Syntax-driven Program Verification of Matching Logic Properties

Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, Dino Mandrioli, **Alessandro M. Rizzi**

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

Long term goal

Develop a
general
approach for
incremental verification

Goal

- show that our general syntax-based framework (SiDECAR) leads to efficient verification
  - comparable with the state of the art for particular applications
- does not incur in performance penalties even in the non incremental case

In this work...

- application of our general syntax-driven framework (SiDECAR) to program verification based on matching logic
- considering the particular case of C-like programs (including recursion, loops, and rich heap specifications)
  - kernelC
  - no penalization of our approach w.r.t. traditional implementation

Outline

- brief overview of matching logic
- syntax-based approach of matching logic verification
- preliminary evaluation

Matching logic
software verification

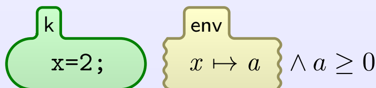
Matching logic

- Hoare-like, language independent, semantic abstraction
- developed by G. Roșu's team
- describes sets of program states through special many-sorted first-order logic with equality formulae called *configuration patterns*
- special term for representing a program configuration

Configuration pattern

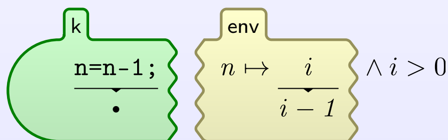
- represents a set of configurations
- is a matching logic formula
- configuration with logic variables

$$\exists a, \rho((\underbrace{\square}_{\text{current configuration}} = \underbrace{\langle \langle \texttt{x=2;} \rangle_k \, \langle \texttt{x} \mapsto a, \rho \rangle_{env} \rangle_{cfg}}_{\text{configuration term}}) \wedge a \geq 0)$$



Reachability rules

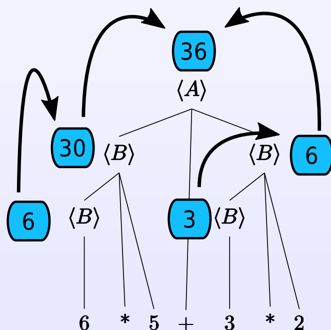
- state trasitions are represented through *reachability rules*
- each *reachability rule* is composed of two patterns
- matching logic defines a set of inference rules which allows program reachability checking



Syntax-based representation of matching logic verification

Synthesized-only attribute grammars

- attribute grammars (AG) are a formalism for attach meaning to syntax trees
- in synthesized-only AGs, attributes of a given node only depends on its children



Attribute description

- reachability rules are derived from a general template inside the syntactical structure of the program
  - instantiation of particular rules based on the current code of the program
  - check performed through explicit generation of program configurations
- we need to maintain this information along the syntax-tree:
  - the code C (sequence of tokens plus symbolic integers)
  - the available reachability rules R (set of reachability rules)
  - the state of every verification task Vt (a set of verification tasks)

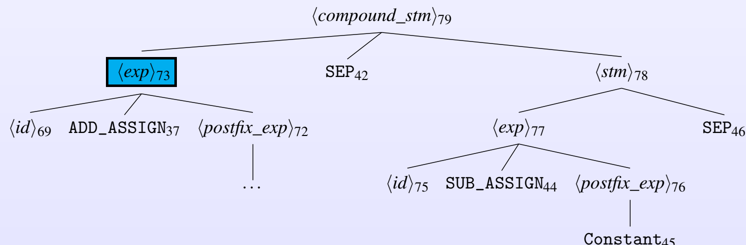
Running example

```
1  int neg(int n){
2      return -n;
3  }
4
5  int sum_iterative(int n)
6  //@pre: n>=0
7  //@post: return = -n*(n+1)/2
8  {
9      int s;
10     s = 0;
11     //@inv s = -(old(n)-n) * (old(n)+n+1) / 2 /\ n>=0
12     while (n > 0) {
13         s += neg(n);
14         n -= 1;
15     }
16     return s;
17 }
```

