Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 183–198

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

MatchC: A Matching Logic Reachability Verifier Using the $\mathbb{K}$ Framework

Andrei Stefanescu

University of Illinois at Urbana-Champaign
stefane1@illinois.edu

Abstract

This paper presents MatchC, a matching logic reachability verifier using the $\mathbb{K}$ framework. $\mathbb{K}$ is a rewriting-based framework for defining and analyzing programming languages. Matching logic is a logic designed to state and reason about structural properties over arbitrary program configurations. Matching logic reachability is a unifying framework for operational and axiomatic semantics of programing languages. The MatchC verifier (`http://matching-logic.org/`) checks reachability properties of programs written in a deterministic fragment of C and is implemented in the $\mathbb{K}$ framework. This paper discusses the correctness of the implementation of the matching logic reachability proof system in MatchC. The main contributions of this paper are the implementation of the verifier, with emphasis on using $\mathbb{K}$ for program verification, and the evaluation of the tool on a large number of programs, including complex ones, like programs implementing the AVL trees data structure and the Schorr-Waite graph marking algorithm.

Keywords:
Program Verification, Rewriting-Based Semantics, $\mathbb{K}$, Matching Logic, MatchC

1 Introduction

Matching logic [29] is a new logic designed to state and reason about structural properties over program configurations. Syntactically, it introduces a new first-order formula construct, called a *pattern*, which is a configuration term, possibly containing variables. Semantically, its models are actually concrete program configurations, where a configuration satisfies a pattern iff it *matches* it. Matching logic reachability [28,27,26] proposes a language-independent proof system (shown in Figure 1) which proves program specifications directly from the operational semantics of a language. MatchC [31,26] is a matching logic reachability verifier for a deterministic fragment of C implemented in the $\mathbb{K}$ framework [30].

In this paper we discuss the architecture of MatchC, with emphasis on the components implemented in $\mathbb{K}$. The soundness of this verifier is based on the recently proposed deduction system of matching logic reachability. We evaluate MatchC on a large number of programs, including implementations of the most popular sorting algorithms (bubble sort, insertion sort, merge sort and quicksort), of operations on

http://dx.doi.org/10.1016/j.entcs.2014.05.010
1571-0661/© 2014 Published by Elsevier B.V.

tree data structures (binary search tree, AVL tree), and of the Schorr-Waite graph marking algorithm.

So far, $\mathbb{K}$ has been successfully used for giving operational semantics to complex programming languages, including Java 1.4 [12], Verilog [23] and C [11]. MatchC is the first project using $\mathbb{K}$ for symbolic execution and logical reasoning.

Generally, a matching logic specification is a reachability rule between matching logic formulae. The tool accepts specifications in the more restricted format:

$$\langle \mathsf{code}\ \cdots\rangle_{\mathsf{k}} \wedge \pi_l \wedge \psi_l \Rightarrow \langle\cdot\ \cdots\rangle_{\mathsf{k}} \wedge \pi_r \wedge \psi_r$$

where π_l, π_r are basic patterns (symbolic program configurations), and ψ_l, ψ_r are existentially quantified first order logic formulae. The rule captures partial correctness: if the program fragment `code` is executed in a configuration that matches π_l and satisfies ψ_l, and the execution terminates, then the resulting configuration matches π_r and satisfies ψ_r.

Currently, three components of MatchC are implemented in $\mathbb{K}$: the semantics of a C fragment, named KernelC, the matching logic reachability deduction and the matching logic formulae implication.

- *The* KernelC *language* is defined in a straightforward manner.
- The *matching logic reachability deduction* module makes use of the $\mathbb{K}$ modularity in extending configurations. A formula $\pi \wedge \psi$ is represented as a task cell containing a config cell and a form cell. The top configuration is a bag of tasks. To prove a specification correct, the prover symbolically executes the task for the left-hand-side of the rule and at the end checks if the resulting configuration implies the right-hand-side of the rule. The (unmodified) original semantics is extended with rules for executing annotated functions and loops, for applying abstraction axioms, and for splitting the state in the case of `if` with symbolic condition. Although the original semantics is intended for concrete execution, due to the nature of $\mathbb{K}$ rewriting, it works for symbolic execution as well.
- The *matching logic formulae implication* consists of two parts: matching the structure and checking the constraints. Structure matching is implemented in $\mathbb{K}$ as a set of rules that attempt to match corresponding parts of each cell's contents and generate the associated constraints. Context transformation is essential in having a reasonable size implementation. The formulae implication is implemented as search for a proof in a rule side condition, a case that does not occur in any other $\mathbb{K}$ definition.

The presentation assumes that the reader is familiar with $\mathbb{K}$, and in particular with the $\mathbb{K}$ notation. The paper is organized as follows. Section 2 presents background information on $\mathbb{K}$ and matching logic reachability. Section 3 presents a sample program verified by MatchC. Section 4 describes the $\mathbb{K}$-based implementation of the key components of the verifier. Section 5 describes the overall implementation and evaluation of MatchC. Section 6 discusses related work. Finally, Section 7 proposes future work and concludes the paper.

2 $\mathbb{K}$ and Matching Logic Background

2.1 $\mathbb{K}$ Framework

The $\mathbb{K}$ framework [30] (`http://k-framework.org/`), is a rewriting-based formalism for developing and analyzing programming languages. $\mathbb{K}$ enables one to modularly define formal executable semantics to a programming language and then use the respective semantics as an interpreter in order to test it. The $\mathbb{K}$ tools [35] are built on top of the Maude rewriting engine [7]. Currently, the following semantics-based tools are part of the $\mathbb{K}$ framework: interpreter, model-checker, run-time verifier and type-checker. Several languages have been given semantics in $\mathbb{K}$, including Java 1.4 [12], Verilog [23] and C [11].

