ICSE'11, ACM, pp. 868-871. 2011

Matching Logic: A New Program Verification Approach (NIER Track)

Grigore Roşu Andrei Ştefănescu
Department of Computer Science
University of Illinois at Urbana-Champaign
{grosu, stefane1}@illinois.edu

ABSTRACT

Matching logic is a new program verification logic, which builds upon operational semantics. Matching logic specifications are constrained symbolic program configurations, called *patterns*, which can be *matched* by concrete configurations. By building upon an operational semantics of the language and allowing specifications to directly refer to the structure of the configuration, matching logic has at least three benefits: (1) One's familiarity with the formalism reduces to one's familiarity with the operational semantics of the language, that is, with the language itself; (2) The verification process proceeds the same way as the program execution, making debugging failed proof attempts manageable because one can always see the "current configuration" and "what went wrong", same like in a debugger; and (3) Nothing is lost in translation, that is, there is no gap between the language itself and its verifier. Moreover, direct access to the structure of the configuration facilitates defining subpatterns that one may reason about, such as disjoint lists or trees in the heap, as well as supporting framing in various components of the configuration at no additional costs.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Languages, Verification, Theory

Keywords

Matching logic, rewriting logic, Maude

1. INTRODUCTION

Program verification is hard. There are many well-known reasons for that, such as inherent complexity of programs, unexpected or missed assumptions, poor tools, lack of training and resources, etc., which we cannot detail here. One less mentioned reason is the formal foundations on which we build our current verification approaches. For example, could it be possible that novel verification foundations, conceptually different from Floyd-Hoare logic [4, 2], or separation logic [5, 6], or dynamic logic [3], wait to be unearthed in order to make program verification more feasible and accessible to non-experts? Common scientific sense tells the answer is more likely positive than negative. Without claiming that it overcomes all the problems of the existing approaches, this paper advances the possibility that *matching logic* [7] could be a promising novel verification foundation.

Matching logic builds upon operational semantics. To use it, one must understand at least the structure of the configurations that are used in the operational semantics of the language. For example, the configuration of some language may contain, besides the code itself, an environment, a heap, stacks, synchronization resources, etc. Matching logic specifications, called *patterns*, allow one to refer directly to the configuration of the program. For example, the pattern

$$\langle \quad \langle \mathsf{root} \mapsto ?\mathsf{root},\ \mathsf{E} \rangle_{\mathrm{env}} \quad \langle \mathsf{tree}(?\mathsf{root})(\mathsf{T}),\ \mathsf{H} \rangle_{\mathrm{heap}} \quad \mathsf{C} \quad \rangle_{\mathrm{config}}$$

specifies the set of configurations where program variable root points to tree T. More precisely: (1) the configuration $\langle \ldots \rangle_{\mathrm{config}}$ contains at least an environment cell and a heap cell, and the rest of the configuration is matched by C (the configuration frame); (2) the environment $\langle \ldots \rangle_{\mathrm{env}}$ holds at least the binding $\mathsf{root} \mapsto ?\mathsf{root}$, and the rest of the environment is matched by E (the environment frame); (3) the heap $\langle \ldots \rangle_{\mathrm{heap}}$ holds at least the term $\mathsf{tree}(?\mathsf{root})(\mathsf{T})$, and the rest of the heap is matched by H. The term $\mathsf{tree}(?\mathsf{root})(\mathsf{T})$ matches a portion of the heap that contains a flattened representation of the tree T in memory. As seen shortly, such terms are not defined, they are axiomatized.

Matching logic patterns can be defined as first-order logic (FOL) formulas [7] over a signature that includes all the constructs for configurations and everything they can hold, such as lists, sets, maps, trees, etc. Variables like ?root are existentially quantified over the pattern, while E, T, H, C are free. From a matching logic perspective, unlike in other program verification logics, program variables like root are *not* logical variables; they are simple syntactic constants. The matching logic derivation rules are nothing else but the operational semantic rules (allowed to work on configurations with variables). For example, an assignment statement "$\mathsf{x} = 5$" changes the pattern above into the pattern

$$\langle \quad \langle \mathsf{root} \mapsto 5,\ \mathsf{E} \rangle_{\mathrm{env}} \quad \langle \mathsf{tree}(?\mathsf{root})(\mathsf{T}),\ \mathsf{H} \rangle_{\mathrm{heap}} \quad \mathsf{C} \quad \rangle_{\mathrm{config}}$$

