From Rewriting Logic, to Programming Language Semantics, to Program Verification

Grigore Roșu

University of Illinois at Urbana-Champaign, USA
grosu@illinois.edu

Abstract. Rewriting logic has proven to be an excellent formalism to define executable semantics of programming languages, concurrent or not, and then to derive formal analysis tools for the defined languages with very little effort, such as model checkers. In this paper we give an overview of recent results obtained in the context of the rewriting logic semantics framework $\mathbb{K}$, such as complete semantics of large programming languages like C, Java, JavaScript, Python, and deductive program verification techniques that allow us to verify programs in these languages using a common verification infrastructure.

1 Introduction

Programming language semantics and program analysis and verification are well developed research areas with a long history. In fact, one might think that all problems would have been solved by now: we would hope that any formal semantics for a language should give rise to a proof system and that a verifier for such a system would simply extend the proof system with a proof strategy; or looked at from the other side, we would assume that any verification system for a particular programming language would be grounded in that language's formal semantics. However, reality tells us that most program verifiers are not directly based on a formal semantics, but rather on complex and adhoc hardwired models of their target programming languages. This has at least two negative consequences. First, it makes the development and maintenance of program verifiers hard and uneconomical, particularly for new programming languages or languages which evolve fast. Second, it allows room for subtle bugs in program verifiers themselves. Consider, for example, the following three-line C program:

```
void main() {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

When compiled with some compilers, e.g., GCC3, ICC, Clang, this program evaluates to 3, while when compiled with others, e.g., GCC4, MSVC, it evaluates to 4. This is correct behavior for the compilers, because according to the ISO C11 standard [1] this function is *undefined* (it writes x twice within the same sequence-point interval), and language implementations or compilers are free to treat undefined programs however they find fit, in particular to apply aggressive optimizations, like above. The overall design of the C language has been conceived in the spirit of improved performance,

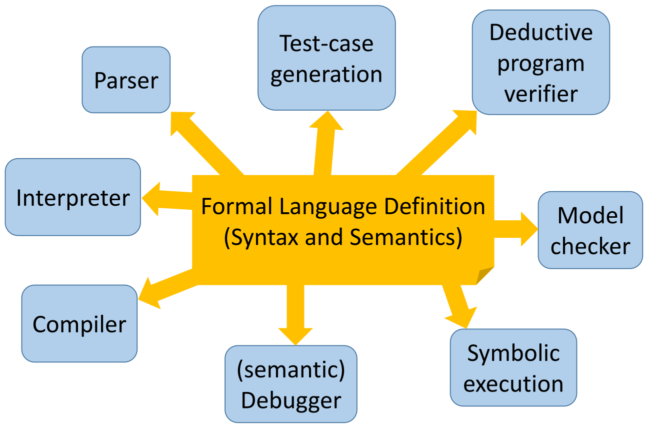

Fig. 1. Rewriting logic semantics can achieve this.

the price to pay being that it becomes programmer's responsibility to ensure that their programs are well-defined. However, what is scary, and in our view unacceptable, is that state-of-the-art program verifiers for C code like VCC [2] and Frama-C [3] *prove* that this program evaluates to 4! That is because in order to simplify implementation, these verifiers use code simplification modules similar to those used by compilers, which work as expected only when the source program is well-defined according to the C semantics.

The unfortunate consequence is that, in spite of more than 40 years of worldwide research and in spite of being hard to use, state-of-the-art software verification tools cannot be trusted. They encode or implement rather adhoc models of programming languages, without any guarantee that their model is faithful to the actual language. The root problem is that most languages *do not even have a formal semantics*, their designers wrongly thinking that a formal semantics is not worth the effort. Hence, the tool developers must rely on informal manuals and on their subjective understanding of the language. Inspired by countless hours of discussions with my mentor, colleague and dear friend José Meseguer, we firmly believe that this can and should change, that *programming languages must have formal semantics!* Moreover, that formal analysis tools and language implementations can and should be derived from such semantics, as shown in Figure 1, so they are correct by construction. This is not a dream. Not anymore.

We present a snapshot of recent and current research on using rewriting logic semantics in the field of programming languages through the $\mathbb{K}$ framework, from giving semantics to real programming languages to using such semantics to verify programs.

2 From Rewriting Logic to Language Semantics

Starting with Meseguer's seminal rewriting logic paper [4], which demonstrated how naturally rewriting logic can capture the various computational paradigms, a series of papers have been published on using rewriting logic to give semantics to programming languages. Verdejo and Martì-Oliet [5,6] show how to use rewriting logic and Maude [7] to define and implement executable semantics for several languages following both big-step [8] and small-step SOS [9,10] approaches, and Șerbănuță *et al.* [11] show how several other semantic approaches can be represented in rewriting logic, including Berry and Boudol's chemical abstract machine (CHAM) [12,13] and Felleisen's reduction semantics with evaluation contests [14,15].

When representing any of these semantic approaches in rewriting logic, the idea is to define the desired program configurations as an algebraic specification, that is as a signature representing the syntax of configurations and equations defining the underlying mathematical domains, and then to define the various types of transitions uniformly as rewrite rules. For example, in a simple C-like imperative language a program configuration can be a pair `< code, state >`, where `code` is a fragment of program and `state` is a finite-domain map from program variables to values. Fragments of programs are nothing but well-formed terms over an appropriate algebraic signature, and finite-domain maps can be defined as an algebraic data type. In a small-step SOS style rewrite logic semantics, for example, the semantics of the assignment construct can then be defined with rewrite rules as follows (we use the Maude notation):