2.2 Matching Logic

Matching logic [29] is a logic designed to state and reason about structural properties over arbitrary program configurations. Syntactically, it introduces a new formula construct, called a *basic pattern*, which is a configuration term possibly containing variables. Semantically, its models are concrete/ground configurations, where a ground configuration satisfies a basic pattern iff it *matches* it; that is, the variables in the basic pattern can be instantiated with ground terms to obtain the ground configuration. Considering a particular configuration structure with a top-level cell $\langle ... \rangle_{\mathsf{cfg}}$ holding, in any order, other cells with semantic data such as the code $\langle ... \rangle_{\mathsf{k}}$, an environment $\langle ... \rangle_{\mathsf{env}}$, a heap $\langle ... \rangle_{\mathsf{heap}}$, an input buffer $\langle ... \rangle_{\mathsf{in}}$ or an output buffer $\langle ... \rangle_{\mathsf{out}}$, configurations then have the structure:

$$\langle \cdots \ \langle ... \rangle_{\mathsf{k}} \ \langle ... \rangle_{\mathsf{env}} \ \langle ... \rangle_{\mathsf{heap}} \ \langle ... \rangle_{\mathsf{in}} \ \langle ... \rangle_{\mathsf{out}} \ \ \cdots \rangle_{\mathsf{cfg}}$$

The contents of the cells can be various algebraic data types, such as trees, lists, sets, maps, etc. Here are two particular configurations (similar to the $\mathbb{K}$ notation, in the interest of space, we use "..." for the irrelevant parts of them):

$$\langle \cdots \ \langle \texttt{x=*y; y=x;} \ \cdots \rangle_{\mathsf{k}} \ \langle \cdots \ \texttt{x} \mapsto 7, \ \texttt{y} \mapsto 3 \ \cdots \rangle_{\mathsf{env}} \ \langle 3 \mapsto 5 \rangle_{\mathsf{heap}} \ \ \cdots \rangle_{\mathsf{cfg}}$$
$$\langle \cdots \ \langle \texttt{x} \mapsto 3 \rangle_{\mathsf{env}} \ \langle 3 \mapsto 5, \ 2 \mapsto 7 \rangle_{\mathsf{heap}} \ \langle 1, \ 2, \ 3 \ \cdots \rangle_{\mathsf{in}} \ \langle \cdots \ 7, \ 8, \ 9 \rangle_{\mathsf{out}} \ \ \cdots \rangle_{\mathsf{cfg}}$$

Different languages may have different configuration structures. For example, languages whose semantics are intended to be purely syntactic and based on substitution, e.g., λ-calculi, may contain only one cell, holding the program itself. Other languages may contain dozens of cells in their configurations; for example, the C semantics in [11] has more than 75 nested cells. However, no matter how complex a language is, its configurations can be defined as ground terms over an algebraic signature, using conventional algebraic techniques. Matching logic takes an arbitrary algebraic definition of configurations as parameter and allows configuration terms with variables as particular formulae. For example, the formula

$$\begin{aligned}&\exists c\colon Cells,\ e\colon Env,\ p\colon Nat,\ i\colon Int,\ \sigma\colon Heap\\ &\quad\langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i\end{aligned}$$

is satisfied by all configurations where program variable x points to a location p holding a positive integer i different from p. Variables such as e, σ and c above are called *structural frames*. If we want to additionally state that p is the only location allocated, then we can just remove σ:

$$\exists c\colon Cells, e\colon Env, p\colon Nat, i\colon Int\ \ \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

Matching logic allows to reason about configurations, for example, to prove that:

$$\begin{aligned}&\models \forall c\colon Cells,\ e\colon Env,\ p\colon Nat\\ &\qquad\langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto 9\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ p > 10\\ &\qquad\rightarrow \exists i\colon Int,\ \sigma\colon Heap\ \ \langle\langle \mathtt{x} \mapsto p,\ e\rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i\end{aligned}$$

Matching logic formulae of the form $\pi \wedge \psi$ with π a basic pattern and ψ a first-order logic with equality (FOL$_=$) formula with no patterns are called **constrained patterns**, ones of the form $\exists X \pi$ with $X \subset Var$ and π a constrained pattern are called **existential patterns**, and ones of the form $\pi_1 \vee \ldots \vee \pi_n$ with each π_i an existential pattern are called **disjunctive patterns**. We call all the above generically **patterns**.

As shown in [29,31], like separation logic, matching logic can also be used as a program logic in the context of conventional axiomatic semantics, allowing us to more easily specify structural properties about the program state. However, this way of using matching logic comes with a big disadvantage, shared with Hoare logics in general: the formal semantics of the target language needs to be redefined axiomatically and the tedious soundness proofs need to be done. Here, we take the different approach in [28,27,26], which allows us to use the operational semantics of the language for program verification as well, as shown next.

2.3 Matching Logic Reachability

A (matching logic) *reachability rule* is a pair $\varphi \Rightarrow \varphi'$, where φ and φ' are matching logic formulae (not necessarily closed). The semantics of the rule is that any ground configuration satisfying φ transits (in zero or more steps, depending on the context) into a configuration satisfying φ'. A (matching logic) *reachability system* is a set of reachability rules. Such reachability systems subsume the main elements of both operational and axiomatic semantics.

Programming languages can be given operational semantics based on reduction rules of the form "$l \Rightarrow r$ if b", where l and r are configuration terms with variables constrained by boolean condition b. PLT Redex [13] and $\mathbb{K}$ [30] are frameworks for

Rules of operational nature

Reflexivity:

$$\frac{\cdot}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}$$

Axiom:

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Substitution:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \theta : \mathit{Var} \to \mathcal{T}_\Sigma(\mathit{Var})}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}$$

Transitivity:

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}$$

Rules of deductive nature

Case analysis:

$$\frac{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Logic framing:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad \psi \text{ is a FOL}_= \text{ formula}}{\mathcal{A} \vdash \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Consequence:

$$\frac{\models \varphi_1 \to \varphi_1' \quad \mathcal{A} \vdash \varphi_1' \Rightarrow \varphi_2' \quad \models \varphi_2' \to \varphi_2}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2}$$

Abstraction:

$$\frac{\mathcal{A} \vdash \varphi \Rightarrow \varphi' \qquad X \cap \mathit{FreeVars}(\varphi') = \emptyset}{\mathcal{A} \vdash \exists X\ \varphi \Rightarrow \varphi'}$$

Rule for circular behavior

$$\textbf{Circularity:}\ \frac{\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'' \qquad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{A} \vdash \varphi \Rightarrow \varphi'}$$

Fig. 1. Matching logic rewriting proof system

defining such semantics. These rules can be expressed as matching logic reachability rules $l \wedge b \Rightarrow r$. On the other hand, a Hoare triple of the form $\{\psi\}\,\texttt{code}\,\{\psi'\}$ can be regarded as a matching logic reachability rule $\langle \texttt{code} \rangle_{\mathsf{k}} \wedge \psi \Rightarrow \langle \rangle_{\mathsf{k}} \wedge \psi'$ between formulae over minimal configurations holding only the code ($\langle \rangle_{\mathsf{k}}$ is the configuration holding the empty code). Therefore, both operational semantics rules and axiomatic semantics Hoare triples are instances of matching logic rules.

