A Language-Independent Program Verification Framework

Xiaohong Chen[1(✉)] and Grigore Roşu[1,2]

[1] University of Illinois at Urbana-Champaign, Champaign, USA
xc3@illinois.edu
[2] Runtime Verification Inc., Champaign, USA

Abstract. This invited paper describes an approach to language-independent deductive verification using the $\mathbb{K}$ semantics framework, in which an operational semantics of a language is defined and a program verifier together with other language tools are generated automatically, correct-by-construction.

1 Introduction and Motivation

Given a program and a specification of the program, the *deductive verification problem* asks if the program satisfies the specification. If the answer is positive, a collection of proof obligations is expected to be generated as evidence, while counterexamples, often of the form of concrete program execution traces, witness the negative answer. Many program verification approaches are associated with a *program logic* and a *proof system* of that logic that allows to derive new facts about programs from axioms and established facts. The *proof rules* of the proof system define the semantics of the target language. Hoare logic [12], for example, is a program logic proposed in 1969 for a simple imperative language which we refer to as IMP. The syntax of IMP is defined in Fig. 1, where *Id* is the category for program variables and *Exp* is the category for arithmetic expressions. If-statements and while-statements use *Exp* as conditions, where zero means false and nonzero values mean true. The specification of an IMP program is written as a *Hoare triple*, consisting of the program and its precondition and postcondition (e.g., Fig. 2). A set of Hoare logic proof rules can then be used to rigorously reason about the correctness of IMP programs (Fig. 3). Notice how every IMP language construct has a corresponding Hoare logic proof rule.

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{Id} \mid \mathit{Int} \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\
\mathit{Stmt} & ::= & \mathit{Id}\ \texttt{=}\ \mathit{Exp}\texttt{;} \mid \mathit{Stmt}\ \mathit{Stmt} \mid \texttt{\{}\ \mathit{Stmt}\ \texttt{\}} \mid \texttt{\{\}} \\
 & \mid & \texttt{if (}\ \mathit{Exp}\ \texttt{)}\ \mathit{Stmt}\ \mathit{Stmt} \mid \texttt{while (}\ \mathit{Exp}\ \texttt{)}\ \mathit{Stmt}
\end{array}
$$

Fig. 1. The syntax of the language IMP.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 92–102, 2018.
https://doi.org/10.1007/978-3-030-03421-4_7

$$\{\texttt{n} = n \wedge n \geq 0\}$$
$$\texttt{s = 0; while(n)\{s = s + n; n = n - 1;\}}$$
$$\{\texttt{s} = n(n+1)/2\}$$

Fig. 2. An IMP program `sum` that calculates the sum from 1 to n, together with its formal specification given as a pair of precondition (the first line) and postcondition (the last line). We use `teletype` font to write program variables (e.g. `n` and `s`) and *italic* font to write mathematical variables (e.g. n).

$$\frac{\cdot}{\{\varphi[E/X]\}\ X = E;\ \{\varphi\}}\ \text{HL-ASGN}$$

$$\frac{\{\varphi_1\}\ P\ \{\varphi_2\} \quad \{\varphi_2\}\ Q\ \{\varphi_3\}}{\{\varphi_1\}\ P\ Q\ \{\varphi_3\}}\ \text{HL-SEQ}$$

$$\frac{\{\varphi_1 \wedge B \neq 0\}\ P\ \{\varphi_2\} \quad \{\varphi_1 \wedge B = 0\}\ Q\ \{\varphi_2\}}{\{\varphi_1\}\ \texttt{if}(B) P\ Q\ \{\varphi_2\}}\ \text{HL-IF}$$

$$\frac{\{\varphi_{inv} \wedge B \neq 0\}\ P\ \{\varphi_{inv}\}}{\{\varphi_{inv}\}\ \texttt{while}(B) P\ \{\varphi_{inv} \wedge B = 0\}}\ \text{HL-WHILE}$$

$$\frac{\models \varphi_1' \to \varphi_1 \quad \{\varphi_1\}\ P\ \{\varphi_2\} \quad \models \varphi_2 \to \varphi_2'}{\{\varphi_1'\}\ P\ \{\varphi_2'\}}\ \text{HL-CNSQ}$$

Fig. 3. The Hoare logic proof system of the language IMP.

