Syntax-driven Program Verification of Matching Logic Properties

Domenico Bianculli*, Antonio Filieri†, Carlo Ghezzi‡, Dino Mandrioli‡, Alessandro Maria Rizzi‡
*University of Luxembourg, Luxembourg, domenico.bianculli@uni.lu
†University of Stuttgart, Germany, filieri@informatik.uni-stuttgart.de
‡Politecnico di Milano, Italy, {carlo.ghezzi,dino.mandrioli,alessandromaria.rizzi}@polimi.it

Abstract—**We describe a novel approach to program verification and its application to verification of C programs, where properties are expressed in matching logic. The general approach is syntax-directed: semantic rules, expressed according to Knuth's attribute grammars, specify how verification conditions can be computed. Evaluation is performed by interplaying attribute computation and propagation through the syntax tree with invocation of a solver of logic formulae. The benefit of a general syntax-driven approach is that it provides a reusable reference scheme for implementing verifiers for different languages. We show that the instantiation of a general approach to a specific language does not penalize the efficiency of the resulting verifier. This is done by comparing our C verifier for matching logic with an existing tool for the same programming language and logic. A further key advantage of the syntax-directed approach is that it can be the starting point for an incremental verifier—which is our long-term research target.**

I. Introduction

Program verification has made considerable progress in recent years, both in terms of the logic in which one can express properties to check and the (semi-)automatic tools available to check these properties. Logical frameworks, often based on Hoare's logic, have been proposed to deal with important language features, such as pointer data structures; see, for example, separation logic [1] and matching logic [2]. Push-button tools have also become available covering realistic languages and supporting program verification, also thanks to progress in the SAT and SMT solving methods and technologies; see, for example, BLAST [3], JavaPathFinder [4], ESC/Java [5], Boogie [6], Z3 [7].

We argue, however, that the engineering of verification tools still needs to make significant steps further before program verification can become a mainstream task in the software development process, as instead software testing is today. On one hand, we aim to make the engineering of verification tools more systematic, to keep the pace with the rapid evolution of programming languages and development tools. On the other hand, the verification step has to seamlessly adapt within a development process that is more and more iterative and agile, supporting continuous changes and evolution.

To overcome these issues, in this paper we explore a syntax-driven approach to verification aiming to provide a unifying framework for the construction of verification tools. Within this framework, verification procedures are expressed as semantic attributes that are computed and propagated through the syntax tree of a program. Precisely, we propose the use of attribute grammars [8], a well-know powerful computing formalism. Given the grammar of a language, attribute rules may be associated with syntax rules to express verification. As verification conditions are computed as part of attribute evaluation[1], they can be passed to a logic solver for evaluation.

The proposed syntax-driven approach is general, and can be applied to any syntactically-specified language and to any logical framework. For example, in our earlier work [11], we applied it to the reliability analysis of structured business workflows. In this paper we instantiate the general approach to the case of verifying programs written in a significant subset of C (called KernelC [12]) against properties specified in matching logic. The main contribution of the paper is a definition of a verification procedure of matching logic properties [13] via an attribute grammar and by interplaying with an SMT solver.

The potential disadvantage of instantiating a general approach like this is that it may lead to inefficient implementations, which cannot compete with ad-hoc developed verifiers. As a second contribution of the paper, we report on a preliminary experimental assessment in which we compare our approach to MatchC [12], the (only) state-of-the-art tool developed to support verification of KernelC programs annotated with matching logic properties. The preliminary results are promising since not only our tool achieves similar results, but it also often performs better than MatchC.

As already mentioned, this syntax-driven approach is also a step in the direction of achieving verifiers that can naturally support continuous program changes. A key driver supporting changes is *incrementality* [14]. This means that verification should be able to automatically isolate the parts of the program that need to be re-evaluated after any change and automatically restrict the analysis effort to the minimal program fragment affected by the change. Although modularity and encapsulation may help in this, we are looking for an approach that can work at any level of fragment granularity and where the minimum amount of re-analysis to be performed is automatically chosen by the tool. This is where a syntax-directed approach can help. Past work [15] has built a theory of incremental syntactic

[1]The approach pursued in this paper is typically bottom-up, i.e., based on purely synthesized attributed. This choice could be questioned and a mixed approach using both synthesized and inherited attributes could also be adopted. The main reason for our choice, however, is that bottom-up parsing better supports incrementality and possibly even parallelism [9], [10].

analysis, which can help detect the minimum fragment to be re-parsed to check syntactic correctness. Incremental parsing can be extended in a natural way to also cover incremental attribute evaluation, achieving a syntactic-semantic incremental approach to program verification [10]. Although our main long-term goal is to develop a generalized incremental verification approach, we remark that this paper focuses *only* on the first step of the roadmap that will eventually lead to it: a syntax-directed approach for verification of C programs with properties expressed in matching logic.

II. Matching Logic at a Glance

Matching logic provides a formal system for reasoning about structural properties of the *configurations* a program goes through during its execution [13]. Each configuration is defined by a bag of labeled elements called *cells*. Each cell represents a relevant aspect of a program configuration, such as the current memory allocation table, the residual code to be executed, the I/O output buffer, and the next instruction pointer; the structure (and the complexity) of the configurations is language-dependent.

An example configuration is $\langle\langle \texttt{x=2}\rangle_k \langle \texttt{x} \mapsto 5\rangle_{env}\rangle$, which is is specified through two cells *k*, and *env*. Cell *k* contains the residual code (as a list of statements) to be executed from the current state. Cell *env* represents the current variables definition as a map from variables to values. The meaning of a symbol depends on the specific cell it appears in. For example, the "2" in cell *k* is a syntactic element of the program text, while the "5" in cell *env* is interpreted as the constant 5.