Figure 1 shows the nine-rule language-independent proof system for matching logic reachability. **Reflexivity** and **Transitivity** are inspired by rewriting logic [24]. **Case analysis**, **Logic framing**, **Consequence** and **Abstraction** are inspired by Hoare logic [16]. **Axiom** and **Substitution** by both. The **Circularity** proof rule is new. It deductively and *language-independently* captures the various circular behaviors that appear in languages, due to loops, recursion, jumps, etc. $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ means that the matching logic rule $\varphi \Rightarrow \varphi'$ is derivable from a set of matching logic rules $\mathcal{A}$ using all nine proof rules, while $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$ means that $\varphi \Rightarrow \varphi'$ is derivable from $\mathcal{A}$ using all proof rules but Reflexivity, indicating that at least one proper semantic step is taking place.

A programming language operational semantics is given as a set of reachability rules, which is the initial $\mathcal{A}$. Subsequent uses of **Circularity** enlarge $\mathcal{A}$ with additional reachability rules. Our proof system in Figure 1 can be then used either to generate such concrete, operational program behaviors (the first eight proof rules), or to prove program properties specified as matching logic rules. During the proof derivation, one may add new rules to $\mathcal{A}$ by means of **Circularity**, which are thus allowed to be used in their own derivation. The correctness of this proof circular-

$$\textbf{Set circularity:}\ \frac{\begin{array}{c}\mathcal{C} \text{ is the set } \{\ \varphi_1 \Rightarrow \varphi_1', \dots,\ \varphi_n \Rightarrow \varphi_n'\ \} \\ \mathcal{A} \vdash \{\ \varphi_1 \Rightarrow^+ \varphi_1'', \dots,\ \varphi_n \Rightarrow^+ \varphi_n''\ \} \\ \mathcal{A} \cup \mathcal{C} \vdash \{\ \varphi_1'' \Rightarrow \varphi_1', \dots,\ \varphi_n'' \Rightarrow \varphi_n'\ \}\end{array}}{\mathcal{A} \vdash \mathcal{C}}$$

Fig. 2. Derived circularity rule schema

ity is given by the fact that progress is required to be made (indicated by $\Rightarrow^+$ in $\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'$) before a circular reasoning step is allowed.

For a reachability system $\mathcal{A}$ associated to a deterministic programming language semantics, we define the semantic validity of a rule as follows: $\mathcal{A} \models \varphi \Rightarrow \varphi'$ iff for all concrete terminating configurations γ matching φ, there exists some concrete configuration γ' matching φ'. We have the following result:

Theorem 2.1 (generic partial correctness) *Let $\mathcal{A}$ be a deterministic set of reachability rules, and $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ a sequent derived with the proof system in Figure 1. Then $\mathcal{A} \models \varphi \Rightarrow \varphi'$. (See [28] for details and proof.)*

The **Circularity** proof rule in Figure 1 only allows for circularly deriving one reachability rule at a time. Figure 2 shows the **Set circularity** proof rule schema. It allows to circularly derive several reachability rules at once. Note that it uses sets of reachability rules in the right-hand sides of the sequents; it actually is syntactic sugar for saying that each of the rules is derivable. This is practical in verifying, for example, mutually recursive functions. **Set circularity** does not increase the expressiveness of our proof system. We refer the reader to [32] for more details.

3 A MatchC Verification Example

Here we present a sample program which MatchC verifies. First we briefly discuss some notations MatchC uses for user convenience:

- While all specifications are reachability rules $\varphi \Rightarrow \varphi'$ between matching logic formulae, often φ and φ' share configuration context; we only mention the context once and distribute the "$\Rightarrow$" arrow through the context where the changes take place.
- To avoid writing existential quantifiers, logical variables starting with "?" are assumed existentially quantified.
- To avoid writing environment cells containing only bindings of the form `x` $\mapsto$?x in almost all specifications, we automatically assume them when not explicitly mentioned and allow users to write the identifier `x` (which is a syntactic constant) instead of the logical variable ?x.
- MatchC desugars invariants `inv` φ `loop` into matching logic proof obligation rules φ[`loop`...] $\Rightarrow \varphi$[...] $\wedge \neg$ *cond*(`loop`), where φ[`code`] is the pattern obtained from φ by replacing the contents of the $\langle ... \rangle_{\mathsf{k}}$ cell with `code`.

```
struct listNode { int val; struct listNode *next; };
struct listNode *readList(int n)
 rule ⟨$ ⇒ return ?x;  ···⟩k ⟨A ⇒ ·  ···⟩in ⟨···  · ⇒ list(?x)(A)  ···⟩heap  if n = len(A)
{
  int i; struct listNode *x, *p;
  if (n == 0) return NULL;
  x = (struct listNode*) malloc(sizeof(struct listNode));
  scanf("%d", &(x->val)); x->next = NULL;
  i = 1; p = x;
 inv ⟨?C  ···⟩in ⟨···  lseg(x,p)(?B), p ↦ [?v, NULL]  ···⟩heap ∧ i ≤ n ∧ len(?C) = n − i ∧ A = ?B@[?v]@?C
  while (i < n) {
    p->next = (struct listNode*) malloc(sizeof(struct listNode));
    p = p->next; scanf("%d", &(p->val)); p->next = NULL;
    i += 1;
  }
  return x;
}
```

Fig. 3. C function reading a sequence of integers from the standard input into a singly-linked list.

