

- SPA described in context of integer programs
 - Rewriting will be widely useful for **simplification in theories** of bitvectors, arrays and heaps
- Efficient **SMT query caching** for Terms
- Incorporate **Function Summaries** to scale up
- **Saturation-based** rewriting instead of Redundancy Removal
- Anti-unification for **computing loop invariants**
- **Lazy strategies** to apply rewrite rules
 - Avoid eager/wasteful simplification attempts

Questions?