```
crl < X = E,Sigma > => < X = E',Sigma' > if < E,Sigma > => < E',Sigma' > .
 rl < X = V,Sigma > => < V,Sigma[V / X] > .
```

The first rule reduces the expression `E` assigned to program variable `X` step by step until it becomes a value `V`, and then the second rule assigns that value to `X` in the state `Sigma`.

A problem faced when attempting the above for real languages, like C or Java, is that the program configuration tends to be huge, comprising dozens of semantic cells, most of them being unused in most rules or their changes just being propagated by rules. The lack of modularity of SOS was visible even in Plotkin's original notes [9,10], where he had to modify the definition of simple arithmetic expressions several times as his initial language evolved. Hennessy also makes it even more visible in his book [16]. Each time he adds a new feature, he also has to change the configurations and the entire existing semantics. However, the lack of modularity of language definitional frameworks was not perceived as a major problem until late 1990s, partly because there were few attempts to give complete and rigorous semantics to real programming languages. Hennessy actually used each language extension as a pedagogical opportunity to teach what new semantic components the feature needs and how and where those are located in each sequent.

The first to pinpoint the limitations of plain SOS when defining non-trivial languages were the inventors of alternative semantic frameworks, such as Berry and Boudol [12,13] who proposed the chemical abstract machine model, Felleisen and his collaborators [14,15] who proposed reduction semantics with evaluation contexts, and Mosses and his collaborators [17–19] who proposed the modular SOS (MSOS) approach. Among these, Mosses is perhaps the one who most vehemently criticized the lack of modularity of plain SOS. Meseguer and Braga [20,21] were the first to observe that rewriting logic, through its powerful support for multiset matching, can seamlessly support modular semantic frameworks, by giving a first rewrite logic representation of MSOS. The representation in [20] also led to the development of the Maude MSOS tool [22].

Learning from previous uses of rewriting logic to define programming language semantics, we proposed the $\mathbb{K}$ framework [23] (`http://kframework.org`) as a formalism and notation inspired from rewrite logic but specialized to the domain of programming languages. In 𝕂, programming languages can be defined using configurations, computations and rules. Configurations organize the state in units called cells, which are labeled and can be nested. Computations carry computational meaning as special nested list structures sequentializing computational tasks, such as fragments of program. Computations extend the original language abstract syntax. 𝕂 (rewrite) rules make it explicit which parts of the term they read-only, write-only, read-write, or do not care about. This makes 𝕂 suitable for defining truly concurrent languages even in the presence of sharing. Computations are like any other terms in a rewriting environment: they can be matched, moved from one place to another, modified, or deleted. This makes 𝕂 suitable for defining control-intensive features such as abrupt termination, exceptions or call/cc.

In 𝕂, a language is defined in one or more files with extension ".k". A language definition consists roughly of three parts: annotated syntax, configuration, and semantic rules. For syntax, 𝕂 uses conventional BNF annotated with 𝕂-specific attributes. For example, the syntax of assignment in a language like above can be defined as

```
syntax Stmt ::= Id "=" Exp   [strict(2)]
```

The attribute `strict(2)` states the evaluation strategy of the assignment construct: first evaluate the second argument, and then apply the semantic rule(s) for assignment.

To allow arbitrarily complex and nested program configurations, 𝕂 proposes a cell-based approach. Each cell encapsulates relevant information for the semantics, including other cells that can "float" inside it. For our simple language, a "top" cell `<T>...</T>` containing a code cell `<k>...</k>` and a state `<state>...</state>` suffices:

```
configuration <T>
                <k> $PGM </k>
                <state> .Map </state>
              </T>
```

The given cell contents tell 𝕂 how to initialize the configuration: `$PGM` says where to put the input program once parsed, and `.Map` is the empty map.

Once the syntax and configuration are defined, we can start adding semantic rules. 𝕂 rules are contextual: they mention a configuration context in which they apply, together with local changes they make to that context. The user typically only mentions the absolutely necessary context in their rules; the remaining details are filled in automatically by the tool. For example, here is the 𝕂 rule for assignment:

```
rule <k> X:Id = V:Val => V ...</k>
     <state>... X |-> (_ => V) ...</state>
```

The ellipses are part of the 𝕂 syntax. Recall that assignment was `strict(2)`, so we can assume that its second argument is a value, say `V`. The context of this rule involves two cells, the `k` cell which holds the current code and the `state` cell which holds the current state. Moreover, from each cell, we only need certain pieces of information: from the `k` cell we only need the first task, which is the assignment "`X = V`", and from the `state` cell we only need the binding "`X |-> _`". The underscore stands for an anonymous variable, the intuition here being that that value is discarded anyway, so there is no need to bother naming it. The irrelevant parts of the cells are replaced with ellipses. Then, once the local context is established, we identify the parts of the context which need to change, and we apply the changes using local rewrite rules with the arrow =>, noting that it has a greedy scoping, grabbing everything to the left and everything to the right until a cell boundary (open or closed) or an unbalanced parenthesis is encountered. In our case, we rewrite both the assignment expression and the value of `X` in the state to the assigned value `V`. Everything else stays unchanged. The concurrent semantics of 𝕂 regards each rule as a transaction: all changes in a rule happen concurrently; moreover, rules themselves apply concurrently, provided their changes do not overlap.

Once the definition is complete and saved in a .k file, say `imp.k`, the next step is to generate the desired language model. This is done with the `kompile` command:

```
kompile imp.k
```

By default, the fastest possible executable model is generated. To generate models which are amenable for symbolic execution, test-case generation, search, model checking, or deductive verification, one needs to provide `kompile` with appropriate options.

The generated language model is employed on a given program for the various types of analyses using the `krun` command. By default, with the default language model, `krun` simply runs the program. For example, if `sum.imp` contains

```
n=100; s=0;
while(n>0) {
  s=s+n; n=n-1;
}
```

then the command

```
krun sum.imp
```

yields the final configuration

```
<T>
  <k> . </k>
  <state>
    n |-> 0, s |-> 5050
  </state>
</T>
```

Using the appropriate options to the `kompile` and `krun` commands, we can enable all the above-mentioned tools and analyses on the defined programming language and the given program. Many languages are provided with the 𝕂 tool distribution, and several others are available from `http://kframework.org` (start with the 𝕂 tutorial). Some of these languages have dozens of cells in their configurations and hundreds of rules.

Besides didactic and prototypical languages, 𝕂 has been used to formalize several existing real-life languages and to design and develop analysis and verification tools for them. The most notable are complete 𝕂 definitions for the following languages: C11 (POPL'12 [24], PLDI'15 [25]), Java 1.4 (POPL'15 [26]), JavaScript ES5 (PLDI'15 [27]). Each of these language semantics has more than 1,000 semantic rules and has been tested on benchmarks and test suites that implementations of these languages also use to test their conformance, where available. The C semantics, when executed, catches undefinedness; for example, the program discussed in the introduction reports the following error when executed with the C semantics:

```
=============================================================
ERROR! KCC encountered an error while executing this program.
=============================================================
Error: EIO1
Description: Unsequenced side effect on scalar object with side effect
  of same object.
Type: Undefined behavior.
See also: C11 sec. 6.5
=============================================================
```

The Java semantics effort has also produced a test suite of several hundreds of programs that thoroughly and systematically test all the Java language constructs, because no such test suite was available. [1] The JavaScript semantics passes all the 2,782 core language tests part of the ECMAScript 5 conformance testuite. To put this in perspective, among the existing implementations of JavaScript, only Chrome's passes all the tests, and no other existing semantics attempt of JavaScript passes more than 90%. In addition to a reference implementation for a language, a $\mathbb{K}$ executable semantics also yields a simple coverage metric for a test suite: the set of semantic rules it exercises. The semantics of JavaScript revealed that the ECMAScript 5 conformance test suite fails to cover several semantic rules. Guided by the semantics, we wrote tests to exercise those rules and those tests revealed bugs both in production JavaScript engines (Chrome, Safari, Firefox) and in other semantics.

3 From Language Semantics to Program Verification

An operational semantics of a programming language, be it defined in $\mathbb{K}$ or not, defines an execution model the language typically in terms of a transition relation $cfg \Rightarrow cfg'$ between program configurations, and can serve as a formal basis for language understanding, design, implementation, and so on. On the other hand, an axiomatic semantics defines a proof system typically in terms of Hoare triples $\{\psi\}$ `code` $\{\psi'\}$, and can serve as a basis for program verification. To increase confidence in program verifiers and thus avoid problems like the one discussed in the introduction, ideally the axiomatic semantics should be proved sound w.r.t. the operational semantics (see, e.g., [28]). Needless to say that defining an axiomatic semantics for a real language like C is no simpler than defining an operational semantics, and that proving their equivalence is a burden that

[1] The official test suite for Java implementations is the Java Compatibility Kit (JCK) from Oracle, which is not publicly available. Oracle offers free access for non-profit organizations willing to implement the whole JDK (`http://openjdk.java.net/groups/conformance/JckAccess/`), i.e., both the language and the class library. We applied and Oracle rejected our request, because we did not "implement" the complete Java library. Also, note that the NIST Juliet testsuite (`http://samate.nist.gov/SARD/testsuite.php`) is meant to asses the capability of Java static analysis tools, and not the completeness of Java implementations.

few can take. Consequently, most program verifiers are actually based on no formal semantics of their target language at all and, as discussed above, end up sometimes proving wrong programs correct.

We have recently proposed a different approach to program verification, *reachability logic* [29–32], which introduces the notion of a reachability rule to express dynamic properties of programs, as a generalization of both a rewrite rule and a Hoare triple. Thus, reachability logic unifies operational and axiomatic semantics. Reachability logic builds upon *matching logic* [33–36] as a formalism to express static state properties. The overall idea of our verification approach is to start with a rewriting-based semantics of a programming language (which is operational), and to derive program properties with the same semantics, without giving the language any other (axiomatic) semantics.

Matching logic allows us to state and reason about structural properties over arbitrary program configurations. The main intuition underlying matching logic is that "terms are predicates and their satisfaction is matching". Syntactically, it introduces a new formula construct, called a *basic pattern*, which is a term possibly containing variables, e.g., a configuration term expressing a desired structure of the program configuration. The formulae, which can be built using basic patterns and arbitrary other conventional logical constructs, are called *patterns*. This way, we can compose structural requirements and add logical constraints over the variables appearing in basic patterns. Semantically, the models of basic patterns are concrete elements, e.g., concrete configurations regarded as ground terms, where such an element satisfies a basic pattern iff it *matches* it. Considering some configuration structure with a top-level cell $\langle ... \rangle_{\mathsf{cfg}}$ holding, in any order, other cells with semantic data such as the code $\langle ... \rangle_{\mathsf{k}}$,[2] an environment $\langle ... \rangle_{\mathsf{env}}$, a heap $\langle ... \rangle_{\mathsf{heap}}$, an input buffer $\langle ... \rangle_{\mathsf{in}}$, an output buffer $\langle ... \rangle_{\mathsf{out}}$, etc., configurations then have the structure:

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

The contents of the cells can be various algebraic data types, such as trees, lists, sets, maps, etc. Here are two particular concrete configurations (note that `x` and `y` are program variables, which unlike in Hoare logics are not logical variables; in matching logic they are constants used to build programs or fragments of programs):

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

Different languages may have different configuration structures. For example, languages whose semantics are intended to be purely syntactic and based on substitution, such as 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 has more than 70 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, as mentioned, allows configuration patterns (i.e., terms with variables) as particular formulae. To simplify terminology, all matching logic's formulae

[2] In mathematical mode, we prefer the notation $\langle ... \rangle_{\mathsf{k}}$ for cells instead of the XML-like notation `<k>...</k>` preferred in ASCII.

are called patterns. As a purposely artificial example, consider the pattern

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

This is satisfied by all configurations where program variable x points to a location p holding a positive integer i different from p. Variables matching the irrelevant parts of a cell, such as the variables e, σ, and c above, are called *structural frames*; when reasoning about languages defined using $\mathbb{K}$, the structural frames typically result from ellipses in rules, that is, from the parts of the configuration which do not change. They are needed in order for the pattern to properly match the expected structure of the desired configurations. For example, if we want to additionally state that p is the only location allocated in the heap, then we can just remove σ from the pattern above and obtain:

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

Matching logic allows us to reason about configurations, e.g., to prove:

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