Function `readList` in Figure 3 reads `n` integers from standard input and stores them in a singly-linked list. The matching logic specification in line 3 says that the function: (1) returns a pointer ?x; (2) reads from the standard input the sequence of integers A of length `n` (matches A and replaces it by the empty sequence ·); (3) allocates a list starting at ?x with contents the read sequence A, represented as the term lseg(`x`, `p`) (replaces the empty heap ·). The rest of the input buffer, the heap and the configuration stay unchanged (··· stands for the *cell frame*). The loop invariant in line 10 states that: the sequence ?C is yet to be read; `x` points to a list segment ending at `p` with contents ?B; `p` points to a `nodeList` structure with the `value` field ?v and the `next` field `NULL`; the loop index `i` is less than or equal to `n`; the length of ?C is `n` − `i`; and the initial sequence A is the concatenation of ?B, [?v] and ?C. The list segment lseg(`x`, `p`) is the list between `x` and `p`, including `x` but excluding `p`. The *operation* symbol len is axiomatized as part of the sequence domain. Like in OCaml, @ concatenates sequences. Variables without ?, like A, are free. Hence, A refers to the same sequence in the function rule and in the loop invariant, while ?B may refer to different sequences in different loop iterations.

4 Using $\mathbb{K}$ for Program Verification

In this section we discuss how to use $\mathbb{K}$ in the context of program verification: Section 4.1 briefly presents the programming language we use for illustration, Section 4.2 describes how to use $\mathbb{K}$ for symbolic execution, Section 4.3 shows how to use $\mathbb{K}$ in checking matching logic formulae implication, and Section 4.4 presents abstraction patterns.

4.1 KernelC

We choose a fragment of C, named KernelC. Its features include

- Expressions: assignment, referencing and dereferencing, structure member (`->`), arithmetic and logic operators, ternary conditional (`_?_:_`), function call

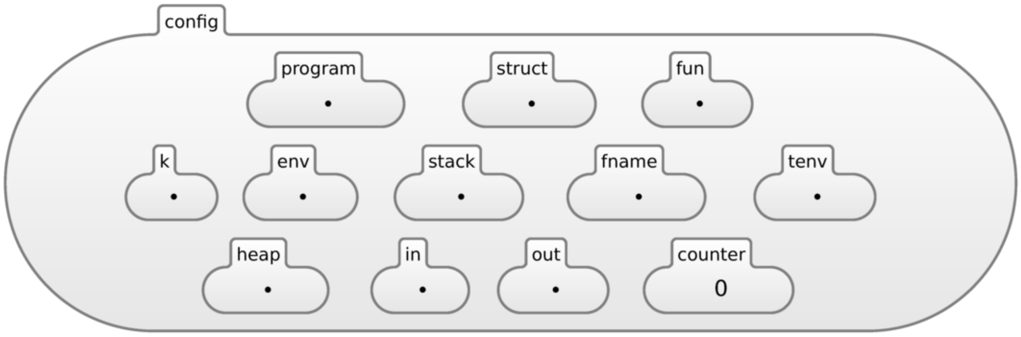

Fig. 4. Configuration of KernelC

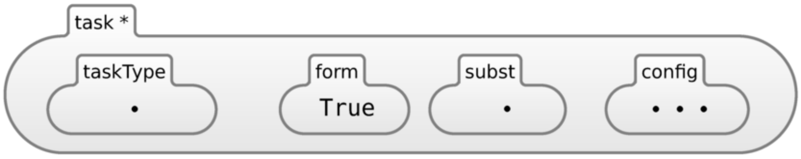

Fig. 5. Symbolic configuration of KernelC

- Statements: semicolon operator (_;), **`while`**, **`if`**, **`if/else`**, **`return`**
- Types: integers, pointers to structures, and pointers to pointers
- Standard Library: `malloc/free`, basic I/O operations

Since our main motivation is to show how to construct a verifier based on a $\mathbb{K}$ definition of a language, we make several simplifying assumptions: the order of expression evaluation is left to right (no non-determinism), integers have infinite precision, variable-size arrays and pointers inside structures are not allowed, the I/O primitives can only read and write integers.

Figure 4 displays the configuration of KernelC. As usually, the $\langle ... \rangle_{\mathsf{k}}$ cell holds the code; $\langle ... \rangle_{\mathsf{env}}$ holds the environment as a mapping from program variables directly into primitive values (integers and addresses); $\langle ... \rangle_{\mathsf{stack}}$ holds the call stack as a list of frames; $\langle ... \rangle_{\mathsf{fname}}$ holds the name of the function currently being executed; $\langle ... \rangle_{\mathsf{tenv}}$ holds the type environment; $\langle ... \rangle_{\mathsf{struct}}$ and $\langle ... \rangle_{\mathsf{fun}}$ hold the structures and the functions declared in the program. The $\langle ... \rangle_{\mathsf{heap}}$ cell contains the dynamically allocated memory as a mapping from addresses to primitive values. The heap entries which are part of structures are labelled with the respective field name. The $\langle ... \rangle_{\mathsf{in}}$ and $\langle ... \rangle_{\mathsf{out}}$ cells hold the input and output buffers as lists of integers. The $\mathbb{K}$ semantic definition of KernelC consists of 41 syntactic constructs, and of 91 semantic rules.

4.2 Symbolic Execution with $\mathbb{K}$

The $\mathbb{K}$ framework is primarily designed for giving concrete executable semantic definitions to programming languages. In this subsection we show how such a definition can be smoothly extended to a symbolic execution definition. Our approach is different from the traditional approaches based on weakest-precondition or strongest-postcondition generation, as explained below.

First, we notice that while concrete semantics are given over ground configurations, symbolic semantics are given over constrained patterns (configurations with variables and constraints). As a consequence, we extend the configuration structure in Figure 4 to the configuration structure in Figure 5. The $\langle ...\rangle_{\mathsf{config}}$ cell holds the basic pattern; $\langle ...\rangle_{\mathsf{form}}$ holds the FOL formula constraining the variables appearing in the basic pattern; $\langle ...\rangle_{\mathsf{subst}}$ holds the substitution history for the free variables along the current execution path; $\langle ...\rangle_{\mathsf{taskType}}$ holds the current status (symbolic execution, reasoning). Since the execution of a constrained pattern could result in a disjunctive pattern, we allow multiple $\langle ...\rangle_{\mathsf{task}}$ cells at the top level. We notice that $\mathbb{K}$ modularity w.r.t. the extension of the configuration structure makes this step possible without affecting the existing rules.

