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 evaluation¹, 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

¹The approach pursued in this paper is typically bottom-up, i.e., based on purely synthesized attributes. 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 \(\langle (x=2)_k \langle \ldots x \mapsto \ldots \rangle_{env} \rangle\), 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:

\[
\langle (x=2)_k \langle \ldots x \mapsto \ldots \rangle_{env} \rangle \land a \geq 0
\]

where \(a\) is an existentially quantified integer variable, \(x\) 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 \(\psi\) and \(\psi'\); informally, such a rule states that a program configuration matching pattern \(\psi\) takes zero or more steps to reach another configuration that matches pattern \(\psi'\). 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 \(\langle (x=2)_k \langle \ldots x \mapsto \ldots \rangle_{env} \rangle \land a \geq 0 \Rightarrow \langle (x=2)_k \langle \ldots x \mapsto \ldots \rangle_{env} \rangle \land a \geq 0\) 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.

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
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 $C_t$ 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 $C_t$ 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.

```
1: function Eval(vk = (C_k, R_k, c_k))
2:     R_k ← R_k ∪ R
3:     repeat
4:         for c_t ∈ C_t do
5:             Changed ← false
6:             temp ← 0
7:             for r_t ∈ R do
8:                 if Matches(r_t, c_t) then
9:                     c_t′ ← ApplyRule(r_t, c_t)
10:                    temp ← temp + c_t′
11:                 end if
12:             end if
13:         end if
14:     end for
15:     if temp ≠ 0 then
16:         Changed ← true
17:     end if
18:     C_t ← C_t ∪ temp ∖ {c_t}
19:     end if
20: until Changed
21: end if
22: if isFinal(c_k) then
23:     return delay
24: end if
25: if c_k ∈ C_k do
26:     for r_k ∈ R_k do
27:         if satisfy(c_k, c_k) then
28:             return false
29:         end if
30:     end for
31:     return true
32: end function
```

Figure 2. The `eval` algorithm for verification tasks
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
discussed 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/en-
vironment snippets:

\[
\begin{align*}
LB & \equiv s := \neg(n); n := 1; \\
L & \equiv \text{while } (n > 0) \{LB\} \\
SUM & \equiv ((\text{old}(n) - n) * (\text{old}(n)+n+1)) / 2 \land n > 0 \\
ANNOT & \equiv @\text{inv } s := \neg(n) \\
RS & \equiv \text{return } s; \\
FB & \equiv \text{int } s; s = 0; \\
e & \equiv n \mapsto x, s \mapsto y \\
e' & \equiv n \mapsto x', s \mapsto y'
\end{align*}
\]

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

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

\[
\begin{array}{l}
\langle s += b \rangle _k \langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle n \mapsto a \rangle _k \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\end{array}
\]

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 \{id\}_70; the details of the attributes synthesis for Node
\{id\}_70 have been omitted for brevity.

\[
\begin{array}{l}
\langle n \mapsto \ldots \rangle _k \\
\langle \ldots \rangle _k \langle n \mapsto a \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\end{array}
\]

The attributes of Node 73 can then be defined as: \( K_{73} = \{s := \neg(n)\}; R_{73} = \{(2), (3)\} \}; V_{73} = \emptyset \) 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).

\[
\begin{array}{l}
\langle n -= 1 \rangle _k \\
\langle \ldots \rangle _k \langle n \mapsto a \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\langle \ldots \rangle _k \langle \ldots \rangle _k \\
\end{array}
\]

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: \( K_{78} = \{n -= 1\}; R_{78} = \{(4)\}; V_{78} = \emptyset \).

**Node 79.** After analyzing Nodes 73 and 78, we can now
analyze their parent node 79, which is a \{compound_stm\}_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:

\[
\begin{array}{l}
\langle \ldots \rangle _k \\
\langle c \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\langle \ldots \rangle _k \\
\end{array}
\]

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.

\[
\begin{align*}
\mathcal{R}_68 & = \{ n > 0 \} ; \mathcal{R}_68 = \{ (6) \} ; \mathcal{V}_68 = \emptyset .
\end{align*}
\]

This node contains the definition of function \( \text{sum} \)-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 \( \text{ERROR} \) in cell \( k \)) when the arguments of the function invocation violate its pre-condition (captured by \( a < 0 \) in the example).
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 $R_{86}$.

\[
\langle\langle FB \rangle\rangle_k (n \mapsto x)_{em} \wedge x \geq 0 = 0 \ \ (19)
\]
\[
\langle\langle y' \rangle\rangle_k (\ldots)_{em} \wedge y = -x*(x+1)/2 \ \ (20)
\]

Starting from the initial configuration in $C_{86}$, 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 $C_{86}$, while the initial configuration is removed.

\[
\langle\langle ANNOT L RS \rangle\rangle_k (n \mapsto x, s \mapsto y)_{em} \wedge x \geq 0 \wedge y = 0 = 0 \ \ (21)
\]

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

\[
\langle\langle L RS \rangle\rangle_k (n \mapsto x', s \mapsto y')_{em} \wedge x \geq 0 \wedge p(x', y', x) = 0 \wedge x \geq 0 \ \ (22)
\]

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

\[
\langle\langle RS \rangle\rangle_k (e'_{em}) \wedge p(x', y', x) \wedge x' = 0 \wedge x \geq 0 \ \ (23)
\]

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.

\[
\langle\langle y' \rangle\rangle_k (e'_{em}) \wedge x > 0 \wedge x' = 0 \wedge y' = -x*(x+1)/2 \ \ (24)
\]

Since the only configuration pattern in $C_{86}$ is final and it matches the target configuration pattern $c_{86}$ (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 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.

\[
\langle\langle neg \rangle\rangle_k (a)_{em} \wedge b = -a \ \ (25)
\]

The attributes K and R for this node are initially defined as $K_{88} = \{\text{return } -a;\}$ 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 $C_{82} = \{(15)\}$, we can apply (25) on the configuration pattern (15), obtaining (26), where the invocation of neg has been evaluated.

\[
\langle\langle s+q' \rangle\rangle_k (n=1)_{em} \wedge p'(x, y, z) \wedge q' = -x \ \ (26)
\]

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

\[
\langle\langle n=1 \rangle\rangle_k (n \mapsto x, s \mapsto (y-x))_{em} \wedge p'(x, y, z) \ \ (27)
\]

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

\[
\langle\langle y' \rangle\rangle_k (e'_{em}) \wedge p'(x, y, z) \wedge x'' - 1 \wedge y'' = y - x \ \ (28)
\]

Since the only configuration pattern in $C_{88}$ is final and it matches the target configuration pattern $c_{82}$ (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-directed 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.
The third program, sort.c, through a list of a certain length; the third program, sort.c, on GitHub, similar to and often even better than the ad-hoc implementation. Inevitably adds some overhead, exhibits execution times similar to those of the general-purpose syntax-directed verification framework, which 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.

### 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