To specify more complex properties, one can use abstractions (e.g., singly-linked lists matched in the heap, etc.), which can be axiomatized and proved sound using conventional means. In fact, as shown in [35–37], like separation logic, matching logic can also be used as a program logic in the context of conventional axiomatic (Hoare) 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 tedious soundness proofs need to be done. Instead, we prefer to use reachability logic, which allows us to use the operational semantics of the language for program verification as well.

An unconditional *reachability rule* is a pair $\varphi \Rightarrow \varphi'$, where φ and φ' are matching logic patterns (not necessarily closed). The semantics of a reachability rule captures the intuition of *partial correctness* in axiomatic semantics: any configuration satisfying φ either rewrites/transits forever or otherwise reaches through (zero or more) successive transitions a configuration satisfying φ'. In $\mathbb{K}$, programming languages can be given operational semantics based on rewrite rules of the form "$l \Rightarrow r$ if b", where l and r are configuration terms with variables constrained by boolean condition b. Such rules can be expressed as reachability rules $l \wedge b \Rightarrow r$. On the other hand, a Hoare triple of the form $\{\psi\}\,\mathtt{code}\,\{\psi'\}$ can be regarded as a reachability rule $\langle\cdots\ \langle\mathtt{code}\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge \overline{\psi} \Rightarrow \langle\cdots\ \langle\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \wedge \overline{\psi'}$ between patterns over minimal configurations holding only the code. Here the ellipses represent appropriate structural frames, $\langle\rangle_{\mathsf{k}}$ is the configuration holding the empty code, and $\overline{\psi}$ and $\overline{\psi'}$ are variants of the original pre/post conditions replacing program variables with appropriate logical variables (an example will be shown shortly). Therefore, reachability rules smoothly capture the basic ingredients of both operational and axiomatic semantics, in that both operational semantics rules and axiomatic semantics Hoare triples are instances of reachability rules.

$$\textsc{Axiom}: \frac{\varphi \Rightarrow \varphi' \in \mathcal{A}, \quad \psi \text{ is a FOL formula (the } \textit{logical frame}\text{)}}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

$$\textsc{Reflexivity}: \mathcal{A} \vdash_\emptyset \varphi \Rightarrow \varphi$$

$$\textsc{Transitivity}: \frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \cup C \vdash_\emptyset \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

$$\textsc{Consequence}: \frac{\models \varphi_1 \rightarrow \varphi_1' \;\; \mathcal{A} \vdash_C \varphi_1' \Rightarrow \varphi_2' \;\; \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

$$\textsc{Case Analysis}: \frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

$$\textsc{Abstraction}: \frac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \text{ where } X \cap FV(\varphi') = \emptyset}{\mathcal{A} \vdash_C \exists X\, \varphi \Rightarrow \varphi'}$$

$$\textsc{Circularity}: \frac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

Fig. 2. Proof system for (one-path) reachability using unconditional rules.

Figure 2 shows the reachability logic proof system for unconditional reachability rules. This is a simplification of a more general proof system in [30], where conditional reachability rules were also considered, for the particular rewrite logic theories supported by $\mathbb{K}$ (recall that in $\mathbb{K}$ the reachability rules are unconditional, because the side conditions can be moved into the LHS of the rule). We here only discuss the one-path variant of reachability logic, where $\varphi \Rightarrow \varphi'$ means that φ' is matched by some configuration reached after some sequence of transitions from a configuration matching φ. The all-path variant is more complex and can be found in [29]. The one-path and all-path reachability logic variants are equally expressive when the target programming language is deterministic. The target language is given as a reachability system $\mathcal{S}$ (from "semantics"). The soundness result in [30] guarantees that $\varphi \Rightarrow \varphi'$ holds semantically in the transition system generated by $\mathcal{S}$ if $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ is derivable. Note that the proof system derives more general sequents of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, where $\mathcal{A}$ and C are sets of reachability rules. Rules in $\mathcal{A}$ are called *axioms* and rules in C are called *circularities*. If C does not appear in a sequent, it means it is empty: $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is a shorthand for $\mathcal{A} \vdash_\emptyset \varphi \Rightarrow \varphi'$. Initially, C is empty and $\mathcal{A}$ is $\mathcal{S}$. During the proof, circularities can be added to C via Circularity and flushed into $\mathcal{A}$ by Transitivity or Axiom.

The intuition is that rules in $\mathcal{A}$ can be assumed valid, while those in C have been postulated but not yet justified. After making progress it becomes (coinductively) valid to rely on them. The intuition for sequent $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, read "$\mathcal{A}$ with circularities C proves $\varphi \Rightarrow \varphi'$", is: $\varphi \Rightarrow \varphi'$ is true if the rules in $\mathcal{A}$ are true and those in C are true after making progress, and if C is nonempty then φ reaches φ' (or diverges) after at least one transition. Let us now discuss the proof rules.

Axiom states that a trusted rule can be used in any *logical frame* ψ. The logical frame is formalized as a patternless formula, as it is meant to only add logical but no structural constraints. Incorporating framing into the axiom rule is necessary to make logical constraints available while proving the conditions of the axiom hold. Since reachability logic keeps a clear separation between program variables and logical variables the logical constraints are persistent, that is, they do not interfere with the dynamic nature of the operational rules and can therefore be safely used for framing.

Reflexivity and Transitivity correspond to homonymous closure properties of the reachability relation. Reflexivity requires C to be empty to meet the requirement above, that a reachability property derived with nonempty C takes one or more steps. Transitivity releases the circularities as axioms for the second premise, because if there are any circularities to release the first premise is guaranteed to make progress.

Consequence and Case Analysis are adapted from Hoare logic. In Hoare logic Case Analysis is typically a derived rule, but there does not seem to be any way to derive it language-independently. Ignoring circularities, we can think of these five rules discussed so far as a formal infrastructure for symbolic execution.

Abstraction allows us to hide irrelevant details of φ behind an existential quantifier, which is particularly useful in combination with the next proof rule.

Circularity has a coinductive nature and allows us to make a new circularity claim at any moment. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If we succeed in proving the claim using itself as a circularity, then the claim holds. This would obviously be unsound if the new assumption was available immediately, but requiring progress before circularities can be used ensures that only diverging executions can correspond to endless invocation of a circularity.

Consider, for example, the simple imperative language together with the sum program fragment, SUM, discussed in Section 2, but without the initial assignment `n=100`, that is:

```
s = 0;
while(n>0) {
  s = s + n;
  n = n - 1;
}
```

In conventional Hoare logic, the property to verify here is

$$\{\texttt{n} = \texttt{oldn} \land \texttt{n} > 0\}\ \ \text{SUM}\ \ \{\texttt{n} = 0 \land \texttt{s} = \texttt{oldn} *_{Int} (\texttt{oldn} +_{Int} 1) /_{Int} 2\}$$

where $\{\texttt{n} = \texttt{oldn} \land \texttt{n} > 0\}$ is the precondition and $\{\texttt{n} = 0 \land \texttt{s} = \texttt{oldn} *_{Int} (\texttt{oldn} +_{Int} 1) /_{Int} 2\}$ is the postcondition, with *Int*-subscripted operations being the corresponding mathematical domain operations (note that in Hoare logic no distinction is made between program variables and logical variables). In order to prove it, an axiomatic semantics of the language is needed, which is given as a proof system in terms of such Hoare triples. When correctness is paramount, a proof of correctness of the Hoare logic proof system wrt a trusted semantics, typically an executable one which acts as a reference model of the language, is also needed. This may look easy and even desirable, but the reality is that semantics of real languages are not trivial to define and they continuously evolve as the language itself evolves. Therefore, giving two different semantics and maintaining proofs of correctness between them are highly non-trivial and demotivating tasks. Finally, if one wants the Hoare logic to also be as powerful as it can be for the target language, that is, to allow us to derive any semantically valid Hoare triples, then one has to also prove its relative completeness, yet another highly non-trivial task. Moreover, and even worse from an engineering perspective, each of the above needs to be done for each programming language separately.

On the other hand, reachability logic requires no additional semantics for the target language and no correctness or completeness proofs specific to each language. It takes the executable semantics of the programming language, which is regarded as reference model, as input axioms, and then the language-independent proof rules (e.g., those in Figure 2) are proved correct and relatively complete once and for all languages [29–32]. Without any syntactic sugar, the reachability logic specifications may be more verbose than the Hoare logic ones. However, we have found that in practice one spends a lot more time on coming up with the conceptually right properties to verify than on writing them in a particular notation. Moreover, in both Hoare logic and reachability logic we typically develop syntactic sugar notations that reduce the user burden (e.g., writing a loop invariant as a formal comment in the code). Without any sugar, the reachability rule below captures the same specification of SUM as the Hoare triple above:

$$\begin{array}{l}\langle\langle \mathrm{SUM}\rangle_{\mathsf{k}}\ \langle \mathtt{s} \mapsto s,\ \mathtt{n} \mapsto n\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}\ \wedge\ n \geq_{Int} 0\\ \Rightarrow\ \langle\langle\rangle_{\mathsf{k}}\ \langle \mathtt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2,\ \mathtt{n} \mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}\end{array}$$

We encourage the reader to derive the reachability rule above on her own, using the proof system in Figure 2. Complete details can be found in [34]. We here only give the high-level structure of the proof. By Axiom with the semantic rule of assignment shown in Section 2 and by Transitivity, we reduce the above reachability rule to one where the s in the left-hand-side pattern is replaced with 0. Let LOOP be the while loop of the SUM program. Like in Hoare logic proofs, we have to derive an invariant for LOOP. In reachability logic, we formalize invariants also as reachability rules, in our case as

$$\begin{array}{l}\langle\langle \mathrm{LOOP}\rangle_{\mathsf{k}}\ \langle \mathtt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2,\ \mathtt{n} \mapsto n'\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}\ \wedge\ n' \geq_{Int} 0\\ \Rightarrow\ \langle\langle\rangle_{\mathsf{k}}\ \langle \mathtt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2,\ \mathtt{n} \mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{cfg}}\end{array}$$

If this “invariant” reachability rule holds, then our original reachability rule can be derived using Abstraction, Consequence and Transitivity. To derive this invariant rule, we first use Circularity to claim it as a circularity, and then unroll the LOOP using the executable semantic rule for while, then do a Case Analysis for the resulting conditional, and then run the executable semantics of the corresponding assignments via Axiom, intermingled with applications of Consequence and Transitivity. Note that we can use the circularity claim in the proof of the positive branch of the conditional, because the Transitivity added it to the set of axioms once the unrolling of the while loop took place. We leave the rest of the details to the reader, as an exercise.

$\mathbb{K}$ implements reachability logic, the same way Maude implements rewriting logic. Since patterns allow variables and constraints on them in configurations, $\mathbb{K}$ rewriting becomes symbolic execution with the semantic rules of the language. Its symbolic execution engine is connected to the Z3 SMT solver [38]. We next show an example C program verified with our current implementation of reachability logic in $\mathbb{K}$, mentioning