Properties are specified in matching logic as first-order predicates (with equality) on the configurations. These predicates are called *configuration patterns* and look like:

$$\langle\langle \texttt{x=2}\rangle_k \langle \ldots \texttt{x} \mapsto a \ldots\rangle_{env}\rangle \wedge a \geq 0 \qquad (1)$$

where a is an existentially quantified integer variable, $\texttt{x}$ is a variable identifier, and dots represent unconstrained configuration elements. This pattern matches every configuration where the residual program to be executed is `x=2` and the variable x is assigned to a non-negative integer, regardless of other conditions such as the assignment of other variables.

Transitions between program configurations are defined through *reachability rules* $\psi \Rightarrow \psi'$ between patterns ψ and ψ'; informally, such a rule states that a program configuration matching pattern ψ takes zero or more steps to reach another configuration that matches pattern ψ'. We follow the notation of the $\mathbb{K}$ framework [16], where the left-hand-side of a rule is above a line and the right-hand-side is below a line; parts of the rule without a line are the same on both sides of the rule. For example, the rule $\langle\langle \frac{\texttt{x=2}}{2}\rangle_k \langle \ldots \texttt{x} \mapsto \frac{a}{2} \ldots\rangle_{env}\rangle \wedge a \geq 0$ allows to move from a configuration matching the pattern in Equation (1) to another configuration where the residual program `x=2` has been interpreted (leaving the value of the expression, the integer value 2, in cell *k*) and variable x is assigned the value 2, as represented in cell *env*.

To use matching logic for the verification of program properties, one has first to define, as a rewrite system, the semantics of the programming language in which the programs to be verified are written. Matching logic reachability is therefore implemented as rewriting rules specifying the valid transitions between program configurations. In other words, program verification in matching logic can be formalized as *reachability checking* of specific configuration patterns. Verification in Hoare's style can be defined as a reachability check between the configuration satisfying the pre-condition and the one satisfying the post-condition. The state-of-the-art work for reachability checking of matching logic properties for C programs has been proposed in [13]; it uses the $\mathbb{K}$ framework as rewrite system. Reference [13] shows that the proof system of matching logic is correct and relatively complete.

```
⟨program⟩ ::= ⟨function_definition⟩
⟨compound_decl⟩ ::= ⟨parameter⟩ ';' ⟨compound_stm⟩
⟨parameter⟩ ::= ⟨type⟩ IDENTIFIER
⟨type⟩ ::= 'int'
⟨id⟩ ::= IDENTIFIER
⟨compound_stm⟩ ::= Annotation? 'while''('⟨exp⟩')''{'⟨compound_stm⟩'}'
        ⟨compound_stm⟩ | ⟨stm⟩ | ⟨exp⟩';' ⟨compound_stm⟩
⟨stm⟩ ::= ⟨exp⟩ ';' | 'return' exp ';'
⟨exp⟩ ::= ⟨unary_exp⟩ ('='|'+='|'-='|'*='|'/='|'%=') ⟨exp⟩ | ⟨relat_exp⟩
⟨relat_exp⟩ ::= ( ⟨relat_exp⟩('>'|'<'|'>='|'<=') )?⟨unary_exp⟩
⟨function_definition⟩ ::= ⟨type⟩ IDENTIFIER ⟨function_def2⟩
⟨function_def2⟩ ::= '(' ⟨parameter⟩ ')' Annotation?
        '{' (⟨compound_stm⟩| ⟨compound_decl⟩)? '}' ⟨function_def⟩?
⟨postfix_exp⟩ ::= ⟨id⟩ | Constant | IDENTIFIER ⟨postfix_exp2⟩
⟨postfix_exp2⟩ ::= '('⟨exp⟩')'
⟨unary_exp⟩ = ⟨postfix_exp⟩ | ('+'|'-'|'*') ⟨unary_exp⟩
```

Figure 1. Excerpt of the KernelC grammar

III. Syntax-driven Reachability Checking

Implementing reachability checking of matching logic specifications for KernelC programs within our syntax-driven verification framework requires the definition of a grammar that specifies the syntax of the programs to analyze, and of an attribute schema (defined on top of the grammar) that encodes the actual verification procedure.

An excerpt of the grammar of KernelC is shown in Figure 1. Notice that the grammar allows for annotating functions or loops with *contracts* (see the placeholder `Annotation`).

The contract of a function consists of its pre- and post-conditions, expressed in the form of configuration patterns. The contract of a loop has as pre-condition its invariant and as post-condition the invariant conjuncted with the negation of the loop condition. The object of our verification procedure is to check the satisfaction of all the contracts specified within the program. We assume that every function and every loop to be verified come with a contract; we will discuss how to relax this assumption later.

We associate to each node in the parse tree the following attributes: 1) K is a list representing the code of the program fragment underlying the subtree rooted in the node. For the leaves of the parse tree, K is directly defined by the corresponding terminal symbol. 2) R is the set of matching logic rules defining the semantics of the code fragment(s) contained in K, and instantiated for this code. 3) V is a set of *verification tasks*, which are special entities created when visiting a node annotated with a contract. A verification task

```
1:  function EVAL($vt = \langle C_R, R_v, c_t \rangle$)
2:    $R_v \leftarrow R_v \cup R$
3:    repeat
4:      for $c_i \in C_r$ do
5:        $Changed \leftarrow false$
6:        $temp \leftarrow \emptyset$
7:        for $r_i \in R_v$ do
8:          if $Matches(c_i, r_i)$ then
9:            $c' \leftarrow ApplyRule(c_i, r_i)$
10:           if $isSatisfiable(c')$ then
11:             $temp \leftarrow temp \cup c'$
12:           end if
13:         end if
14:       end for
15:       if $temp \neq \emptyset$ then
16:         $Changed \leftarrow true$
17:         $C_r \leftarrow C_r \cup temp \setminus \{c_i\}$
18:       end if
19:     end for
20:   until $\neg Changed$
21:   for $c_i \in C_r$ do
22:     if $\neg IsFinal(c_i)$ then
23:       return $delay$
24:     end if
25:   end for
26:   for $c_i \in C_r$ do
27:     if $\neg satisfy(c_i, c_t)$ then
28:       return $false$
29:     end if
30:   end for
31:   return $true$
32: end function
```

