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 nonincremental 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 \textit{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

$$\exists a, \rho \left( \square \left( \langle x=2; \rangle \langle x \mapsto a, \rho \rangle \right) \wedge a \geq 0 \right)$$

k
x=2;

env
x \mapsto a \wedge a \geq 0
Reachability rules

- state transitions 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

\[
\begin{align*}
\text{n} &= \text{n} - 1; \\
\text{k} &\rightarrow \text{i} - 1 \\
\text{env} &\rightarrow \text{i} > 0
\end{align*}
\]
Syntax-based representation of matching logic verification
Synthesized-only attribute grammars

- attribute grammars (AG) are a formalism for attaching meaning to syntax trees
- in synthesized-only AGs, attributes of a given node only depend 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

```c
int neg(int n)
{
    return -n;
}

int sum_iterative(int n)
{
    //@pre: n>=0
    //@post: return = -n*(n+1)/2
    
    int s;
    s = 0;
    //@inv s = -(old(n)-n) * (old(n)+n+1) / 2 /
    while (n > 0) {
        s += neg(n);
        n -= 1;
    }

    return s;
}
```
Rule generation

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

```c
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

$R_{73} = \{SUM, N\}$

$C_{73} = \{s \;+\; \text{neg}(n); \}$

13 $s \;+\; \text{neg}(n);$
Rule propagation

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

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 $V_t$
- such $V_t$ checks the correctness of the loop