Rule generation

- we want generate rules of $\langle exp \rangle_{73}$

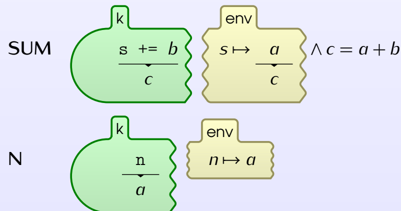
```
1  int neg(int n){
2    return −n;
3  }
4
5  int sum_iterative(int n)
6  //@pre: n>=0
7  //@post: return = −n∗(n+1)/2
8  {
9    int s;
10   s = 0;
11   //@inv s = −(old(n)−n) ∗ (old(n)+n+1) / 2 /\ n>=0
12   while (n > 0) {
13     s += neg(n);
14     n −= 1;
15   }
16   return s;
17 }
```



Rule generation

- in each node attribute R maintains the set of available rules
- rules instantiated on the actual code from general template
- since the code is needed for producing each rule, another attribute maintains the sequence of code tokens
- $R_{73} = \{SUM, N\}$
- $C_{73} = \{\texttt{s += neg(n);}\}$

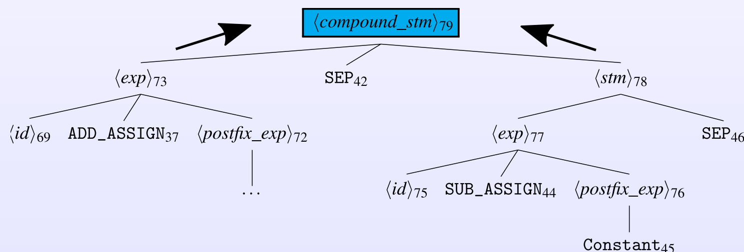
13 s += neg(n);



Rule propagation

- we have to propagate the rules on $\langle compound_stm \rangle_{79}$
- $R_{79} = R_{73} \cup R_{78} = \{SUM, N\} \cup R_{78}$
- we have also to compute the code of the node
- $C_{79} = C_{73} C_{78} = \{$`s += neg(n); n -= 1;`$\}$

```
13 s += neg(n);
14 n -= 1;
```



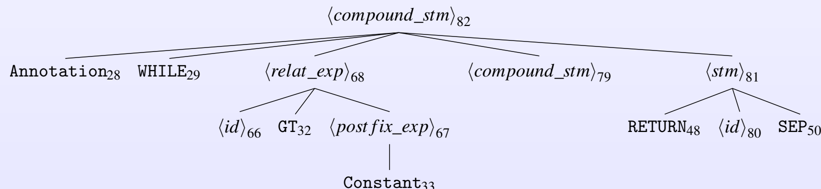
Software verification

- split into different units, e.g., annotated loops or functions
- each unit forms a *verification task*
- if every verification task is successfully checked the program follows its specifications
- each verification task can succeed, fail or be unknown

Example

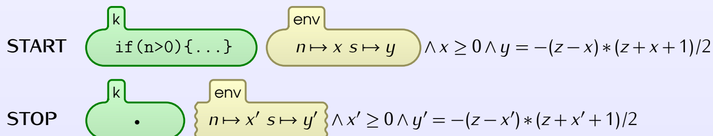
- an annotated loop defines a new verification task *Vt*
- such *Vt* checks the correctness of the loop

```
10 //@inv s = −(old(n)−n) ∗ (old(n)+n+1) / 2 /\ n>=0
11 while (n > 0) {
12       s += neg(n);
13       n −= 1;
14 }
15 return s;
```



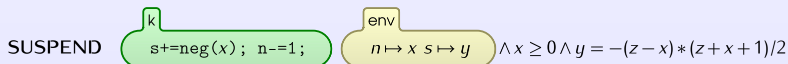
Checking a verification task

- each verification task define one (or more) starting configuration pattern
- it defines also the final pattern which must be reached
- it is checked by applying all possible rule to the starting pattern



Syntax-based verification task evaluation

- the evaluation can be performed during the semantic evaluation
- not all the relevant rules may be available at a certain point
- in such case the evaluation is suspended to be resumed afterwards
- e.g., function call `neg` is not available inside the loop