```
struct listNode { int val; struct listNode *next; };
struct listNode* reverseList(struct listNode *x)
 rule ⟨$ ⇒ return ?p;  ···⟩k ⟨···  list(x)(A) ⇒ list(?p)(rev(A))  ···⟩heap
{
  struct listNode *p, *y;
  p = NULL;
 inv ⟨···  list(p)(?B), list(x)(?C)  ···⟩heap  ∧  A = rev(?B)@?C
  while(x != NULL) {
    y = x->next;
    x->next = p;
    p = x;
    x = y;
  }
  return p;
}
```

Fig. 3. C function reversing a singly-linked list.

that we have similarly verified various programs manipulating lists and trees, performing arithmetic and I/O operations, and implementing sorting algorithms, binary search trees, AVL trees, and the Schorr-Waite graph marking algorithm. The Matching Logic web page, `http://matching-logic.org`, contains an online interface to run MatchC, an instance of our verifier for C, where users can try more than 50 existing examples (or upload their own). To simplify writing properties, MatchC allows users to write reachability rules and invariant patterns as comments in the C program.

Figure 3 shows the classic list reverse program, together with all the specifications that the user of MatchC has to provide (grayed areas, given as code annotations). MatchC verifies this program for full correctness, not only memory safety, in 0.06 seconds. The user-provided specifications are translated into reachability rule proof obligations by MatchC and then attempted to be proved automatically. The "$" stands for the function body, the "···" for structural frame variables, the variables starting with "?" are existentially quantified over the current formula, etc. We do not mean to explain the MatchC notation in detail here; we only show this example to highlight the fact that reachability logic verification, in spite of being based on "low-level" operational semantics, still allows a comfortable level of abstraction.

Let $\mathcal{A}$ be the rewrite system giving the semantics of the C language, and let C be the set of reachability rules corresponding to user-provided specifications (properties that one wants to verify, like the grayed ones above). MatchC derives the rules in C using the proof system in Figure 2. It begins by applying Circularity for each rule in C and reduces the task to deriving individual sequents of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$. To prove them, MatchC rewrites φ using rules in $\mathcal{A} \cup C$ searching for a formula that implies φ'. Whenever the semantics rule for `if` in $\mathcal{A}$ cannot apply because its condition is symbolic, a Case Analysis is applied and formula split into a disjunction. When no rule can be applied, abstraction axioms are attempted. If application of an abstraction axiom would result into a more concrete formula, the verifier applies the respective axiom (for instance, knowing the head of a linked list is not null results in an automatic list unrolling).

Regarding from reachability logic's perspective, $\mathbb{K}$ consists of a collection of heuristics and optimizations to perform *proof search*, that is, to derive proof derivations using the reachability logic proof system. For example, suppose that the initial configuration pattern is all concrete/ground (i.e., when it contains no variables) and that $\mathbb{K}$ is requested to use its rewrite engine to simply execute the program in its initial state. In terms of proof derivation with the one-path reachability proof system in Figure 2 and its more general variant in [30], this corresponds to a derivation of a one-path reachability rule. If $\mathbb{K}$ is requested to search the entire configuration-space to find all the configurations that can be reached as a consequence of a non-deterministic semantics, that corresponds to a derivation of an all-path reachability rule using the generalized proof system in [29]. Similarly, when using $\mathbb{K}$'s model-checking capabilities or when deductively verifying programs like above, all we do can be framed in terms of deriving proofs using a rigorously defined sound and relatively complete proof system.

4 Additional Related Work

The idea of developing $\mathbb{K}$ as a programming language semantic framework has been first suggested in 2003, in a programming language class taught at the University of Illinois by the author, whose lecture notes were published as a technical report [39]. One year later, the main idea was published in [40]. At that time, we used Maude as an execution language and hand-translated $\mathbb{K}$ definitions to Maude, one language at a time, so $\mathbb{K}$ was simply a Maude methodology for defining rewriting logic semantics to programming languages. $\mathbb{K}$ had several implementations in the meanwhile, most of them consisting of parsers and translators to Maude using Perl or Haskell. The current implementation, reachable from `http://kframework.org`, is implemented fully in Java except for the mathematical domain solver needed for program verification, which currently is Z3 [38].

In addition to the complete language semantics already mentioned in Section 2, there are several incomplete or yet unfinished language semantics, such as Python [41], Scheme [42], as well as various aspects of features of Haskell [43], X10 [44], a RISC assembly [45,46], LLVM [47], Verilog [48], as well as a static policy checker for C [49] and a framework for domain specific languages [50,51].

$\mathbb{K}$'s ability to express truly concurrent computations has been used in researching safe models for concurrency [52], synchronization of agent systems [53], models for P-Systems [54, 55], and for the x86-TSO relaxed memory model [56]. $\mathbb{K}$ has been used for designing type checkers/inferencers [57], for model checking executions with predicate abstraction [58,59] and heap awareness [60], for symbolic execution [61–64], computing worst case execution times [65–67], studying program equivalence [68,69], programming language aggregation [70], and runtime verification [56,71]. Additionally, the C definition mentioned above has been used as a program undefinedness checker to analyze C programs [72]. The theoretical relationships between $\mathbb{K}$ language definitions and their Maude counterparts, both at the concrete and at the symbolic level, are studied in [73]. $\mathbb{K}$, through its underlying reachability logic foundation, is organized as an institution in [74]. Finally, some techniques for automatic inference of matching logic specifications are investigated in [75, 76].

5 Conclusion

This paper presented a glimpse of the 𝕂 framework for formally defining programming languages, which was inspired from using rewrite logic as a semantic framework. 𝕂 aims at bringing formal semantics mainstream, by providing an intuitive notation and an attractive set of language-independent tools that can be used with any language once a semantics is given to that language. 𝕂 builds upon our firm belief that all programming languages must have formal semantics, and that their semantics is not only a mathematical artifact that helps us better understand the language in question, but that it can in fact be incredibly useful in practice. Virtually all program analysis tools can be generated automatically from formal semantics. 𝕂 may not be the final answer to this quest, but we believe that it has proven the concept, that it is indeed possible to build an effective collection of language tools based entirely and only on the language semantics.