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

There is no backwards substitution like in the Hoare rule for assignment, as well as no introduction of existential quantifiers like in the Floyd rule for assignment [7]. The variables appearing in patterns often need to be constrained; as patterns are just FOL formulas, one can conjunct them with a formula expressing the desired constraints. For example:

$$\langle\langle \mathsf{root} \mapsto ?\mathsf{root}, \mathsf{E}\rangle_{\text{env}}\ \langle \mathsf{tree}(?\mathsf{root})(\mathsf{T}), \mathsf{H}\rangle_{\text{heap}}\ \mathsf{C}\rangle_{\text{config}} \wedge \mathsf{T} \neq \mathsf{empty}$$

There is no need for explicit "separation". In matching logic, separation is implicitly achieved at the structural (i.e., term) level and not at the logical level. For example, if one matches two terms in a multiset, then the two terms are obviously distinct. In our pattern above, the structural separation tells that the binding of root is separated from the rest of the environment E, the term tree(?root)(T) is separated from the rest of the heap H, and the three mentioned cells are separated from the rest of the configuration C.

Framing can appear in any cell of the configuration. Same like separation, it needs no special logical support, in particular no derivation rules. Cell framing simply falls under the general principle of matching. Consider, for example, the assignment "x=5" discussed above, and the free variable H. Since H appears free in both patterns, it must match the same term. In other words, any concrete program configuration that matches the first pattern will induce a binding for H, which will be the same after the assignment.

Like in Hoare logic, each programming language undergoing program verification needs to be associated a matching logic proof system. However, unlike in Hoare logic where there is an inherent gap between the operational semantics of the programming language and the Hoare logic proof system (which needs to be filled by proving corresponding soundness results), the matching logic proof system associated to a given programming language is derived directly from the operational semantics of the programming language [7].

Matching logic is very new (the first non-report paper was published in 2010 [7]). Hence, there is a lot of work to be done, both at the foundational and at the implementation levels. To test the practicality of the approach, we implemented a proof-of-concept matching logic verifier for a fragment of C, called MatchC; see `http://fsl.cs.uiuc.edu/ml` for download and online execution. MatchC builds upon an executable rewrite-based semantics of the fragment of C, extending it (unchanged) with semantics for pattern specifications. Both the executable semantics and the verifier are implemented using the $\mathbb{K}$ language definitional framework [8], which compiles into Maude [1].

MatchC is the main motivation for this paper. More precisely, the unexpectedly good results in terms of generality and automation obtained while using it on verifying challenging programs, like the ones in the next section. We have collected many such programs from other program verifiers, from colleagues, and from our own experience; they can be reached through the web interface at the URL above. Two factors guided us in our experiments: proving full correctness of programs (as opposed to just memory safety which is much easier) and doing so completely automatically (the user only provides the pattern specifications).

A preliminary version of this work was presented at a meeting on usable verification (UV'10) sponsored by the National Science Foundation and Microsoft Research in November 2010. The UV'10 papers have been made available to the participants, but no proceedings have been published.

2. MATCHC AT WORK

Here we discuss a few examples that MatchC can verify in milliseconds. Figure 1 shows a collection of list manipulating C functions: listReverse (reverses a linked list), listAppend (appends two linked lists), listReadWrite (reads a sequence of integers from standard input and writes it to standard output in reverse order). Figure 2 shows a function that flattens a tree into a list, traversing the tree in infix order.

Every function is annotated with pre and post pattern conditions. When a function is verified, its precondition is assumed as the (symbolic and constrained) configuration in which its body is executed using the operational semantics of the language. The semantics is extended with pattern assertions as expected: when a pattern assertion is encountered, MatchC attempts to prove that the current (symbolic and constrained) configuration matches the asserted pattern. The verification fails if any such match fails. One can assert patterns anywhere in the program. The postcondition is automatically asserted at function return. We call asserted patterns invariants when they are associated to loops. An invariant always holds before each loop iteration (we only consider partial correctness for the time being).

Note the patterns in Section 1 contain environment cells, while those in Figures 1 and 2 do not. MatchC's configurations internally hold environment cells mapping program variables to values, and the user can use them in patterns. However, as a user convenience, we implemented a simple shell to the semantics that computes and adds an environment cell to any configuration which does not already have one. This syntactic sugar allows for more compact patterns.