Hoare logic remains one of the most popular program logics since the day it was born, and researchers have proposed many variants of Hoare logic for more complicated languages and programs [3,14,17,18,21,22]. In the following, we will use the term "Hoare logic" to refer to all Hoare-style program logics, where the semantics of the target language is defined/axiomatized by the proof rules of that logic. Obviously, this makes Hoare logic *language-dependent*, as every language construct is associated with one or even more proof rules. When the language changes, the Hoare logic proof system for that language has to change accordingly, and thus all verification tools based on Hoare logic and its variants are language-dependent: a Java verifier cannot be used to verify C programs. Another notable characteristic of Hoare logic is that it is not directly executable. Therefore, in practice, language semanticists may need to define a separate trusted operational semantics that is executable, and carry out complex proofs of equivalence between the two semantics, which can take years to complete. All these facts make language design a highly expensive task, and *changing* the language rather inconvenient and demotivating, as it requires a thorough change of the Hoare logic proof system for that language and thus of all the related verification tools. If a trusted operational semantics is given, it needs to change, too, and a new proof of equivalence between the new Hoare logic and the new operational semantics should be carried out. This high cost brings us poor *reusability* of verification tools. Considering the fact that these tools often need several man-years to develop, the lack of reusability leads to a remarkable waste of resources and talent, as well as to duplicate work.

A common practice is then to develop verification tools for *intermediate verification languages* (IVL) such as Boogie [2] and Why [9], and translate the target languages to IVL. This brings some reusability, as verification tools are designed and implemented for IVL, in isolation from the target languages. However, correct program translation can be hard to develop. The proof of its correctness (called *soundness proof*) often involves the usage of higher-order theorem provers such as Coq [16] and Isabelle [20], not to mention that many real languages such as Java do not even have an official formal specification of the semantics. Thus, research about language-specific program logics and IVL tools sometimes have to compromise and claim "no intention of formally proving the soundness result" [1] (Fig. 5).

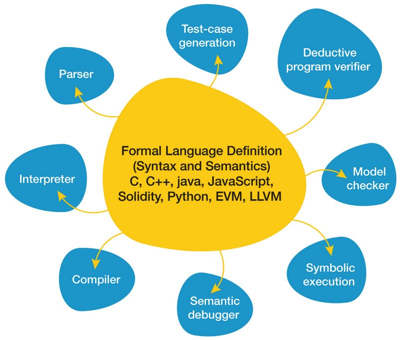

Fig. 4. The $\mathbb{K}$ framework approach to language design and verification.

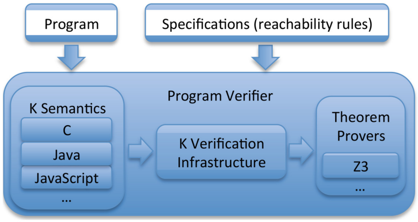

Fig. 5. A language-independent program verifier takes a program and its specification, and verifies it with respect to its formal semantics.

This motivated us to look for a verification methodology that is *language-independent*, which allows us to build verification tools that can verify *any property* of *any program* written in *any programming language*. The $\mathbb{K}$ framework (www.kframework.org) is our attempt towards such a verification methodology, based on the firm belief that every language should have a formal semantics, and all related language tools should be automatically generated from the semantics in a correct-by-construction manner (Fig. 4). $\mathbb{K}$ provides a meta-programming language to design programming languages. The formal semantics of a language, written as a $\mathbb{K}$ definition, serves as the *only* canonical reference to *all* language tools, and no other formal or informal semantics is needed. Case studies with a variety of real languages demonstrates that this ideal scenario is indeed feasible and practical.

The rest of the paper is organized as follows. Section 2 briefly introduces the $\mathbb{K}$ framework, and Sect. 3 shows how program verification is carried out with $\mathbb{K}$. We conclude in Sect. 5.