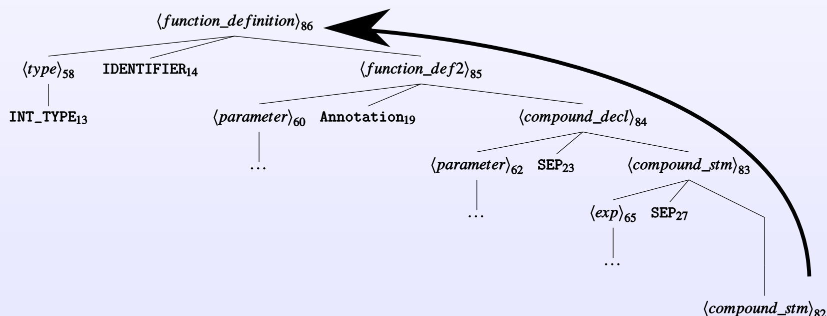


Verification task state must be saved

- a triplet $v_t = \langle C_r, R_v, C_t \rangle$ maintains the state of a verification task
- C_r is the set containing the *frontier* of reached configuration patterns
- R_v are the rule available for this task
- C_t is the set of configuration patterns representing the postcondition
- e.g., $Vt_{82} = \langle SUSPEND, R_{82}, STOP \rangle$
- in every node, attribute V contains the set of the current verification tasks

Verification task propagation

- each uncompleted verification task is propagated towards the root of the tree
- if new rule are available, they are use to further process it
- e.g., $V_{83} = V_{65} \cup V_{82}$

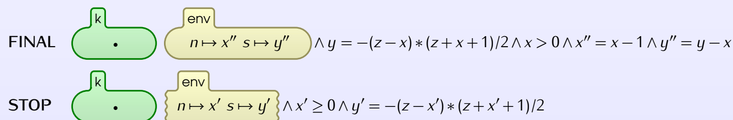


Function definition

- a verification task is initialized also for annotated functions
- reachability rules which contains the behavior of the function from the specifications are provided

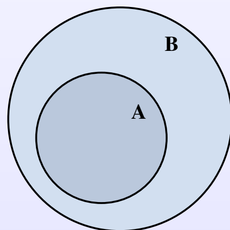
Resuming a verification task

- when the missing rules are available the verification task can continue
- the `while` verification can proceed until no further steps can follow (e.g., *FINAL*)
- what next?



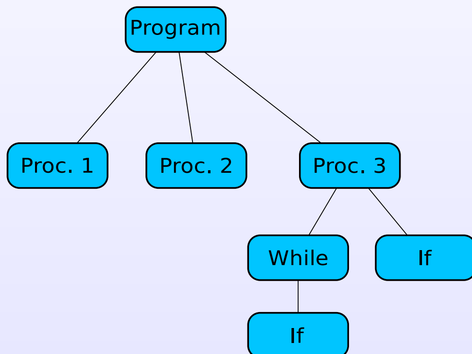
Reachability checking

- imagine to have reached the final pattern φ which defines the set of states A
- postcondition pattern φ' defines the set of states B
- we want to check $A \subseteq B$
- SMT solver for checking formula implications



Intrinsically compositional approach

- decoupling different portions of the code
  - incrementality
  - parallel evaluation
- however some non-local information (e.g. function call) may not be available at a certain point



Preliminary *evaluation*

Comparison on MatchC benchmarks

Program	MatchC	*SiDECAR*
DivisionByZero	557	23
UninitVariable	548	9
UnalLocation	504	18
UninitMemory	540	79
Average	439	18
Minimum	439	65
Maximum	445	81
MultiByAddition	519	58
SumRecursive	468	81
SumIterative	518	61
CommAssoc	432	43
Head	443	64
Tail	452	36
Add	488	91
Swap	481	75