Next, since $\mathbb{K}$ rewriting does not distinguish between constants and variables, most rules work as well in the symbolic case as in the concrete case. However, there are rules that require concrete values to continue the execution. For example, after the first argument of `if/else` is evaluated to an integer, the following rules capture the cases when the concrete integer is zero or non-zero ($NzI \in \mathbb{Z} \setminus \{0\}$):

$$\text{rule}\ \ \texttt{if}(NzI)\ \ S_1\ \ \texttt{else}\ \ S_2 \Rightarrow S_1$$

$$\text{rule}\ \ \texttt{if}(0)\ \ S_1\ \ \texttt{else}\ \ S_2 \Rightarrow S_2$$

However, if the first argument evaluates to a symbolic integer (a term containing at lest one integer variable), the rules above can not apply. As a result, we add the following rule that makes case analysis:

$$\begin{aligned} \text{rule}\ \ & \langle\langle\langle(\texttt{if}(I)\ \ S_1\ \ \texttt{else}\ \ S_2) \curvearrowright K\rangle_{\mathsf{k}}\ C\rangle_{\mathsf{config}}\ \langle\phi\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}} \\ \Rightarrow\ \ & \langle\langle\langle S_1 \curvearrowright K\rangle_{\mathsf{k}}\ C\rangle_{\mathsf{config}}\ \langle\phi \wedge I \neq 0\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}} \\ & \langle\langle\langle S_2 \curvearrowright K\rangle_{\mathsf{k}}\ C\rangle_{\mathsf{config}}\ \langle\phi \wedge I = 0\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}} \end{aligned}$$

A similar case occurs for the function `scanf`, which may require case analysis on whether the symbolic input buffer contains an integer followed by a list or is empty. Another case is memory access, which is discussed in Section 4.4. The addition of these rules is sound according to the **Case Analysis** proof rule in Figure 1.

Finally, in the case of annotated while loops and functions, we use the specification instead of the code. Let $\pi \wedge \psi$ be the current constrained pattern, and let $\pi_l \wedge \psi_l \Rightarrow \pi_r \wedge \psi_r$ be the rule capturing the behaviour of the code. Then we check whether there exists a substitution θ such that $\pi \wedge \psi \to \theta(\pi_l \wedge \psi_l)$ (see Section 4.3). If the answer is affirmative, we transit into the constrained pattern $\theta(\pi_r \wedge \psi_r) \wedge \psi$.

This process is implemented with $\mathbb{K}$ rules. It is sound because it corresponds to the application of **Circularity** (to prove the specification correct), **Axiom**, **Substitution**, **Logic Framing**, and **Consequence** proof rules in Figure 1.

The soundness of one step of symbolic execution is given by **Axiom** and **Substitution** (apply the semantic rule) and **Logic Framing** (propagate the constrains). The soundness of multiple steps is given by **Transitivity**.

4.3 Checking Implication

To enable the application of the **Consequence** proof rule in Figure 1, we need to check implication of matching logic formulae. In practice, given two constrained patterns φ_1 and φ_2, we need to decide whether there exists a substitution θ such that $\varphi_1 \rightarrow \theta(\varphi_2)$. This question is in general undecidable, so our procedure is incomplete. Checking implication consists of two parts: matching the configurations and checking the constraints. We implement the first with $\mathbb{K}$ rules. Each such rule matches and eliminates corresponding subterms in the two configurations, and may generate new constraints. The process ends when all the contents of all the cells are eliminated. We represent the implication with two top-level $\langle ... \rangle_{\mathsf{task}}$ cells (see Figure 5), one for the left-hand-side and one for the right-hand-side. For example, the following rule eliminates the entry for address X from both heaps and adds the constraint that the values stored at address X must be equal:

$$\text{rule } \langle \cdots \ \langle \mathit{Hypothesis} \rangle_{\mathsf{taskType}} \langle \cdots \ X \mapsto V_1 \Rightarrow \cdot \ \cdots \rangle_{\mathsf{heap}} \ \cdots \rangle_{\mathsf{task}}$$
$$\langle \cdots \ \langle \mathit{Conclusion} \rangle_{\mathsf{taskType}} \langle \cdots \ X \mapsto V_2 \Rightarrow \cdot \ \cdots \rangle_{\mathsf{heap}} \langle \phi \Rightarrow (\phi \wedge V_1 = V_2) \rangle_{\mathsf{form}} \ \cdots \rangle_{\mathsf{task}}$$

4.4 Abstraction Patterns

In order to express properties and reason about arbitrary cell contents, we need an abstraction mechanism. There is a large body of literature on heap abstraction (for example, see [33,25]). Our approach is different in that it applies abstraction at the term level rather than at the predicate level. For example, a heap abstraction pattern $\mathsf{heap\_abs}(x_1, \ldots, x_n)(\alpha_1, \ldots, \alpha_m)$ is a term representing the portion of the heap delimited in some way by the pointers $x_1, \ldots, x_n$ and storing the elements of mathematical domains $\alpha_1, \ldots, \alpha_m$ (can be integers, sequences, sets, . . .). Similar abstractions can be defined for any cell in the configuration. An abstraction pattern is not defined, it is axiomatized. For that reason, one would have to check that all the axioms are consistent; currently, we do not perform this check in MatchC.

For clarity, we present below the list heap abstraction pattern for a singly-linked list which is part of the library of MatchC:

$$\begin{aligned}
& \langle \langle \mathsf{list}(p)(\alpha), \sigma \rangle_{\mathsf{heap}} \ c \rangle_{\mathsf{config}} \\
\leftrightarrow \ & \langle \langle \sigma \rangle_{\mathsf{heap}} \ c \rangle_{\mathsf{config}} \ \wedge p = 0 \wedge \alpha = [] \\
\vee \ & \exists a, q, \beta \ (\langle \langle p \mapsto [a, q], \mathsf{list}(q)(\beta), \sigma \rangle_{\mathsf{heap}} \ c \rangle_{\mathsf{config}} \wedge \alpha = [a] @ \beta)
\end{aligned}$$

It abstracts heap subterms into list terms and captures two cases, one in which the list is empty and the other in which the list has at least one element. We use p to denote the pointer that is the head of list and α the sequence of integers stored in the list. The notation $p \mapsto [a, q]$ stands for a heap subterm with two locations, namely "$p \mapsto a,\ p+1 \mapsto q$".