2 $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewrite-based executable semantics framework for programming language design. We use the language IMP in Fig. 1 as our running example (with minor modification on its syntax) to illustrate how to define programming languages and verify programs in $\mathbb{K}$.

The complete $\mathbb{K}$ definition for IMP is shown in Fig. 6, consisting of two $\mathbb{K}$ modules `IMP-SYNTAX` and `IMP`. The module `IMP-SYNTAX` defines the syntax of the language using the conventional BNF grammar, where terminals are in quotes. Syntax productions are separated by the "`|`" and "`>`", where "`|`" means the two productions have the same precedence while "`>`" means the previous production has higher precedence (binds tighter) than the one that follows. In other words, in the language IMP, all language constructs bind tighter than the sequential operator. `Int` and `Id` are two built-in categories of integers and identifiers (program variables), respectively. `Exp` is the category of expressions, which subsumes `Int` and `Id`, and contains two other productions for plus and minus. `Pgm` is the category of IMP programs. A wellformed IMP program declares a list of program variables in the beginning, followed by a statement. `Ids` is the category for lists of program variables, and it is defined using $\mathbb{K}$'s built-in template `List`. The first argument is the base category `Id`, and second argument is the separating character `","`.

Attributes are wrapped with braces "`[`" and "`]`". Some attributes are only for parsing purpose while others may carry additional semantic meaning and affect how $\mathbb{K}$ executes programs. The attribute `left` means that "`+`" and "`-`" are left-associative, so `1 - 2 + 3` should be parsed as `(1 - 2) + 3`. The attribute `strict` defines evaluation contexts. When $\mathbb{K}$ sees the expression e_1 `+` e_2 (and similarly e_1 `-` e_2), it first evaluates e_1 to an integer i_1 and e_2 to an integer i_2 in a *fully nondeterministic* way, and then evaluates $i_1 + i_2$. For example, there are in total $3! = 6$ different orders to evaluate the expression `((1 + 2) + (3 + 4)) + (5 + 6)`, because the most inner three parentheses must be evaluated first, and they can be evaluated in any order. The attribute `strict(1)` defines evaluation contexts only for the first argument. Therefore, when $\mathbb{K}$ sees an if-statement `if(`b`)` P Q, it only evaluates the condition b and keeps the branches P and Q untouched. In other words, the two branches of if-statements are *frozen* and will not be evaluated until the condition becomes a value. The attribute `bracket` tells $\mathbb{K}$ that certain productions are only used for grouping, and $\mathbb{K}$ will not generate nodes in its internal abstract syntax trees for those productions. Here, parentheses "`( )`" are used to group arithmetics expressions while curly brackets "`{ }`" are used to group program statements. The empty curly bracket "`{}`" represents the empty statement.

```
module IMP-SYNTAX
  imports DOMAINS-SYNTAX
  syntax Exp  ::= Int | Id
                | Exp "+" Exp                                      [left, strict]
                | Exp "-" Exp                                      [left, strict]
                | "(" Exp ")"                                           [bracket]
  syntax Stmt ::= Id "=" Exp ";"                                      [strict(2)]
                | "if" "(" Exp ")" Stmt Stmt                          [strict(1)]
                | "while" "(" Exp ")" Stmt
                | "{" Stmt "}"                                          [bracket]
                | "{" "}"
                > Stmt Stmt                                                [left]
  syntax Pgm  ::= "int" Ids ";" Stmt
  syntax Ids  ::= List{Id, ","}
endmodule
module IMP
  imports IMP-SYNTAX
  imports DOMAINS
  syntax KResult ::= Int
  configuration  <T> <k> $PGM:Pgm </k> <state> .Map </state> </T>
  rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
  rule I1 + I2 => I1 +Int I2
  rule I1 - I2 => I1 -Int I2
  rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
  rule S1:Stmt S2:Stmt => S1 ~> S2                                   [structural]
  rule if (I) S _ => S requires I =/=Int 0
  rule if (0) _ S => S
  rule while(B) S => if(B) {S while(B) S} {}                         [structural]
  rule {} => .                                                       [structural]
  rule <k> int (X, Xs => Xs); S </k> <state>... (. => X |-> 0) </state> [structural]
  rule int .Ids; S => S                                              [structural]
endmodule
```

Fig. 6. The complete 𝕂 definition for the language IMP.