Figure 2. The `eval` algorithm for verification tasks

is a triple $vt = \langle C_r, R_v, c_t \rangle$, where C_r is a set of configuration patterns, R_v is a set of reachability rules, and c_t is the target configuration pattern, which encodes the post-condition.

Verification tasks play a crucial role in our verification procedure, since a verification task verifies whether the contract in a certain node holds. In other words, it checks whether every execution path originating from a configuration pattern satisfying the pre-condition of the contract will eventually reach a configuration pattern satisfying its post-condition.

When a new verification task is instantiated in an annotated node to verify its contract, its component C_r contains only the configuration pattern which is constructed joining the pre-condition with the code K synthesized from the children nodes. R_v is initialized with the set R of semantic rules synthesized from the children, related to the code K.

The algorithm `eval` in Figure 2 is used to verify a contract within a verification task. First, the algorithm adds to R_v the rules contained in the attribute R of the nodes where the verification task is evaluated. Afterwards, all the rules in R_v are applied to the configuration patterns in C_r (lines 4–19). The application of a matching logic rule to a configuration pattern c (line 9) corresponds in general to the abstract execution of the next statement in cell k of c. Such statement is thus consumed moving the abstract execution towards a new configuration pattern c'. We check (through the function `isSatisfiable` via an SMT solver) if the constraints in this new configuration pattern are satisfiable. If this is the case we add c' to an auxiliary list *temp*. When all the applicable rules have been applied to c, if at least a new configuration has been discovered, c is removed from C_r, while the newly generated configuration patterns stored in *temp* are added to C_r, representing the maximal front of reachable configuration patterns from the initial configuration (lines 15–17). The application of the rules in R_v to C_r is iteratively performed until reaching a fixed point, i.e., when no rule from R_v is applicable to any configuration pattern in C_r. When the fixed point is reached (line 20), the configuration patterns in C_r might be either *final* or *non-final*.

We say that a configuration pattern is *final* if its cell k does not contain executable code, otherwise it is *non-final*. We also consider as final configuration patterns with cells that contain only the integer value returned by the evaluation of a statement, and the ones that contain the special token `ERROR`, which is introduced when an error situation arises, e.g., when the program invokes a function violating its pre-conditions or when an illegal operation is performed.

If C_r contains at least one non-final configuration pattern (lines 21–25), the verification task cannot be completed with the information available in the current node (e.g., when the code invokes a function that has not been parsed within the subtree rooted at the node). In this case, the verification task is marked with `delay` and propagated to the parent node, where more information could be available. If the new information in the parent node is again insufficient to complete the verification task, the latter is propagated up to the parent nodes, until the root is reached. If a verification task is not completed in the root, no conclusion can be drawn about the corresponding contract and the user may need to refine the annotations in the program. If all the configuration patterns in C_r are final the verification task is completed. We have now to check whether all these patterns match the target configuration pattern c_t (line 26–30). We do it with the help of an SMT solver, through the invocation of function `satisfy` (line 27). The contract is satisfied if and only if all the configurations match c_t; unsatisfied otherwise.

Our syntax-driven reachability checking procedure computes, for each node in the parse tree, the attributes in the following way. It takes as input the attributes computed for all the children of the node. Attribute K is built by concatenating the code fragment for the node with the lists of attributes K of the children, preserving the order of execution of the corresponding statements. Attribute R is the union of the corresponding attributes R of the children, plus the reachability rules created for the node. For computing attribute V, we first collect the incomplete verification tasks from children nodes (by calling the `eval` function on each of them), and save them in the temporary variable. If the current node is annotated with a contract, we generate a new verification task. If this verification task is incomplete, we add it to the temporary list. We also redefine R with the rules representing the contract in the annotation, i.e., the rules mapping its pre-condition to its post-condition. Moreover, if the current node is a function definition, we remove from K the code fragment corresponding to the function body.

IV. Reachability Checking Step by Step

In this section we show the attribute synthesis process that encodes the reachability checking procedure for KernelC programs annotated with matching logic specifications. To ground the concepts, we walk through the analysis of the simple program in Figure 3. This program consists of two function definitions; the main function is `sum_iterative(int n)`, which returns the opposite of the sum of the first `n` integers.

The contract of the function, specified on Line 3, defines both the pre-condition (`n>=0`) and the post-condition, which states that the return value has to be `-(n*(n+1))/2`.

```
 1 int neg(int n){ return −n; }
 2 int sum_iterative(int n){
 3 //@rule <k> $ => return −(n∗(n+1)) / 2; ...</k> if n>=0
 4 {
 5    int s; s = 0;
 6    //@inv s = −(old(n)−n) ∗ (old(n)+n+1) / 2 ∧ n>=0
 7    while (n > 0) {
 8         s += neg(n);
 9         n −= 1;
10    }
11    return s;
12 }
```

Figure 3. Listing of the Running Example

The first step for applying our approach is the construction of the parse tree, which is partially sketched in Figure 4; the nodes highlighted in the figure will be referenced and discusssed next. We recall that our attributes are synthesized-only; this allows us to start the attributes synthesis from any node whose children attributes have been already evaluated.