The precondition of listReverse states that parameter x points to a linked list holding the sequence A; the variable A is free in both the pre and the post conditions, indicating that it must be bound to the same sequence (defined algebraically in a trivial way; see below). There is another free variable in the pre and the post conditions, H, which will be bound to the remaining contents of the heap cell, that is, to its corresponding frame. The postcondition binds pattern existential variable ?x to the return value, which points to a list with contents rev(A) (the reverse sequence of A, also trivially defined algebraically). For the verification of listReverse, one only needs to refer to the heap cell. All the other cells of the configuration are captured by an implicit universally quantified frame variable and remain unchanged. The loop invariant asserts that the heap contains two lists, one starting at p which contains the part of the sequence that was already reversed (rev(?B)), and one starting at x which contains the rest of the sequence (?C), and the initial sequence A equals (rev(?B) followed by ?C. The remainder of the heap (H) in the invariant happened to stay unchanged.

Similarly, the annotations of listAppend only mention the heap cell, while the rest of the cells in the configuration are again captured by the implicit frame. The precondition states that there are two lists in the heap with contents A and B, while the postcondition asserts that the return value points to a list with contents the concatenation of A and B. The invariant states that p splits the first list into a list segment (with contents $?\mathsf{A}_1$) and a list (with contents $?\mathsf{A}_2$). Heap frame !H subsumes the function frame (H) and the second list, which is not used by the loop.

Function listReadWrite reads an integer sequence from standard input, stores it into a list, then writes it at the standard output and frees the list. To prove its correctness,

```
struct listNode { int val; struct listNode *next; };

struct listNode* listReverse(struct listNode *x)
        pre  ⟨list(x)(A), H⟩heap
        post ⟨list(?x)(rev(A)), H⟩heap ∧ returns ?x
{
  struct listNode *p; struct listNode *y;
  p = 0;
   inv ⟨list(p)(?B), list(x)(?C), H⟩heap ∧ A = rev(?B)@?C
  while(x) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}

struct listNode*
listAppend(struct listNode *x, struct listNode *y)
        pre  ⟨list(x)(A), list(y)(B), H⟩heap
        post ⟨list(?x)(A@B), H⟩heap ∧ returns ?x

{
  struct listNode *p;
  if (x == 0) return y;
  p = x;
        inv ⟨lseg(x, p)(?A1), list(p)(?A2), !H⟩heap
            ∧ A = ?A1@?A2 ∧ ¬(p = 0)
  while (p->next) p = p->next;
  p->next = y ;
  return x;
}

void listReadWrite(int n)
        pre  ⟨A @ In⟩in ⟨Out⟩out ∧ len(A) = n
        post ⟨In⟩in ⟨Out @ rev(A)⟩out

{
  int i;
  struct listNode *x;  struct listNode *y;
  i = 0;  x = 0;
    inv ⟨?B @ In⟩in ⟨list(x)(?A), !H⟩heap
        ∧ i ≤ n ∧ len(?B) = n − i ∧ A = rev(?A) @ ?B

  while (i < n) {
    y = x;
    x = (struct listNode*)
        malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }
  i = 0;
inv ⟨Out @ ?A⟩out ⟨list(x)(?B), !H⟩heap ∧ A = rev(?A @ ?B)

  while (x) {
    y = x->next;
    printf("%d␣",x->val);
    free(x);
    x = y;
  }
}
```