References

1. ISO/IEC, “Programming languages—C,” ISO/IEC WG14, ISO 9899:2011, December 2011. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG14/www/standards
2. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “VCC: A practical system for verifying concurrent C,” in *Theorem Proving in Higher Order Logics*, ser. LNCS. Springer, 2009, vol. 5674, pp. 23–42.
3. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C,” in *Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247.
4. J. Meseguer, “Conditioned rewriting logic as a united model of concurrency,” *Theoretical Computer Science*, vol. 96, no. 1, pp. 73–155, 1992.
5. A. Verdejo and N. Martí-Oliet, “Executable structural operational semantics in Maude,” Departamento de Sistemas Informàticos y Programaciòn, Universidad Complutense de Madrid, Tech. Rep. 134-03, 2003.
6. ——, “Executable structural operational semantics in Maude,” *Journal of Logic and Algebraic Programming*, vol. 67, no. 1-2, pp. 226–293, 2006.
7. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott, *All About Maude*, ser. LNCS, 2007, vol. 4350.
8. G. Kahn, “Natural semantics,” in *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, ser. LNCS, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., vol. 247. Springer, 1987, pp. 22–39.
9. G. D. Plotkin, “A structural approach to operational semantics,” University of Aarhus, Tech. Rep. DAIMI FN-19, 1981, republished in Journal of Logic and Algebraic Programming, Volume 60-61, 2004.
10. ——, “A structural approach to operational semantics,” *Journal of Logic and Algebraic Programming*, vol. 60-61, pp. 17–139, 2004.
11. T.-F. Şerbănuţă, G. Roşu, and J. Meseguer, “A rewriting logic approach to operational semantics,” *Information & Computation*, vol. 207, no. 2, pp. 305–340, 2009.
12. G. Berry and G. Boudol, “The chemical abstract machine,” in *POPL*, 1990, pp. 81–94.

13. ——, "The chemical abstract machine," *Theoretical Computer Science*, vol. 96, no. 1, pp. 217–248, 1992.
14. M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theoretical Computer Science*, vol. 103, no. 2, pp. 235–271, 1992.
15. A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information and Computation*, vol. 115, no. 1, pp. 38–94, 1994.
16. M. Hennessy, *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. New York, N.Y.: John Wiley and Sons, 1990.
17. P. D. Mosses, "Foundations of modular SOS," in *MFCS*, ser. LNCS, M. Kutylowski, L. Pacholski, and T. Wierzbicki, Eds., vol. 1672. Springer, 1999, pp. 70–80.
18. ——, "Pragmatics of modular SOS," in *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, ser. LNCS, H. Kirchner and C. Ringeissen, Eds., vol. 2422, 2002, pp. 21–40.
19. ——, "Modular structural operational semantics," *Journal of Logic & Algebraic Programming*, vol. 60-61, pp. 195–228, 2004.
20. J. Meseguer and C. Braga, "Modular rewriting semantics of programming languages," in *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, ser. LNCS, C. Rattray, S. Maharaj, and C. Shankland, Eds., vol. 3116. Springer, 2004, pp. 364–378.
21. C. Braga and J. Meseguer, "Modular rewriting semantics in practice," *Electronic Notes in Theoretical Computer Science*, vol. 117, pp. 393–416, 2005.
22. F. Chalub and C. Braga, "Maude MSOS tool," in *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, ser. Electronic Notes in Theoretical Computer Science, G. Denker and C. Talcott, Eds., vol. 176(4), 2007, pp. 133–146.
23. G. Roșu and T. F. Șerbănuță, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.
24. C. Ellison and G. Roșu, "An executable formal semantics of C with applications," in *POPL*, 2012, pp. 533–544.
25. C. Hathhorn, C. Ellison, and G. Roșu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 2015.
26. D. Bogdănaș and G. Roșu, "K-Java: A Complete Semantics of Java," in *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*. ACM, January 2015, pp. 445–456.
27. D. Park, A. Ștefănescu, and G. Roșu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 2015.
28. A. W. Appel, "Verified software toolchain," in *ESOP*, ser. LNCS, vol. 6602, 2011, pp. 1–17.
29. A. Ștefănescu, S. Ciobâcă, R. Mereuță, B. M. Moore, T. F. Șerbănuță, and G. Roșu, "All-path reachability logic," in *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, ser. LNCS, vol. 8560. Springer, July 2014, pp. 425–440.
30. G. Roșu, A. Ștefănescu, S. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *LICS'13*. IEEE, 2013.
31. G. Roșu and A. Ștefănescu, "Checking reachability using matching logic," in *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012, pp. 555–574.
32. ——, "Towards a unified theory of operational and axiomatic semantics," in *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, ser. LNCS, vol. 7392. Springer, 2012, pp. 351–363.

33. G. Roșu, “Matching logic — extended abstract,” in *Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA’15)*, ser. LNCS, vol. to appear. Springer, 2015.
34. G. Roșu and A. Ștefănescu, “From Hoare logic to matching logic reachability,” in *FM*, ser. LNCS, D. Giannakopoulou and D. Méry, Eds., vol. 7436. Springer, 2012, pp. 387–402.
35. ——, “Matching logic: a new program verification approach,” in *ICSE (NIER track)*, 2011, pp. 868–871.
36. G. Roșu, C. Ellison, and W. Schulte, “Matching logic: An alternative to Hoare/Floyd logic,” in *AMAST*, ser. LNCS, vol. 6486, 2010, pp. 142–162.
37. G. Roșu and A. Ștefănescu, “Matching logic rewriting: Unifying operational and axiomatic semantics in a practical and generic framework,” University of Illinois, Tech. Rep. http://hdl.handle.net/2142/28357, November 2011.
38. L. M. de Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 337–340.
39. G. Roșu, “CS322, Fall 2003 - Programming language design: Lecture notes,” University of Illinois at Urbana-Champaign, Department of Computer Science, Tech. Rep. UIUCDCS-R-2003-2897, Dec. 2003, lecture notes of a course taught at UIUC.
40. J. Meseguer and G. Roșu, “Rewriting logic semantics: From language specifications to formal analysis tools,” in *Proceedings of the 2nd International Joint Conference on Automated Reasoning (IJCAR’04)*, ser. LNCS, vol. 3097. Springer, 2004, pp. 1–44.
41. D. Guth, “A formal semantics of Python 3.3,” Master’s thesis, University of Illinois at Urbana-Champaign, July 2013. [Online]. Available: https://github.com/kframework/python-semantics
42. P. Meredith, M. Hills, and G. Roșu, “An Executable Rewriting Logic Semantics of K-Scheme,” in *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME’07), Technical Report DIUL-RT-0701*, D. Dube, Ed. Laval University, 2007, pp. 91–103.