For the sake of readability, we use a simplified version of the reachability rules, which omits advanced features like the representation of the stack in the configurations and the management of multiple return points. Nonetheless, our approach supports these features. We use the following naming convention for symbols: letters at the beginning of the alphabet correspond to symbolic integer constants whose scope is confined within each rule; letters at the end of the alphabet represent generic configuration variables with a global scope. Furthermore, we use the following abbreviations for code/environment snippets:

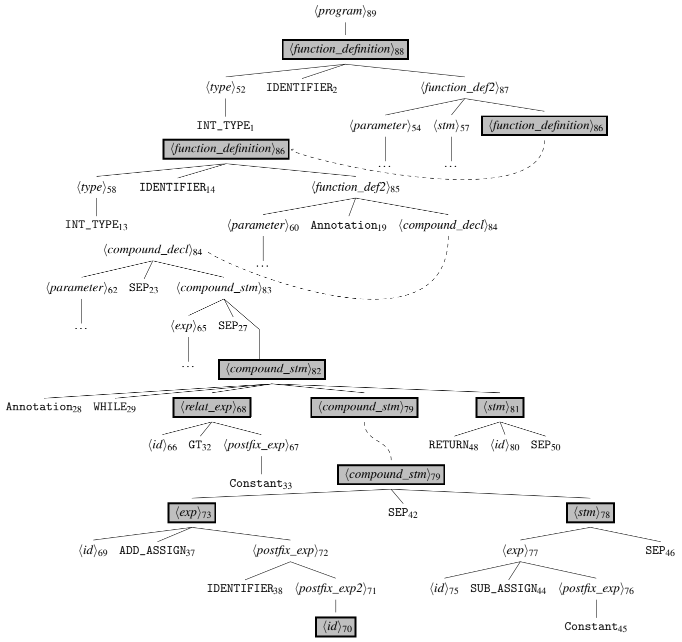
LB	$\equiv$	s += neg(n); n −= 1;
L	$\equiv$	**while** (n > 0){LB}
SUM	$\equiv$	((old(n)−n)∗(old(n)+n+1))/2
$ANNOT$	$\equiv$	*//@inv s*=$-SUM$ ∧ n>=0
RS	$\equiv$	**return** s;
FB	$\equiv$	**int** s; s=0; $ANNOT$ L **return** s;
e	$\equiv$	$\texttt{n} \mapsto x, \texttt{s} \mapsto y$
e''	$\equiv$	$\texttt{n} \mapsto x'', \texttt{s} \mapsto y''$

and we introduce a parametric macro expansion $p(a,b,c)$ for the logical expression $a \geq 0 \wedge b = -(c-a)*(c+a+1)/2$ and macro $p'(a,b,c)$ for expression $p(a,b,c) \wedge a > 0$.

We first illustrate how the attribute synthesis works for the basic constructs of the KernelC language, by starting the analysis from the statements included in the loop body.

Node 73. This node corresponds to the statements at Line 8 of Figure 3. In this node, function `neg` is invoked and its return value added to variable `s`; the operations involved are the addition-assignment, the invocation of `neg(n)`, and the evaluation of the actual parameter `n`.

The semantic rule associated to the addition-assignment is shown in Equation (2). This rule states that the execution can move from a configuration matching the pattern in which 1) cell k contains the code `s`+=b and 2) in cell env variable `s` is assigned a certain value a, to another configuration where 1)



$\langle program \rangle_{89}$
$\langle function\_definition \rangle_{88}$
$\langle type \rangle_{52}$
IDENTIFIER$_2$
$\langle function\_def2 \rangle_{87}$
INT_TYPE$_1$
$\langle parameter \rangle_{54}$
$\langle stm \rangle_{57}$
$\langle function\_definition \rangle_{86}$
$\langle function\_definition \rangle_{86}$
...
...
$\langle type \rangle_{58}$
IDENTIFIER$_{14}$
$\langle function\_def2 \rangle_{85}$
INT_TYPE$_{13}$
$\langle parameter \rangle_{60}$
Annotation$_{19}$
$\langle compound\_decl \rangle_{84}$
$\langle compound\_decl \rangle_{84}$
...
$\langle parameter \rangle_{62}$
SEP$_{23}$
$\langle compound\_stm \rangle_{83}$
...
$\langle exp \rangle_{65}$
SEP$_{27}$
...
$\langle compound\_stm \rangle_{82}$
Annotation$_{28}$
WHILE$_{29}$
$\langle relat\_exp \rangle_{68}$
$\langle compound\_stm \rangle_{79}$
$\langle stm \rangle_{81}$
$\langle id \rangle_{66}$
GT$_{32}$
$\langle postfix\_exp \rangle_{67}$
RETURN$_{48}$
$\langle id \rangle_{80}$
SEP$_{50}$
Constant$_{33}$
$\langle compound\_stm \rangle_{79}$
$\langle exp \rangle_{73}$
SEP$_{42}$
$\langle stm \rangle_{78}$
$\langle id \rangle_{69}$
ADD_ASSIGN$_{37}$
$\langle postfix\_exp \rangle_{72}$
$\langle exp \rangle_{77}$
SEP$_{46}$
IDENTIFIER$_{38}$
$\langle postfix\_exp2 \rangle_{71}$
$\langle id \rangle_{75}$
SUB_ASSIGN$_{44}$
$\langle postfix\_exp \rangle_{76}$
$\langle id \rangle_{70}$
Constant$_{45}$

Figure 4. Parse tree (simplified) of the running example

in cell k, the residual program `s`+=b has been executed and 2) in cell env, `s` is assigned a value $c = a + b$.