We generate $\mathbb{K}$ rules based on this axiom. The first rule applies the axiom from left to right and generates more concrete heaps from more abstract ones

$$\begin{aligned}
&\text{rule } \langle\langle \mathit{Run}\rangle_{\mathsf{taskType}}\langle\langle \mathtt{access}(P') \curvearrowright K\rangle_{\mathsf{k}}\langle \mathsf{list}(P)(\alpha),\ H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{config}}\ \langle\phi\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}}\\
&\quad\Rightarrow \langle\langle \mathit{Run}\rangle_{\mathsf{taskType}}\langle\langle \mathtt{access}(P') \curvearrowright K\rangle_{\mathsf{k}}\langle H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{config}}\ \langle\phi \wedge P = 0 \wedge \alpha = []\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}}\\
&\qquad \langle\langle \mathit{Run}\rangle_{\mathsf{taskType}}\langle\langle \mathtt{K}\rangle_{\mathsf{k}}\langle P \mapsto [A, Q],\ \mathsf{list}(Q)(\beta),\ H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{config}}\ \langle\phi \wedge \alpha = [A]@\beta\rangle_{\mathsf{form}}\ T\rangle_{\mathsf{task}}\\
&\quad \text{if } \phi \rightarrow P' = P \vee P' = P + 1
\end{aligned}$$

where A, Q and α are fresh variables (implemented with a counter which is not shown here for brevity). The **Abstraction** proof rule allows us to make A, Q and α free instead of existential. Notice the presence of the $\langle \mathit{Run}\rangle_{\mathsf{taskType}}$ cell; it means this rule can only be applied during symbolic execution, and not while checking an implication. To prevent the infinite application of the axiom, the rule uses a memory access on the head of the list as a trigger. The following two rules cover the two cases when orienting the axiom from right to left:

$$\begin{aligned}
&\text{rule}\quad \langle \mathit{Hypothesis}\rangle_{\mathsf{taskType}}\langle\cdots\ \ P \mapsto [A, 0] \Rightarrow \mathsf{list}(P)([A])\ \ \cdots\rangle_{\mathsf{heap}}\\
&\text{rule}\quad \langle \mathit{Hypothesis}\rangle_{\mathsf{taskType}}\langle\cdots\ \ P \mapsto [A, Q],\ \mathsf{list}(Q)(\beta) \Rightarrow \mathsf{list}(P)([A]@\beta)\ \ \cdots\rangle_{\mathsf{heap}}
\end{aligned}$$

For efficiency, these rules only apply during the implication checking on the hypothesis. There are similar rules for the conclusion.

Currently, MatchC has axioms for the following heap abstraction patterns: single-linked and doubly-linked lists, single-linked and doubly-linked list segments, queues, binary trees and points-to graphs. Also, it has axioms for a call stack abstraction pattern. The rules are manually generated from the axioms.

5 Implementation and Evaluation

Here we discuss the implementation and evaluation of the MatchC verifier.

The front end of MatchC is implemented in Java and Python. The mathematical domains, as well as checking constraints over them (as part of checking pattern implication), are specified in Maude [7]. Currently, the library of MatchC contains the following domains: integers, sequences, trees, sets, multisets and graphs. If a constraint does not simplify to true, it is passed to SMT solvers CVC3 [2] and Z3 [9].

The user provides a set of program properties using the notation in Section 3 (properties that one wants to verify). The tool only supports annotations for functions and while loops. Let $\mathcal{S}$ be the $\mathbb{K}$ semantic definition of KernelC, and let $\mathcal{C}$ be the set of specifications (both given as sets of reachability rules). Then MatchC derives the sequent $\mathcal{S} \vdash \mathcal{C}$ using the proof system in Figures 1 and 2. Note that $\mathcal{C}$

Program	Cells	Time(s)	C4	C5
1. Undefined programs				
uninitialized memory	—	0.01	1	no
division by zero	—	0.01	1	no
uninitialized variable	—	0.01	1	no
unallocated location	—	0.01	1	no
2. Simple programs that need only the environment cell				
average	—	0.02	1	no
min	—	0.04	2	no
max	—	0.04	2	no
mul by add	—	0.13	3	yes
sum (recursive)	—	0.06	2	yes
sum (iterative)	—	0.08	2	yes
assoc comm	—	0.03	1	no
3. Lists				
list head	heap	0.02	2	no
list tail	heap	0.02	1	no
list add	heap	0.02	1	no
list swap	heap	0.03	3	no
list deallocate	heap	0.04	2	no
list length (recursive)	heap	0.05	2	no
list length (iterative)	heap	0.07	2	no
list sum (recursive)	heap	0.05	2	no
list sum (iterative)	heap	0.07	2	no
list reverse	heap	0.06	2	no
list append	heap	0.1	3	no
list copy	heap	0.13	3	no
list filter	heap	0.22	5	no
4. Input and output				
read write	in, out	0.12	4	no
list read	in, heap	0.14	7	no
list write	heap, out	0.06	2	no
list read write	heap, in, out	0.15	5	no

Program	Cells	Time (s)	C4	C5
5. Trees				
tree height	heap	0.1	4	no
tree size	heap	0.07	3	no
tree find	heap	0.12	5	no
tree mirror	heap	0.7	3	no
tree in-order	heap	0.7	3	no
tree pre-order	heap	0.7	3	no
tree post-order	heap	0.7	3	no
tree deallocate	heap	0.14	7	no
tree to list (recursive)	heap, out	0.1	4	no
tree to list (iterative)	heap, out	0.24	11	no
6. Call stack				
only g calls f	call stack	0.04	2	no
h in stack when f	call stack	0.04	2	no
stack inspection	call stack	0.24	8	no
7. Sorting algorithms				
insert	heap	0.35	5	no
insertion sort	heap	0.41	6	no
bubble sort	heap	0.30	6	no
quicksort	heap	0.47	8	no
merge sort	heap	1.97	16	yes
8. Search trees				
BST find	heap	0.15	5	yes
BST insert	heap	0.13	4	yes
BST delete	heap	0.38	10	yes
AVL find	heap	0.15	5	yes
AVL insert	heap	43.5	23	yes
AVL delete	heap	133.58	36	yes
9. Schorr-Waite				
tree Schorr Waite	heap	0.28	6	no
graph Schorr Waite	heap	1.73	8	no

C4=# paths (number of paths)

C5=SMT? (need for SMT support)

Fig. 6. Results of MatchC program verification

contains one candidate rule for each function and one candidate rule for each loop. Currently, we require all the rules in $\mathcal{C}$ to be of the form $\langle \mathtt{code}\ \ \cdots\rangle_{\mathsf{k}} \wedge \pi \Rightarrow \langle\ \ \cdots\rangle_{\mathsf{k}} \wedge \pi'$, with π and π' patterns.