43. D. Lazar, “K definition of Haskell’98,” 2012. [Online]. Available: https://github.com/davidlazar/haskell-semantics
44. M. Gligoric, D. Marinov, and S. Kamin, “CoDeSe: fast deserialization via code generation,” in *ISSTA*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 298–308.
45. M. Asăvoae, “K semantics for assembly languages : A case study,” in *K’11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., vol. 304, 2014, pp. 111–125.
46. ——, “A K-based methodology for modular design of embedded systems,” in *WADT (preliminary proceedings)*, ser. TR-08/12, Universidad Complutense de Madrid, 2012, p. 16. [Online]. Available: http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf
47. C. Ellison and D. Lazar, “K definition of the LLVM assembly language,” 2012. [Online]. Available: https://github.com/davidlazar/llvm-semantics
48. P. O. Meredith, M. Katelman, J. Meseguer, and G. Roșu, “A formal executable semantics of Verilog,” in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE’10)*. IEEE, 2010, pp. 179–188.
49. M. Hills, F. Chen, and G. Roșu, “A rewriting logic approach to static checking of units of measurement in C,” in *RULE’08*, ser. Electronic Notes in Theoretical Computer Science, G. Kniesel and J. S. Pinto, Eds., vol. 290. Elsevier, 2012, pp. 51–67.
50. V. Rusu and D. Lucanu, “A K-based formal framework for domain-specific modelling languages,” in *FoVeOOS*, ser. LNCS, B. Beckert, F. Damiani, and D. Gurov, Eds., vol. 7421. Springer, 2011, pp. 214–231.
51. A. Arusoaie, D. Lucanu, and V. Rusu, “Towards a K semantics for OCL,” in *K’11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., vol. 304, 2014, pp. 81–96.
52. S. Heumann, V. S. Adve, and S. Wang, “The tasks with effects model for safe concurrency,” in *PPOPP*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. Vuduc, Eds. ACM, 2013, pp. 239–250.

53. P. Dinges and G. Agha, "Scoped synchronization constraints for large scale actor systems," in *COORDINATION*, ser. LNCS, M. Sirjani, Ed., vol. 7274. Springer, 2012, pp. 89–103.
54. T. F. Şerbănuţă, G. Ştefănescu, and G. Roşu, "Defining and executing P systems with structured data in K," in *WMC*, ser. LNCS, D. W. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, Eds., vol. 5391. Springer, 2008, pp. 374–393.
55. C. Chira, T.-F. Şerbănuţă, and G. Ştefănescu, "P systems with control nuclei: The concept," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 326–333, 2010.
56. T. F. Şerbănuţă, "A rewriting approach to concurrent programming language design and semantics," Ph.D. dissertation, University of Illinois at Urbana-Champaign, December 2010, https://www.ideals.illinois.edu/handle/2142/18252.
57. C. Ellison, T. F. Şerbănuţă, and G. Roşu, "A rewriting logic approach to type inference," in *WADT*, ser. LNCS, A. Corradini and U. Montanari, Eds., vol. 5486. Springer, 2008, pp. 135–151.
58. I. M. Asăvoae and M. Asăvoae, "Collecting semantics under predicate abstraction in the K framework," in *WRLA*, ser. LNCS, P. C. Ölveczky, Ed., vol. 6381. Springer, 2010, pp. 123–139.
59. I. M. Asăvoae, "Systematic design of abstractions in K," in *WADT (preliminary proceedings)*, ser. TR-08/12, Universidad Complutense de Madrid, 2012, p. 9. [Online]. Available: http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf
60. J. Rot, I. M. Asăvoae, F. S. de Boer, M. M. Bonsangue, and D. Lucanu, "Interacting via the heap in the presence of recursion," in *ICE*, ser. Electronic Proceedings in Theoretical Computer Science, M. Carbone, I. Lanese, A. Silva, and A. Sokolova, Eds., vol. 104, 2012, pp. 99–113.
61. I. M. Asăvoae, M. Asăvoae, and D. Lucanu, "Path directed symbolic execution in the K framework," in *SYNASC*, T. Ida, V. Negru, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, Eds. IEEE Computer Society, 2010, pp. 133–141.
62. I. M. Asăvoae, "Abstract semantics for alias analysis in K," in *K'11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., vol. 304, 2014, pp. 97–110.
63. A. Arusoaie, D. Lucanu, and V. Rusu, "A generic framework for symbolic execution," in *SLE*, ser. Lecture Notes in Computer Science, vol. 8225, 2013, pp. 281–301.
64. A. Arusoaie, "A generic framework for symbolic execution: Theory and applications," Ph.D. dissertation, Faculty of Computer Science, Alexandru I. Cuza, University of Iasi, https://fmse.info.uaic.ro/publications/193/, sep 2014.
65. M. Asăvoae, D. Lucanu, and G. Roşu, "Towards semantics-based WCET analysis," in *WCET*, ser. Austian Computer Society (OCG), C. Healy, Ed., 2011.
66. M. Asăvoae, I. M. Asăvoae, and D. Lucanu, "On abstractions for timing analysis in the K framework," in *FOPARA*, ser. LNCS, R. Pena, M. Eekelen, and O. Shkaravska, Eds., vol. 7177. Springer Berlin Heidelberg, 2012, pp. 90–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32495-6_6
67. M. Asăvoae and I. M. Asăvoae, "On the modular integration of abstract semantics for WCET analysis," in *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, 2013, pp. 19–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-12466-7_2
68. D. Lucanu and V. Rusu, "Program Equivalence by Circular Reasoning," *Formal Aspects of Computing*, 2014, to appear. [Online]. Available: https://hal.inria.fr/hal-01065830
69. S. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A Language-Independent Proof System for Mutual Program Equivalence," in *ICFEM'14 - 16th International Conference on Formal Engineering Methods*, ser. LNCS, vol. 8829. Luxembourg-Ville, Luxembourg: Springer, Nov. 2014, pp. 75–90. [Online]. Available: https://hal.inria.fr/hal-01030754

70. ——, "A Theoretical Foundation for Programming Languages Aggregation," in *22nd International Workshop on Algebraic Development Techniques*, ser. LNCS (to appear). Sinaia, Romania: Spriger Verlag, Sep. 2014. [Online]. Available: https://hal.inria.fr/hal-01076641
71. G. Roșu, W. Schulte, and T. F. Șerbănuță, "Runtime verification of C memory safety," in *RV*, ser. LNCS, S. Bensalem and D. Peled, Eds., vol. 5779. Springer, 2009, pp. 132–151.
72. J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *PLDI*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 335–346.
73. A. Arusoaie, D. Lucanu, V. Rusu, T.-F. Șerbănuță, A. Ștefănescu, and G. Roșu, "Language Definitions as Rewrite Theories," in *International Workshop on Rewriting Logic and Application*, Grenoble, France, Apr. 2014, (To appear in Springer LNCS). [Online]. Available: https://hal.inria.fr/hal-00950775
74. C. E. Chiriță and T. F. Șerbănuță, "An institutional foundation for the şemantic framework," in *22nd International Workshop on Algebraic Development Techniques (WADT'14)*, ser. LNCS, 2014, to appear.
75. M. A. Feliú, "Logic-based techniques for program analysis and specification synthesis," Ph.D. dissertation, Universitat Politècnica de València, Departamento de Sistemas Informáticos y Computación, September 2013.
76. M. Alpuente, M. A. Feliú, and A. Villanueva, "Automatic inference of specifications using matching logic," in *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*. ACM, 2013, pp. 127–136.