$$\left\langle \left\langle \frac{\texttt{s += } b}{c} \cdots \right\rangle_k \left\langle \cdots \texttt{s} \mapsto \frac{a}{c} \cdots \right\rangle_{env} \right\rangle \wedge c = a + b \tag{2}$$

As for the invocation of function `neg(n)`, no rule is available at this stage, since the function has not been analyzed yet. Finally, the evaluation of the actual parameter `n` is formalized by the rule in Equation (3), which has been propagated from Node $\langle id \rangle_{70}$; the details of the attributes synthesis for Node $\langle id \rangle_{70}$ have been omitted for brevity.

$$\left\langle \left\langle \frac{\texttt{n}}{a} \cdots \right\rangle_k \langle \cdots \texttt{n} \mapsto a \cdots \rangle_{env} \right\rangle \tag{3}$$

The attributes of Node 73 can then be defined as: $K_{73} = \{\texttt{s += neg(n)}\}$; $R_{73} = \{(2),(3)\}$; $V_{73} = \emptyset$ where K_{73} represents the code of the program fragment underlying the subtree rooted in Node 73; R_{73} is the set of matching logic rules defining the semantics of the code in K_{73}; V_{73} is the set of uncompleted verification tasks, which is empty since no verification tasks have been instantiated up to this node.

Node 78. This node refers to the subtraction-assignment at Line 9 of Figure 3; its semantic rule is shown in Equation (4).

$$\left\langle \left\langle \frac{\texttt{n -= 1;}}{\cdot} \cdots \right\rangle_k \left\langle \cdots \texttt{n} \mapsto \frac{a}{b} \cdots \right\rangle_{env} \right\rangle \wedge b = a - 1 \tag{4}$$

Notice that a single dot in a cell of a configuration pattern means that the cell is empty. The attributes of Node 78 can then be defined as: $K_{78} = \{\texttt{n -= 1;}\}$; $R_{78} = \{(4)\}$; $V_{78} = \emptyset$.

Node 79. After analyzing Nodes 73 and 78, we can now analyze their parent node 79, which is a $\langle compound\_stm \rangle_{79}$ concatenating the two statements rooted at Nodes 73 and 78. The semantic rule generated in this node which consumes the value c resulting from (2) is:

$$\left\langle \left\langle \frac{c\texttt{;}}{\cdot} \cdots \right\rangle_k \langle \cdots \rangle_{env} \right\rangle \tag{5}$$

The attributes computed at Node 79 are: $K_{79} = \{LB\}$; $R_{79} = \{(2),(3),(4),(5)\}$; $V_{79} = \emptyset$.

Before proceeding upward along the parse tree to Node 82 (corresponding to the while loop) we have to analyze its siblings nodes 68 and 81.

Node 68. This node represents the loop condition at Line 7 of the example; its semantic rule is shown in Equation (6). The value b (left in cell k after evaluating the condition `n > 0`) is defined according to the C convention where 0 means *false* and any other value means *true*.

$$\left\langle \left\langle \frac{\texttt{n>0}}{b} \cdots \right\rangle_k \langle \ldots \texttt{n} \mapsto a \ldots \rangle_{env} \right\rangle \wedge b = a > 0 \tag{6}$$

Attributes of Node 68 are: $K_{68} = \{\texttt{n>0}\}$; $R_{68} = \{(6)\}$; $V_{68} = \emptyset$.

Node 81. This node corresponds to the return statement at Line 11. Since in this example we do not consider stack management and multiple return points, the operation of popping the frame of the function off of the stack is omitted. The semantic rule created in this node is shown in Equation (7).

$$\left\langle \left\langle \frac{\texttt{return s;}}{a} \cdots \right\rangle_k \langle \ldots \texttt{s} \mapsto a \ldots \rangle_{env} \right\rangle \tag{7}$$

The attributes synthesized in this node are: $K_{81} = \{\texttt{return s;}\}$; $R_{81} = \{(7)\}$; $V_{81} = \emptyset$.

Node 82. This node contains the loop and its contract annotation (plus the return statement, which has already been discussed above with Node 81).

The semantic rules needed to analyze this node are shown in Equations (8), (9), and (10). Rule (8) first checks that the invariant holds before starting executing the loop (when $n = old(n)$), and then substitutes the values of `n` and `s` with new ones that satisfy the invariant. Rule (9) characterizes the cases in which the loop condition is false (which inhibits the execution of the loop). Rule (10) characterizes the execution of a single loop iteration loop, which requires its condition to be satisfied; this rule is needed for verifying the invariant.

$$\left\langle \left\langle \frac{ANNOT\ L}{L} \cdots \right\rangle_k \left\langle \ldots \texttt{n} \mapsto \frac{a}{c}, \texttt{s} \mapsto \frac{b}{d} \cdots \right\rangle_{env} \right\rangle$$
$$\wedge b = 0 \wedge a \geq 0 \wedge d = -(a-c)*(a+c+1)/2 \wedge c \geq 0 \tag{8}$$

$$\left\langle \left\langle \frac{\texttt{while(}a\texttt{)}\ \{LB\}}{\cdot} \cdots \right\rangle_k \langle \ldots \rangle_{env} \right\rangle \wedge a = 0 \tag{9}$$

$$\left\langle \left\langle \frac{\texttt{if(}a\texttt{)}\ \{LB\}}{LB} \cdots \right\rangle_k \langle \ldots \rangle_{env} \right\rangle \wedge a \neq 0 \tag{10}$$

The attributes K and R synthesized at Node 82 are: $K_{82} = \{ANNOT\ L\ RS\}$; $R_{82} = R_{68} \cup R_{79} \cup R_{81} \cup \{(8),(9),(10)\}$.