To derive the sequent $\mathcal{S} \ \vdash \ \mathcal{C}$, MatchC begins by applying **Set Circularity** for $\mathcal{C}$ and reduces the task to deriving individual sequents of the form $\mathcal{S} \cup \mathcal{C} \ \vdash \ \pi \Rightarrow \pi'$, with π and π' patterns. To prove each such rule, the tool rewrites π to π'' using rules in $\mathcal{S} \cup \mathcal{C}$ as described in Section 4.2, with π'' such that the code of π'' and π' is the same. The verification fails if the execution "gets stuck" before reaching such a π''. Then MatchC checks the implication $\pi'' \rightarrow \pi'$. The verification succeeds if the check succeeds and fails otherwise. Notice that MatchC is sound but incomplete w.r.t. the proof system in Figure 1. As an optimisation, when a pattern can be rewritten with rules from both $\mathcal{S}$ and $\mathcal{C}$, the verifier only uses the rules from $\mathcal{C}$. In particular, only a loop without a specification is unrolled, and only a function without a specification is called. Also, if the current pattern implies that the application of an abstraction axiom would result into a more concrete pattern, the tool applies the respective axiom (for instance, knowing the head of a linked list is not null results in an automatic list unrolling).

Figure 6 summarises the results of our experiments. Two factors guided us in our selection: (1) proving functional correctness, not just memory safety; and (2) doing so automatically, requiring only the user provided specifications.

Brief explanations on the examples follow.

1. Undefined programs: MatchC detects undefined behaviour, like read of uninitialized memory, division by zero, read of an uninitialized variable or access of unallocated memory.

2. Simple programs: MatchC verifies several programs performing basic arithmetic operations.

3 & 5. Lists and trees: MatchC proves the full correctness of various list and tree manipulating programs. For each function, it checks the expected memory safety and heap shape properties, and also the functional behaviour, which is algebraically axiomatized.

4. I/O: MatchC verifies several programs manipulating I/O. Similar to the `readList` example in Figure 3, for each function, it proves that the content of the heap, as well as the I/O buffers, is as expected.

6. Call stack: MatchC certifies several function call policies, like only the function `g` calls `f` or `h` must be in the call stack when `f` is called.

7. Sorting algorithms: MatchC verifies the most common sorting algorithms. For each sorting function, it proves that the returned sequence is indeed sorted and it consists of exactly the same elements as the original sequence.

5. Trees: MatchC proves the full correctness of binary search tree and AVL tree data structures. For each function, the tool checks that it maintains the data structure invariant and that the multiset of elements is as expected. We mention that the AVL insert and delete programs take approximately 3 minutes together because some of the auxiliary functions (like balance) are not given specifications and thus their bodies are being executed, resulting in a larger number of paths to analyze.

9. Schorr-Waite: The Schorr-Waite graph marking algorithm [34] computes all the nodes in a graph that are reachable from a set of starting nodes. To achieve that, it visits the graph nodes in depth-first search order, by reversing pointers on the way down, and then restoring them on the way up. Its main application is in garbage collection. The Schorr-Waite algorithm presents considerable verification challenges [17,21]. We analyzed the algorithm itself, and a simplified version in which the graph is in fact a tree. For both cases we proved that a node is marked if and only if it is reachable from the set of initial nodes, and that the set of nodes does not change.

Most of these examples are proved in milliseconds and do not require SMT support. The experiments were conducted on a quad-core, 2.2GHz, 4GB machine running Linux. The source code of MatchC, as well as an online interface allowing one to verify and experiment with all C programs discussed here, or to introduce new ones, is publicly available on the matching logic web page at `http://matching-logic.org/`.

6 Related Work

In this section we discuss several of the existing verification tools and frameworks.

Bedrock [6] is a framework which uses computational higher-order separation logic and supports mostly-automated proofs about low-level programs. Unlike MatchC, Bedrock requires the user to annotate the source code with hints for lemma applications (like list rolling and unrolling). Specifications use operators defined in a pure functional language, similarly to the operators defined algebraically in matching logic. It is likely that the tactics employed by Bedrock could be adapted for higher-order matching logic.

Shape analysis [33] allows one to examine and verify properties of heap structures. It has been shown to be quite powerful when reasoning about heaps. The ideas of shape analysis have also been combined with those of separation logic [10] to quickly infer invariants for programs operating on lists. They can likely be also combined with matching logic in order to infer patterns.

There are many Hoare-logic-based verification frameworks, such as ESC/-Java [15], VCC [8], Spec# [1], HAVOC [19] and Dafny [20]. Frama-C/Why [14,17] proved many properties related to the Schorr-Waite algorithm. However, their proofs were not entirely automated. The weakness of traditional Hoare-like approaches is that reasoning about non-inductively defined data-types and about heap structures tend to be difficult, requiring extensive manual intervention in the proof process. Jahob [36] is another verification framework that mixes automated and interactive reasoning. Among the separation-logic-based tools, we mention SLAyer [4], Xisa [5] and Thor [22], which automatically check memory safety, shape and/or arithmetic properties, and Smallfoot [3], Hip [25] and Verifast [18], which can prove functional correctness.

7 Conclusion and Future Work

In this paper we have presented MatchC, a matching logic verifier for a deterministic fragment of C based on the $\mathbb{K}$ framework. We have described the implementation of the key components of the verifier, and have argued their soundness based on the matching logic reachability proof system. We have evaluated MatchC on a large number of programs, some quite challenging from a verification point of view.

We choose KernelC as the case study programing language as it is simpler than full C, yet complex enough for verification purposes. While building this prototype, the emphasis has been on quick development, expressiveness and efficiency.