Program	MatchC	*SiDECAR*
Deallocate	492	51
LengthRecursive	508	54
LengthIterative	504	92
SumRecursive	471	91
SumIterative	521	53
Reverse	513	56
Append	547	217
Copy	597	394
Filter	687	566
Insert	750	730
InsertionSort	802	764
BubbleSort	757	898
QuickSort	2,442	524
MergeSort	2,004	1,667

Results in milliseconds

. . . with few exceptions

Program	MatchC	*SiDECAR*
DivisionByZero	557	23
UninitVariable	548	9
UnalLocation	504	18
UninitMemory	540	79
Average	439	18
Minimum	439	65
Maximum	445	81
MultiByAddition	519	58
SumRecursive	468	81
SumIterative	518	61
CommAssoc	432	43
Head	443	64
Tail	452	36
Add	488	91
Swap	481	75

Program	MatchC	*SiDECAR*
Deallocate	492	51
LengthRecursive	508	54
LengthIterative	504	92
SumRecursive	471	91
SumIterative	521	53
Reverse	513	56
Append	547	217
Copy	597	394
Filter	687	566
Insert	750	730
InsertionSort	802	764
BubbleSort	757	898
QuickSort	2,442	524
MergeSort	2,004	1,667

Results in milliseconds

Comparison on recursion: Fibonacci

N	MatchC	*SiDECAR*	N	MatchC	*SiDECAR*
1	461	73	10	3,237	1,174
2	476	75	11	4,325	1,665
3	506	126	12	9,690	2,344
4	535	167	13	13,127	3,141
5	575	249	14	31,641	4,412
6	707	346	15	42,621	6,003
7	880	499	16	107,802	9,351
8	1,327	711	17	146,594	13,855
9	1,678	852	18	OutOfMemory	OutOfMemory

Results in milliseconds

Comparison on list unrolling through loop

Length	MatchC	*SiDECAR*
2	487	102
4	530	138
8	$1,323$	295
16	OutOfMemory	675
32	OutOfMemory	$2,960$
64	OutOfMemory	$23,017$
128	OutOfMemory	$295,477$

Results in milliseconds

Comparison on list sorting

Length	MatchC	*SiDECAR*
1	499	113
2	512	250
3	694	751
4	2,030	3,944
5	34,200	27,310
6	1,024,254	220,875

Results in milliseconds

Conclusions

- our general syntax-based framework (SiDECAR) can be applied to the particular case of matching logic software verification
- for the particular case of software verification with respect of matching logic formalism
  - for KernelC programming language
  - no performance issues w.r.t. current solutions

Future work

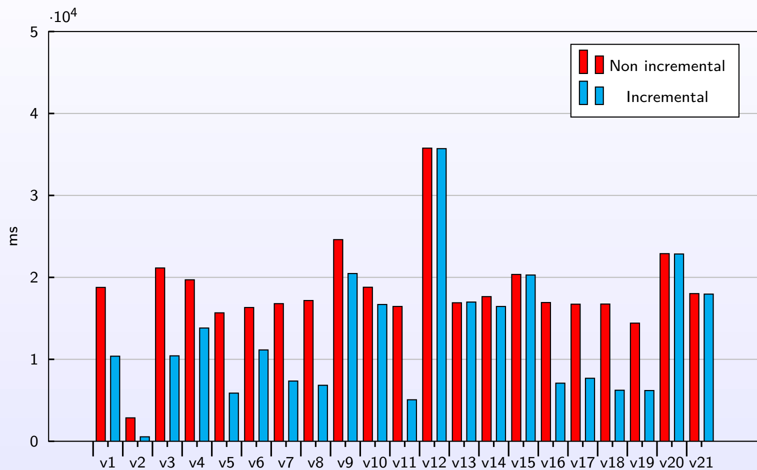
- develop a generalized framework for incremental software verification
  - extend our framework (SiDECAR) to other incremental applications
  - extend to other kinds of verification

Thank you for your attention!