As for attribute V_{82}, since Node 82 contains an annotation, we need to instantiate a new verification task (called vt_{82}) to verify whether the contract holds. The initial and target configuration patterns of the verification task can be constructed from $\langle Annotation \rangle_{28}$ and attributes K_{68}, K_{79}, as shown in Equations (11) and (12).

$$\langle \langle \texttt{if(n>0)\{}\ LB\ \texttt{\}} \rangle_k \langle \texttt{n} \mapsto x, \texttt{s} \mapsto y \rangle_{env} \rangle \wedge p(x,y,z) \tag{11}$$

$$\langle \langle \cdot \rangle_k \langle \ldots \texttt{n} \mapsto x', \texttt{s} \mapsto y' \ldots \rangle_{env} \rangle \wedge p(x',y',z) \tag{12}$$

To instantiate the verification task vt_{82}, we initialize the set of reachable configuration patterns to $C_{r_{82}} = \{(11)\}$, the target configuration patterns to $c_{t_{82}} = (12)$, and the set of semantic rules to $R_{v_{82}} = R_{82}$. When the verification task is instantiated, the only applicable rule is rule (6) ; it evaluates the loop condition and generates the configuration pattern in Equation (13).

$$\langle \langle \texttt{if(}q\texttt{)}\ \{LB\} \rangle_k \langle e \rangle_{env} \rangle \ \wedge\ p(x,y,z)\ \wedge\ q\ =\ x\ >\ 0 \tag{13}$$

The constraint $p(x,y,z) \wedge q = x > 0$ (part of Equation (13)) is passed to the SMT solver, which checks its satisfiability. Since in this case the constraint is satisfiable, the newly-generated configuration pattern replaces the initial configuration pattern in $C_{r_{82}}$. In this new configuration pattern, we can apply rule (10), obtaining the configuration pattern in Equation (14).

$$\langle \langle LB \rangle_k \langle \texttt{n} \mapsto x, \texttt{s} \mapsto y \rangle_{env} \rangle\ \wedge\ p(x,y,z)\ \wedge\ x\ >\ 0 \tag{14}$$

Finally, by applying Rule (3) on (14), we get the new configuration pattern in Equation (15).

$$\langle \langle \texttt{s+=neg(}x\texttt{);\ n-=1;} \rangle_k \langle e \rangle_{env} \rangle \wedge p(x,y,z) \wedge x > 0 \tag{15}$$

At this point, when $C_{r_{82}} = \{(15)\}$, no rule is applicable, because the next statement to be matched in cell k of (15) would require the invocation of function `neg(n)`, which is unknown at the current stage.

Since the configuration pattern in $C_{r_{82}}$ is not final (because its cell k contains executable code), the verification task vt_{82} cannot be completed. Hence, it is added to attribute V_{82}, which will be propagated to the parent node. We have $V_{82} = \{vt_{82}\} = \langle C_{r_{82}}, c_{t_{82}}, R_{v_{82}} \rangle$. Attribute R_{82} is also redefined removing the rules related to the loop body, which are already stored in vt_{82}. The final attributes for Node 82 are $K_{82} = \{ANNOT\ L\ RS\}$; $R_{82} = R_{68} \cup R_{81} \cup \{(8),(9)\}$; $V_{82} = \{vt_{82}\}$.

Node 86. This node contains the definition of function `sum_iterative`, including both the contract annotation and the body. Equations (16) and (17) are the semantic rules for this node, characterizing the function invocation. Rule (16) corresponds to the case in which the pre-condition of the function (`n >= 0`) is satisfied; in such a case, as shown in cell k, the execution of the function yields value b, which is constrained to satisfy the post-condition. Rule (17) drives the execution toward an error configuration (denoted with the special symbol *ERROR* in cell k) when the arguments of the function invocation violate its pre-condition (captured by $a < 0$ in the example).

$$\left\langle \left\langle \frac{\texttt{sum\_iterative(}a\texttt{)}}{b} \cdots \right\rangle_k \langle \ldots \rangle_{env} \right\rangle$$
$$\wedge b = -a*(a+1)/2 \wedge a \geq 0 \tag{16}$$

$$\left\langle \left\langle \frac{\texttt{sum\_iterative(}a\texttt{)}}{ERROR} \cdots \right\rangle_k \langle \ldots \rangle_{env} \right\rangle \wedge a < 0 \tag{17}$$

The attributes K and R synthesized at Node 86 at this stage are: $K_{86} = \{FB\}$; $R_{86} = R_{84} \cup \{(16),(17)\}$, where R_{84} contains R_{82} and rule (18); the latter is the semantic rule for the code at line 5 of the example (not shown here for brevity).

$$\left\langle \left\langle \frac{\texttt{int s; s=0;}}{\cdot} \cdots \right\rangle_k \left\langle \cdots \frac{\cdot}{\texttt{s} \mapsto a} \cdots \right\rangle_{env} \right\rangle \wedge a = 0 \tag{18}$$

As for the attribute V_{86}, besides the verification task propagated from Node 82, a new task vt_{86} has to be instantiated to verify the contract of `sum_iterative`. We need to identify the initial configuration for initializing $C_{r_{86}}$, the target configuration $c_{t_{86}}$ (representing the post-condition), and the set of semantic rules $R_{v_{86}}$. By combining the pre-condition and the code in K_{86}, we obtain the initial configuration pattern in Equation (19). The post-condition is instead encoded in the configuration pattern in Equation (20). All the required semantic rules are already contained in R_{86}.