We plan the develop a language-parametric matching logic verification system.

In such a system, one would plug in a $\mathbb{K}$ semantic definition of a programming language to obtain a verifier. One would only have to extend the front-end with language specific-syntactic sugar, to specify the syntactic constructs that require case analysis, and to axiomatize the language-specific abstraction patterns and mathematical domains. We also want to allow specification around any fragment of code, not just functions and loops.

References

[1] Barnett, M., M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte and H. Venter, *Specification and verification: the Spec# experience*, Commun. ACM **54** (2011), pp. 81–91.

[2] Barrett, C. and C. Tinelli, *CVC3*, in: *CAV*, 2007, pp. 298–302.

[3] Berdine, J., C. Calcagno and P. W. O'Hearn, *Smallfoot: Modular automatic assertion checking with separation logic*, in: *FMCO*, 2005, pp. 115–137.

[4] Berdine, J., B. Cook and S. Ishtiaq, *Slayer: Memory safety for systems-level code*, in: *CAV*, 2011, pp. 178–183.

[5] Chang, B.-Y. E. and X. Rival, *Relational inductive shape analysis*, in: *POPL*, 2008, pp. 247–260.

[6] Chlipala, A., *Mostly-automated verification of low-level programs in computational separation logic*, in: *PLDI*, 2011, pp. 234–245.

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

[8] Cohen, E., M. Moskal, W. Schulte and S. Tobies, *A practical verification methodology for concurrent programs*, Technical Report MSR-TR-2009-15, Microsoft Research (2009).

[9] de Moura, L. M. and N. Bjørner, *Z3: An efficient SMT solver*, in: *TACAS*, 2008, pp. 337–340.

[10] Distefano, D., P. W. O'Hearn and H. Yang, *A local shape analysis based on separation logic*, in: *TACAS*, 2006, pp. 287–302.

[11] Ellison, C. and G. Roşu, *An executable formal semantics of C with applications*, in: *POPL*, 2012, pp. 533–544.

[12] Farzan, A., F. Chen, J. Meseguer and G. Roşu, *Formal analysis of Java programs in JavaFAN*, in: *CAV'04*, LNCS **3114**, 2004, pp. 501–505.

[13] Felleisen, M., R. B. Findler and M. Flatt, "Semantics Engineering with PLT Redex," MIT Press, 2009, I-XII, 1-502 pp.

[14] Filliâtre, J.-C. and C. Marché, *The Why/Krakatoa/Caduceus platform for deductive program verification*, in: *CAV*, 2007, pp. 173–177.

[15] Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, *Extended static checking for Java*, in: *PLDI*, 2002, pp. 234–245.

[16] Hoare, C. A. R., *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), pp. 576–580.

[17] Hubert, T. and C. Marché, *A case study of C source code verification: the Schorr-Waite algorithm*, in: *SEFM*, 2005, pp. 190–199.

[18] Jacobs, B., J. Smans, P. Philippaerts, F. Vogels, W. Penninckx and F. Piessens, *Verifast: A powerful, sound, predictable, fast verifier for c and java*, in: *NASA Formal Methods*, 2011, pp. 41–55.

[19] Lahiri, S. K. and S. Qadeer, *Verifying properties of well-founded linked lists*, in: *POPL*, 2006, pp. 115–126.

[20] Leino, K. R. M., *Dafny: An automatic program verifier for functional correctness*, in: *LPAR*, 2010, pp. 348–370.

[21] Loginov, A., T. W. Reps and M. Sagiv, *Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm*, in: *SAS*, 2006.

[22] Magill, S., M.-H. Tsai, P. Lee and Y.-K. Tsay, *Automatic numeric abstractions for heap-manipulating programs*, in: *POPL*, 2010, pp. 211–222.

[23] Meredith, P. O., M. Katelman, J. Meseguer and G. Roşu, *A formal executable semantics of Verilog*, in: *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)* (2010), pp. 179–188.

[24] Meseguer, J., *Conditional rewriting logic as a united model of concurrency*, Theor. Comput. Sci. **96** (1992), pp. 73–155.

[25] Nguyen, H. H., C. David, S. Qin and W.-N. Chin, *Automated verification of shape and size properties via separation logic*, in: *VMCAI*, 2007, pp. 251–266.

[26] Roşu, G. and A. Ştefănescu, *Checking reachability using matching logic*, in: *OOPSLA* (2012), pp. 555–574.

[27] Roşu, G. and A. Ştefănescu, *From hoare logic to matching logic reachability*, in: *FM'12*, LNCS **7436** (2012), pp. 387–402.

[28] Roşu, G. and A. Ştefănescu, *Towards a unified theory of operational and axiomatic semantics*, in: *ICALP'12*, LNCS **7392** (2012), pp. 351–363.

[29] Rosu, G., C. Ellison and W. Schulte, *Matching logic: An alternative to Hoare/Floyd logic*, in: *AMAST*, 2010, pp. 142–162.

[30] Rosu, G. and T.-F. Serbanuta, *An overview of the K semantic framework*, J. Log. Algebr. Program. **79** (2010), pp. 397–434.

[31] Rosu, G. and A. Stefanescu, *Matching logic: a new program verification approach (NIER track)*, in: *ICSE*, 2011, pp. 868–871.

[32] Rosu, G. and A. Stefanescu, *Matching logic rewriting: Unifying operational and axiomatic semantics in a practical and generic framework*, Technical Report http://hdl.handle.net/2142/28357, University of Illinois (2011).

[33] Sagiv, S., T. W. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM Trans. Prog. Lang. Syst. **24** (2002), pp. 217–298.

[34] Schorr, H. and W. M. Waite, *An efficient machine-independent procedure for garbage collection in various list structures*, Commun. ACM **10** (1967), pp. 501–506.

[35] Şerbănuţă, T. F. and G. Roşu, *K-Maude: A rewriting based tool for semantics of programming languages*, in: *8th International Workshop on Rewriting Logic and its Applications (WRLA'09)*, LNCS **6381**, 2010, pp. 104–122.

[36] Zee, K., V. Kuncak and M. C. Rinard, *An integrated proof language for imperative programs*, in: *PLDI*, 2009, pp. 338–351.