```plaintext
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;
```

```
Annotation\_28 ~ WHILE\_29 ~ <relat\_exp>\_68 ~ <compound\_stm>\_79 ~ <stm>\_81
  \<id>\_66 ~ GT\_32 ~ <postfix\_exp>\_67
    Constant\_33
```

RETURN\_48 ~ \<id>\_80 ~ SEP\_50
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

\[
\begin{align*}
\text{START} & \quad \text{if}(n > 0)\{...\} \quad n \mapsto x \quad s \mapsto y \\
\quad & \quad \land x \geq 0 \land y = -(z - x) \times (z + x + 1)/2 \\
\text{STOP} & \quad . \quad n \mapsto x' \quad s \mapsto y' \\
\quad & \quad \land x' \geq 0 \land y' = -(z - x') \times (z + x' + 1)/2
\end{align*}
\]
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

\[
\begin{align*}
    & s+\text{neg}(x);\ n-=1; \quad \text{SUSPEND} \\
    & n \mapsto x, s \mapsto y \quad \text{env} \\
    & \land x \geq 0 \land y = -(z-x) \times (z+x+1)/2
\end{align*}
\]
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?

\[
\begin{align*}
\text{FINAL } & \quad k \mapsto x'' \quad s \mapsto y'' \\
& \quad \land y = -(z - x)(z + x + 1)/2 \land x > 0 \land x'' = x - 1 \land y'' = y - x
\\
\text{STOP } & \quad k \mapsto x' \quad s \mapsto y' \\
& \quad \land x' \geq 0 \land y' = -(z - x')(z + x' + 1)/2
\end{align*}
\]
Reachability checking

- imagine to have reached the final pattern $\phi$ which defines the set of states $A$
- postcondition pattern $\phi'$ 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

<table>
<thead>
<tr>
<th>Program</th>
<th>MatchC</th>
<th>SiDECAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>DivisionByZero</td>
<td>557</td>
<td>23</td>
</tr>
<tr>
<td>UninitVariable</td>
<td>548</td>
<td>9</td>
</tr>
<tr>
<td>UnallocLocation</td>
<td>504</td>
<td>18</td>
</tr>
<tr>
<td>UninitMemory</td>
<td>540</td>
<td>79</td>
</tr>
<tr>
<td>Average</td>
<td>439</td>
<td>18</td>
</tr>
<tr>
<td>Minimum</td>
<td>439</td>
<td>65</td>
</tr>
<tr>
<td>Maximum</td>
<td>445</td>
<td>81</td>
</tr>
<tr>
<td>MultiByAddition</td>
<td>519</td>
<td>58</td>
</tr>
<tr>
<td>SumRecursive</td>
<td>468</td>
<td>81</td>
</tr>
<tr>
<td>SumIterative</td>
<td>518</td>
<td>61</td>
</tr>
<tr>
<td>CommAssoc</td>
<td>432</td>
<td>43</td>
</tr>
<tr>
<td>Head</td>
<td>443</td>
<td>64</td>
</tr>
<tr>
<td>Tail</td>
<td>452</td>
<td>36</td>
</tr>
<tr>
<td>Add</td>
<td>488</td>
<td>91</td>
</tr>
<tr>
<td>Swap</td>
<td>481</td>
<td>75</td>
</tr>
</tbody>
</table>

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

Results in milliseconds
Results in milliseconds
Comparison on recursion: Fibonacci

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

Results in milliseconds
Comparison on list unrolling through loop

<table>
<thead>
<tr>
<th>Length</th>
<th>MATCHC</th>
<th>SiDECAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>487</td>
<td>102</td>
</tr>
<tr>
<td>4</td>
<td>530</td>
<td>138</td>
</tr>
<tr>
<td>8</td>
<td>1,323</td>
<td>295</td>
</tr>
<tr>
<td>16</td>
<td>OutOfMemory</td>
<td>675</td>
</tr>
<tr>
<td>32</td>
<td>OutOfMemory</td>
<td>2,960</td>
</tr>
<tr>
<td>64</td>
<td>OutOfMemory</td>
<td>23,017</td>
</tr>
<tr>
<td>128</td>
<td>OutOfMemory</td>
<td>295,477</td>
</tr>
</tbody>
</table>

Results in milliseconds
Comparison on list sorting

<table>
<thead>
<tr>
<th>Length</th>
<th>MATCHC</th>
<th>SiDECAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>499</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>512</td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>694</td>
<td>751</td>
</tr>
<tr>
<td>4</td>
<td>2,030</td>
<td>3,944</td>
</tr>
<tr>
<td>5</td>
<td>34,200</td>
<td>27,310</td>
</tr>
<tr>
<td>6</td>
<td>1,024,254</td>
<td>220,875</td>
</tr>
</tbody>
</table>

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!
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 benchmark

The diagram compares the performance of non-incremental and incremental methods over various versions of Siemens TCAS benchmark. The y-axis represents time in milliseconds (ms), and the x-axis lists the versions from v1 to v21. The red bars represent non-incremental results, and the blue bars represent incremental results.
Incremental results over Siemens TCAS benchmark

Non incremental
Incremental
Veriﬁcation task evaluation algorithm

1: function \( \text{Eval}(v_t = \langle C_R, R_v, C_t \rangle) \)
2: \hspace{1em} repeat
3: \hspace{2em} for \( c_i \in C_r \) do
4: \hspace{3em} Changed \( \leftarrow \) false
5: \hspace{3em} temp \( \leftarrow \emptyset \)
6: \hspace{3em} for \( r_i \in R_v \) do
7: \hspace{4em} if Matches\( (c_i, r_i) \) then
8: \hspace{5em} \( c' \leftarrow \text{ApplyRule}(c_i, r_i) \)
9: \hspace{4em} if isSat\( (c') \) then
10: \hspace{5em} temp \( \leftarrow \) temp \( \cup \) c'
11: \hspace{4em} end if
12: \hspace{3em} end if
13: \hspace{2em} end for
14: \hspace{2em} if temp \( \neq \emptyset \) then
15: \hspace{3em} Changed \( \leftarrow \) true
16: \hspace{3em} \( C_r \leftarrow C_r \cup \text{temp} \setminus \{c_i\} \)
17: \hspace{3em} end if
18: \hspace{2em} end for
19: \hspace{1em} until \( \neg \)Changed
20: \hspace{2em} for \( c_i \in C_r \) do
21: \hspace{3em} if \( \neg \)IsFinal\( (c_i) \) then
22: \hspace{4em} return false
23: \hspace{3em} end if
24: \hspace{2em} end for
25: \hspace{2em} for \( c_i \in C_r \) do
26: \hspace{3em} Satisfied \( \leftarrow \) false
27: \hspace{3em} for \( c_t \in C_t \) do
28: \hspace{4em} if satisfy\( (c_i, c_t) \) then
29: \hspace{5em} Satisfied \( \leftarrow \) true
30: \hspace{4em} end if
31: \hspace{3em} end for
32: \hspace{3em} if \( \neg \)Satisfied then
33: \hspace{4em} return false
34: \hspace{3em} end if
35: \hspace{2em} end for
36: \hspace{2em} return true
37: end function
Overall attribute evaluation algorithm

1: function COMP_ATTRIBUTE(a₁,...aₙ)
2:     K ← a₁.K...aₙ.K
3:     Rₜemp ← gen(K)
4:     R ← ∪ᵢ₌₁ⁿ aᵢ.R ∪ Rₜemp
5:     Vₜemp ← ∪ᵢ₌₁ⁿ aᵢ.V
6:     if hasContract() then
7:         Vₜemp ← Vₜemp ∪ genVT(R)
8:     end if
9:     V ← ∅
10:    for vᵢ ∈ Vₜemp do
11:        vᵢ' ← vᵢ
12:        vᵢ'.Rᵥ ← vᵢ'.Rᵥ ∪ Rₜemp
13:        eval(vᵢ')
14:    V ← V ∪ vᵢ'
15: end for
16:    a₀ ← (K, R, V)
17:    if isRootNode() then
18:        for vᵢ ∈ Vₜemp do
19:            if eval((vᵢ) ≠ true then
20:                return (a₀, false)
21:            end if
22:        end for
23:        return (a₀, true)
24:    else
25:        return (a₀, delay)
26:    end if
27: end function
from \textit{FINAL} we have:

\[ y = -(z - x) \ast (z + x + 1)/2 \land x > 0 \land x'' = x - 1 \land y'' = y - x \]  \tag{1}

from \textit{STOP} we have:

\[ x' \geq 0 \land y' = -(z - x') \ast (z + x' + 1)/2 \] \tag{2}

the additional constraints are the matching of \( n \) and \( s \):

\[ x' = x'' \land y' = y'' \] \tag{3}

\[ \psi = (1) \land (3) \text{ and } \psi' = (2) \]