Figure 1: Collection of C functions manipulating linked lists and standard I/O

```
struct treeNode {
  int val;
  struct treeNode *left;
  struct treeNode *right;
};

struct listNode { int val; struct listNode *next; };
struct treeNodeList {
  struct treeNode *val;
  struct treeNodeList *next;
};

struct listNode *treeToList(struct treeNode *root)
        pre  ⟨tree(root)(T), H⟩heap
        post ⟨list(?a)(tree2list(T)), H⟩heap ∧ returns ?a

{
  struct listNode *a;  struct listNode *node;
  struct treeNode *t;
  struct treeNodeList *stack;  struct treeNodeList *x;
  if (root == 0) return 0;
  a = 0;
  stack = (struct treeNodeList *)
          malloc(sizeof(struct treeNodeList));
  stack->val = root; stack->next = 0;
      inv ⟨list{tree}(stack)(?TS), list(a)(?A), H⟩heap
          ∧ tree2list(T) = list{tree}2list(rev(?TS))@?A

  while (stack != 0) {
    x = stack;
    stack = stack->next;
    t = x->val;
    free(x);
    if (t->left != 0) {
      x = (struct treeNodeList *)
          malloc(sizeof(struct treeNodeList));
      x->val = t->left; x->next = stack;
      stack = x;
    }
    if (t->right != 0) {
      x = (struct treeNodeList *)
          malloc(sizeof(struct treeNodeList));
      x->val = t; x->next = stack;
      stack = x;
      x = (struct treeNodeList *)
          malloc(sizeof(struct treeNodeList));
      x->val = t->right; x->next = stack;
      stack = x;
      t->left = t->right = 0;
    } else {
      node = (struct listNode *)
             malloc(sizeof(struct listNode));
      node->val = t->val; node->next = a;
      a = node;
      free(t);
    }
  }
  return a;
}
```

Figure 2: Iterative C program flattening a tree into a list: traverses the tree in infix order and, as it reaches each tree node, it deallocates it and allocates a corresponding list node. The matching logic annotations state the full correctness of this program: the tree is completely deallocated, the resulting list is allocated and contains exactly the same elements in the desired order, and that nothing else changes. MatchC automatically verifies the annotated program above in milliseconds.

one needs the input and the output cells in addition to the heap cell. The precondition states that in the input cell there is a sequence A of length n, while the postcondition states that the sequence has been read and written to the output in reverse order (rev(A)). Note that the heap cell does not appear in either the pre or the post conditions, which means that the function can use the heap, but upon return it must be restored to its initial content. The first loop invariant asserts that x points to a list with contents the reverse sequence of the read elements. The second loop invariant asserts that x points to a list with contents the sequence of the yet to be written elements. Heap frame variable !H matches the initial contents of the heap.

Function treeToList in Figure 2 flattens a binary tree into a list, traversing the tree in infix order. Each node of the initial tree (structure treeNode) has three fields: the value, and two pointers, for the left and the right subtrees. Each node of the final list (structure listNode) has two fields: the value and a pointer to the next node of the list. The program makes use of an auxiliary structure (treeNodeList) to represent a stack of trees. For demonstration purposes (to highlight the invariant capability of our verifier), we prefer an iterative version of this program. We need a stack to keep track of our position in the tree. Initially that stack contains the tree passed as argument (as a pointer). The loop repeatedly pops a tree from the stack, and it either pushes back the left tree, the root, and the right tree onto the stack, or if the right tree is empty it pushes back the left subtree and appends the value in the root node at the beginning of the list of tree elements. As the loop processes the tree, it frees the tree nodes and it allocates the corresponding list nodes.

The precondition states that the parameter root points to a binary tree holding the contents T. The postcondition binds ?a to the return value of the function, which points to a list with contents tree2list(T) (the infix traversal sequence of T, also trivially defined algebraically). In matching logic, list, tree, and tree2list are ordinary operation symbols added to the signature and constrained through. The invariant of our function asserts that the heap contains a stack of trees (represented as a list of trees) with contents ?TS and a list with contents ?A, and that the infix traversal sequence of T, tree2list(T), is equal to the concatenation in reverse order of the infix traversal sequences of the trees in the stack concatenated with the contents of the list. All these operations on trees and lists are axiomatized below.

We next list the axioms added in the mathematical library of MatchC in order to verify the programs above automatically. Our axiom for lists is the FOL formula:

$$\begin{array}{rl} & \langle\langle \mathsf{list}(\mathsf{p},\alpha), \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{config} \wedge \phi \\ \Leftrightarrow & \langle\langle \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{\mathrm{config}} \wedge \langle \mathsf{p} = 0 \wedge \alpha = \mathsf{nil} \wedge \phi\rangle_{\mathrm{form}} \\ \vee & \langle\langle \mathsf{p} \mapsto [?\mathsf{a}, ?\mathsf{q}], \mathsf{list}(?\mathsf{q}, ?\beta), \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{\mathrm{config}} \\ & \wedge\ \alpha = [?\mathsf{a}]@?\beta \wedge \phi \end{array}$$

The above captures the two cases, one in which the list is empty and the other in which it has at least one element. We borrowed the notation $\mathsf{p} \mapsto [?\mathsf{a}, ?\mathsf{q}]$ from separation logic; it stands for two bindings, namely for "$\mathsf{p} \mapsto ?\mathsf{a},\ \mathsf{p}{+}1 \mapsto ?\mathsf{q}$". we also borrowed the notation for lists from OCAML: [?a] is a one-element list and @ concatenates two lists. All the non-existential variables in the axiom above are assumed universally quantified, not free; in other words, the H and C variables in this axiom have nothing to do with the homonymous variables in the program annotations.