$$\langle\langle FB\rangle_k \langle \mathrm{n} \mapsto x\rangle_{env}\rangle \wedge x \geq 0 \qquad (19)$$

$$\langle\langle y'\rangle_k \langle \dots\rangle_{env}\rangle \wedge y' = -x*(x+1)/2 \qquad (20)$$

Starting from the initial configuration in $C_{r_{86}}$, the only applicable rule is (18), which yields the new configuration pattern (21) where the variable `s` has been declared and initialized. This new configuration is added to $C_{r_{86}}$, while the initial configuration is removed.

$$\langle\langle ANNOT\ L\ RS\rangle_k \langle \mathrm{n} \mapsto x, \mathrm{s} \mapsto y\rangle_{env}\rangle \wedge x \geq 0 \wedge y = 0 \qquad (21)$$

From the pattern (21), we can apply rule (8), obtaining the configuration pattern in (22). This rule processes the loop assuming its contract holds, and constraints the final value of s according to the contract.

$$\langle\langle L\ RS\rangle_k \langle \mathrm{n} \mapsto x'', \mathrm{s} \mapsto y''\rangle_{env}\rangle \wedge x \geq 0 \wedge p(x'',y'',x) \qquad (22)$$

From pattern (22), we apply rule (9) to obtain (23).

$$\langle\langle RS\rangle_k \langle e''\rangle_{env}\rangle \wedge p(x'',y'',x) \wedge x'' = 0 \wedge x \geq 0 \qquad (23)$$

Finally, we obtain through (7) the configuration pattern (24)), which is a final configuration pattern, since its residual program does not contain any executable code.

$$\langle\langle y''\rangle_k \langle e''\rangle_{env}\rangle \wedge x > 0 \wedge x'' = 0 \wedge y'' = -x*(x+1)/2 \qquad (24)$$

Since the only configuration pattern in $C_{r_{86}}$ is final and it matches the target configuration pattern $c_{t_{86}}$ (by the renaming $y' \mapsto y''$), we invoke the SMT solver to check whether the constraint $x > 0 \wedge x'' = 0 \wedge y'' = -x*(x+1)/2$ implies $y' = -x*(x+1)/2$ (upon renaming). Since this implication holds, the verification is successful and the verification task vt_{86} is completed (and thus removed from V_{86}). This means that function `sum_iterative` complies with its contract. Attributes K_{86} and R_{86} are redefined by removing the rules already stored in the verification task; the new attributes are $K_{86} = \{\}$; $R_{86} = \{(16),(17)\}$; $V_{86} = \{vt_{82}\}$. Notice that, since no information about `neg` has been collected up to Node 86, the verification task vt_{82}, propagated from Node 82, cannot be completed. It is thus kept into the attribute V_{86}, to be propagated upwards.

Node 88. The definition of function `neg` becomes available when the attribute synthesis reaches Node 88. Since no contract is specified for this function, its semantic rule in Equation (25) summarizes the execution of the function body: the execution yields a value b in cell k, which is the inverse of the input parameter.

$$\left\langle\left\langle \frac{\mathtt{neg}\,(a)}{b} \dots \right\rangle_k \langle \dots\rangle_{env}\right\rangle \wedge b = -a \qquad (25)$$

The attributes K and R for this node are initially defined as $K_{88} = \{$`return -n;`$\}$ and $R_{88} = R_{86} \cup \{(25)\}$. As for V_{88}, though no contract is specified in Node 88, it receives the incomplete verification task vt_{82} from Node 86. The new semantic rules discovered up to Node 88 (i.e., R_{88}) are added to vt_{82} and the verification task is resumed.

With $C_{r_{82}} = \{(15)\}$, we can apply (25) on the configuration pattern (15), obtaining (26), where the invocation of `neg` has been evaluated.

$$\langle\langle \mathtt{s+=}q'\mathtt{;}\ \mathtt{n-=1;}\rangle_k \langle e\rangle_{env}\rangle \wedge p'(x,y,z) \wedge q' = -x \qquad (26)$$

The residual code can then be processed using Rules (2) and (5) in this order, yielding the configuration pattern (27).

$$\langle\langle \mathtt{n-=1;}\rangle_k \langle \mathrm{n} \mapsto x, \mathrm{s} \mapsto (y-x)\rangle_{env}\rangle \wedge p'(x,y,z) \qquad (27)$$

Finally, from this configuration pattern, by applying Rule (4), we reach the pattern (28).

$$\langle\langle \mathtt{.}\rangle_k \langle e''\rangle_{env}\rangle \wedge p'(x,y,z) \wedge x'' - 1 \wedge y'' = y - x \qquad (28)$$

Since the only configuration pattern in $C_{r_{88}}$ is final and it matches the target configuration pattern $c_{t_{82}}$ (by the renaming $x' \mapsto x''$, $y' \mapsto y''$), we invoke the SMT solver to check whether the constraint $p'(x,y,z) \wedge x'' = x - 1 \wedge y'' = y - x$ implies $p(x',y',z)$ (upon renaming). Since this implication holds, the verification is successful and the verification task vt_{82} is completed (and removed from V_{88}).

The attributes for node 88 are then $K_{88} = \{\}$, $R_{88} = R_{86} \cup \{(25))\}$, and $V_{88} = \emptyset$.

Discussion. As seen in the example for the definition of function `neg`, when no contract is provided for a function definition, its semantic rules correspond to those obtained by inlining the function body. In case of recursive calls, this approach may not terminate, since the inlining rule can be applied infinitely many times. A similar issue concerns non-annotated loops. Our approach can only unroll the loop up to a given bound. This is compatible with the original definition of reachability for matching logic [13], where the verification may not terminate in these cases. In our tool, we added the option to specify a maximum number of inlining operations, in order to perform a bounded verification of non-annotated recursive programs. Finally, we remark that we have omitted the description of the matching logic rules concerning memory management, due to space reasons; nevertheless, our tool supports dynamic data structures. The definitions of these rules are conceptually similar to those for the basic constructs shown above, and conform to the semantics specified in [13].