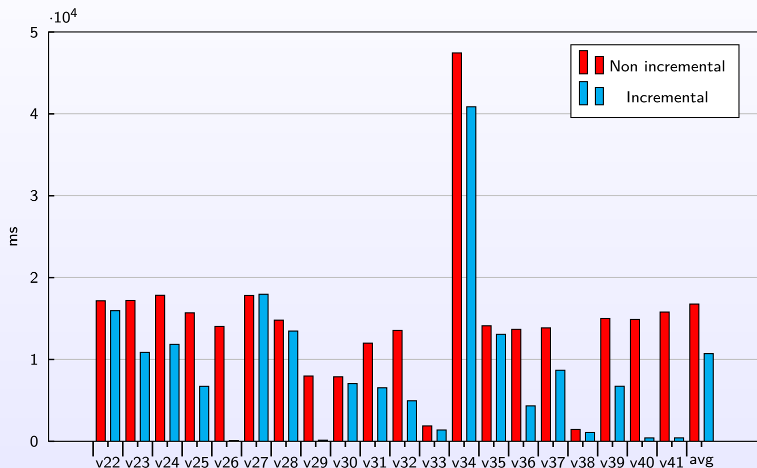
Incrementality

- our long term objective is **incremental verification**
- verification technique which efficiently handle re-verification of a new software version
- in this first step we provide an encoding of software verification through matching logic using a syntax-driven approach
- we aim at apply our incremental techniques on top of it

Incremental results over Siemens TCAS bechmark



Incremental results over Siemens TCAS bechmark



Verification task evaluation algorithm

```
1:  function EVAL(v_t = ⟨C_R, R_v, C_t⟩)
2:      repeat
3:          for c_i ∈ C_r do
4:              Changed ← false
5:              temp ← ∅
6:              for r_i ∈ R_v do
7:                  if Matches(c_i, r_i) then
8:                      c′ ← ApplyRule(c_i, r_i)
9:                      if isSat(c′) then
10:                         temp ← temp ∪ c′
11:                     end if
12:                 end if
13:             end for
14:             if temp ≠ ∅ then
15:                 Changed ← true
16:                 C_r ← C_r ∪ temp ∖ {c_i}
17:             end if
18:         end for
19:     until ¬Changed
20:     for c_i ∈ C_r do
21:         if ¬IsFinal(c_i) then
22:             return false
23:         end if
24:     end for
25:     for c_i ∈ C_r do
26:         Satisfied ← false
27:         for c_t ∈ C_t do
28:             if satisfy(c_i, c_t) then
29:                 Satisfied ← true
30:             end if
31:         end for
32:         if ¬Satisfied then
33:             return false
34:         end if
35:     end for
36:     return true
37: end function
```

Overall attribute evaluation algorithm

```
1:  function CompAttribute(a_1, ... a_n)
2:      K ← a_1.K ... a_n.K
3:      R_temp ← gen(K)
4:      R ← ⋃_{i=1}^{n} a_i.R ∪ R_temp
5:      V_temp ← ⋃_{i=1}^{n} a_i.V
6:      if hasContract() then
7:          V_temp ← V_temp ∪ genVT(R)
8:      end if
9:      V ← ∅
10:     for v_i ∈ V_temp do
11:         v'_i ← v_i
12:         v'_i.R_v ← v'_i.R_v ∪ R_temp
13:         eval(v'_i)
14:         V ← V ∪ v'_i
15:     end for
16:     a_0 ← (K, R, V)
17:     if isRootNode() then
18:         for v_i ∈ V_temp do
19:             if eval((v_i) ≠ true then
20:                 return (a_0, false)
21:             end if
22:         end for
23:         return (a_0, true)
24:     else
25:         return (a_0, delay)
26:     end if
27: end function
```

Back to the example

- from *FINAL* we have:

$$y = -(z-x)*(z+x+1)/2 \wedge x > 0 \wedge x'' = x-1 \wedge y'' = y-x \tag{1}$$

- from *STOP* we have:

$$x' \geq 0 \wedge y' = -(z-x')*(z+x'+1)/2 \tag{2}$$

- the additional constraints are the matching of `n` and `s`:

$$x' = x'' \wedge y' = y'' \tag{3}$$

- $\psi = (1) \wedge (3)$ and $\psi' = (2)$