The tree pattern is axiomatized similarly ($tree(p, \tau_l, \tau_r)$ is our constructor for trees as mathematical objects):

$$\begin{array}{rl} & \langle\langle \mathsf{tree}(\mathsf{p},\tau), \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{config} \wedge \phi \\ \Leftrightarrow & \langle\langle \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{\mathrm{config}} \wedge \mathsf{p} = 0 \wedge \tau = \mathsf{empty} \wedge \phi \\ \vee & \langle\langle \mathsf{p} \mapsto [?\mathsf{a}, ?\mathsf{l}, ?\mathsf{r}], \mathsf{tree}(?\mathsf{l}, ?\tau_l), \mathsf{tree}(?\mathsf{r}, ?\tau_r), \mathsf{H}\rangle_{\mathrm{heap}}\mathsf{C}\rangle_{\mathrm{config}} \\ & \wedge\ \tau = tree(?\mathsf{a}, ?\tau_l, ?\tau_r) \wedge \phi \end{array}$$

The axiomatization for list{tree} is similar to that of list, but the data field is a pointer to a tree.

The only thing left to show is our axioms for reasoning within the mathematical domains that provided the data stored in the heap patterns above. The equations below are self-explanatory; we only mention that rewrite engines like Maude are quite suitable for handling such axiomatizations:

$$\begin{array}{rl} \mathsf{rev}(\mathsf{nil}) & = \mathsf{nil} \\ \mathsf{rev}([a]) & = [a] \\ \mathsf{rev}(\mathsf{A}_1@\mathsf{A}_2) & = \mathsf{rev}(\mathsf{A}_2)@\mathsf{rev}(\mathsf{A}_1) \\ \mathsf{tree2list}(\mathsf{empty}) & = \mathsf{nil} \\ \mathsf{tree2list}(tree(\mathsf{a}, \tau_l, \tau_r)) & = \mathsf{tree2list}(\tau_l)@[\mathsf{a}]@\mathsf{tree2list}(\tau_r) \\ \mathsf{list}\{\mathsf{tree}\}\mathsf{2list}(\mathsf{nil}) & = \mathsf{nil} \\ \mathsf{list}\{\mathsf{tree}\}\mathsf{2list}([\tau]) & = \mathsf{tree2list}(\tau) \\ \mathsf{list}\{\mathsf{tree}\}\mathsf{2list}(\mathsf{A}_1@\mathsf{A}_2) & = \mathsf{list}\{\mathsf{tree}\}\mathsf{2list}(\mathsf{A}_1)@\mathsf{list}\{\mathsf{tree}\}\mathsf{2list}(\mathsf{A}_2) \end{array}$$

3. CONCLUSIONS AND FUTURE WORK

This paper advanced the idea that the recently introduced matching logic approach can be a viable alternative to existing program verification approaches, avoiding their limitations. For example, notoriously difficult aspects such as heap separation or framing fall as special cases of a general and relatively easy to understand notion of pattern matching. Even though it only took a few months to develop, our current matching logic prototype verifier, MatchC, can automatically verify rather challenging programs, proving not only memory safety but also full data-correctness as well as properties about the I/O (see the MatchC URL for stack properties, too). While the current practical results are encouraging, there is much work to be done, such as: formal proofs of soundness and relative completeness, preferably in a generic manner that applies to all programming languages; proving termination of programs, too (we only considered partial correctness so far); inferring pattern invariants; proving consistency of axiomatic definitions of heap or other configuration sub-patterns.

4. REFERENCES

[1] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[2] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[3] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969.

[5] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symb. Logic*, 5:215–244, 1999.

[6] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[7] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST '10*, volume 6486. LNCS, 2010.

[8] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.