V. Preliminary Evaluation

We have implemented our approach as a plugin [17] for our general-purpose syntax-direct verification framework SiDECAR [10]. The plugin is implemented in Java and uses the Z3 [7] SMT solver to check pattern reachability. In this section we report on our preliminary evaluation, comparing SiDECAR with MatchC [12], the state-of-the-art tool developed for verification of C programs with matching logic. The benchmarks have been executed on quad-core machine with 8GB of memory, using Z3 v.4.2 and MatchC v.1.0r563.

The first benchmark includes the sample programs[2] (and their specifications in matching logic) accompanying the

[2]For this preliminary report, we excluded from the benchmark the programs using advanced data structures such as AVL trees, which are supported by MatchC thanks to the native libraries included in the underlying Maude tool. As part of future work, we plan to incorporate theories for these data structures also in Z3. Notice that this issue is orthogonal to the definition of our syntax-directed verification procedure.

Table I
EXECUTION TIMES (IN MS) OF THE 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
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

(a) MATCHC examples

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

(b) `fib.c`

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

(c) `memWhile.c`

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

(d) `sort.c`

MATCHC distribution[3]; the execution times are shown in Table I(a). Our implementation, despite being based on a general-purpose syntax-directed verification framework, which inevitably adds some overhead, exhibits execution times similar to and often even better than the ad-hoc implementation.

The second benchmark includes three programs, available on GitHub[4], which stress the bounded verification procedure by analyzing underspecified programs, i.e., lacking invariants and contracts for recursive functions. The first program, `fib.c`, computes the nth Fibonacci number using a recursive algorithm. We selected it to assess the performance of our approach in the case of a large number of (recursive) function calls; Table I(b) shows the execution time when varying the parameter n. The second program, `memWhile.c`, walks through a list of a certain length; the third program, `sort.c`, performs insertion sort on a list of a certain length (this is the same code analyzed for insertion sort in Table I(a) where the invariants have been removed). Both programs assess the performance of our approach when dealing with dynamic data structures; the execution times for various list lengths are shown in Tables I(c) and I(d), respectively. In all cases, when the complexity of the program increases (e.g., because of the size of the call stack or the length of a list), our tool outperforms MATCHC.

[3]http://fsl.cs.illinois.edu/index.php/Special:MatchCOnline
[4]https://github.com/alessandro89/sidecarBenchmarks

VI. CONCLUSION AND FUTURE WORK

In this paper we have presented a syntax-direct approach for the verification of KernelC programs with properties expressed in matching logic. In our approach, the verification procedure is expressed as Knuth's semantic attributes that are computed and propagated through the syntax tree, relying on an SMT solver for the evaluation of logic formulae. Our preliminary evaluation shows that our tool achieves similar results, often better, than the state-of-the-art verifier, despite the overhead introduced by the syntax-driven framework.

Our long-term goal is to develop a generalized framework for implementing verifiers that can naturally support continuous program changes, exploiting incremental verification [14]. The work described here is part of this research line and represents one of its first steps. We plan to further develop it to fully support *incremental* reachability checking of C programs with properties expressed in matching logic. The preliminary results achieved with an incremental version of our tool (not described here because of space reasons) look very promising.

REFERENCES

[1] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. of LICS '02*. IEEE, 2002, pp. 55–74.

[2] G. Roşu, C. Ellison, and W. Schulte, "Matching logic: An alternative to Hoare/Floyd logic," in *Proc. of AMAST '10*, ser. LNCS, vol. 6486, 2010, pp. 142–162.

[3] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *STTT*, vol. 9, pp. 505–525, 2007.

[4] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *JASE*, vol. 10, no. 2, pp. 203–232, 2003.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. of PLDI '02*. ACM, 2002, pp. 234–245.

[6] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proc. of FMCO '05*, ser. LCNS. Springer, 2006, vol. 4111, pp. 364–387.

[7] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. of TACAS'08/ETAPS'08*. Springer, 2008, pp. 337–340.

[8] D. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[9] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella, "Parallel parsing of operator precedence grammars," *Inf. Process. Lett.*, vol. 113, no. 7, pp. 245–249, 2013.

[10] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli, "Syntactic-semantic incrementality for agile verification," *Sci. Comput. Program.*, vol. 97, part 1, no. 0, pp. 47–54, 2015.

[11] ——, "Incremental syntactic-semantic reliability analysis of evolving structured workflows," in *Proc. of ISOLA 2014*, ser. LNCS. Springer, 2014, vol. 8802, pp. 41–55.

[12] A. Ştefănescu, "MatchC: A matching logic reachability verifier using the framework," *Electron. Notes Theor. Comput. Sci.*, vol. 304, no. 0, pp. 183 – 198, 2014.

[13] G. Roşu and A. Ştefănescu, "Checking reachability using matching logic," in *Proc. of OOPSLA'12*. ACM, 2012, pp. 555–574.

[14] C. Ghezzi, "Evolution, adaptation, and the quest for incrementality," in *Proc. of the 17th Monterey Workshop*, ser. LNCS, vol. 7539. Springer, 2012, pp. 369–379.

[15] C. Ghezzi and D. Mandrioli, "Incremental parsing," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 58–70, 1979.

[16] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *J.LAP*, vol. 79, no. 6, pp. 397–434, 2010.

[17] A. M. Rizzi, "Incremental reachability checking of KernelC programs using matching logic," in *Companion of ICSE'14 Proc.* ACM, 2014, pp. 724–726.