The module `IMP` defines the operational semantics of IMP in terms of a set of human-readable rewrite rules (followed by the keyword `rule`). The category `KResult` tells 𝕂 which categories contain non-reducible values. It helps 𝕂 perform efficiently with evaluation contexts. The only category of values here is `Int`. Configuration is a core concept in the 𝕂 framework. A *configuration* represents a *program execution state*, holding all information that is needed for program execution. Configurations are organized into *cells*, which are labeled and can be nested. Simple languages such as IMP have only a few cells, while complex real languages such as C may have a lot more. Configurations are written in XML format.

The configurations of IMP have two cells: a `k` cell and a `state` cell. For clarity, we gather both cells and put them in a top-level cell called the `T` cell, but it is not mandatory. The `k` cell holds the rest computation (program fragments) that needs to execute and the `state` cell holds a map from program variables to their values in the memory. Initially, the `state` cell holds the empty map, denoted as `.Map`. In 𝕂, we write "`.`" for "nothing", and `.Map` means that nothing has type `Map`.

Initially, the `k` cell contains an IMP program `$PGM:Pgm`, where `$PGM` is a special 𝕂 variable name that tells 𝕂 the program is saved in a source file, and the name of the file is passed as argument in the command line when 𝕂 is invoked. 𝕂 will then read the source file and parse it as a `Pgm`, and put the result in the `k` cell.

𝕂 defines the language semantics in terms of a set of rewrite rules. A rewrite rule has the form *lhs* `=>` *rhs*, saying that any configuration γ that matches *lhs* rewrites to *rhs*, but as we will see later, 𝕂 offers a more flexible and succinct way to define rewrite rules. All rewrite rules in a language definition specify a transition system on *configurations*, giving an operational semantics of the language. Notice that rewrites rules are inherently nondeterministic and concurrent, which makes it easy and naturally to define semantics for nondeterministic/concurrent languages in 𝕂.

We emphasize two important characteristics of rewrites rules in 𝕂. The first is *local rewrites*, i.e., the rewrite symbol "`=>`" does not need to appear in the top level, but can appear locally in which the rewrite happens. Take as an example the rule that looks up the value of a program variable in the state. Instead of writing

```
rule <k> X:Id ...</k> <state>... X |-> I ...</state>
  => <k> I ...</k> <state>... X |-> I ...</state>
```

one writes

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

to not only reduce space but also avoid duplicates. The "`...`" has a special meaning in 𝕂. It stands for things "that exist but do not change in the rewrite". The rule, therefore, says that if a program variable `X:Id` is in the top of the computation in the `k` cell, and `X` binds to the integer `I` somewhere in the `state` cell, then rewrite `X:Id` to its value `I`, and do not change anything else.

The second characteristic of rewrite rules in 𝕂 is *configuration inference and completion*. The rewrite rules may not explicitly mention all cells in the configuration, but only mention related ones. 𝕂 will infer the implicit cells and compete the configuration automatically. For example, instead of writing

```
rule <T> <k> I1 + I2 => I1 +Int I2 ... </k> <state> M </state> </T>
```

one writes

```
rule I1 + I2 => I1 +Int I2
```

which is simpler. It is also more modular: if in the future we need to add a new cell to the configuration, then we do not need to modify the rules above, as the new cells can be inferred and completed by 𝕂 automatically. In fact, configuration inference and completion is one of the most important features that make 𝕂 definitions extensible and easy to adapt to language changes.

The rest of the semantics are self-explained. The rule for assignment `X = I:Int;`. updates the value that is bound to `X` in the `state` cell, as specified in the local rewrite `X |-> (_ => I)`. Here the underscore "`_`" is an anonymous 𝕂 variable. After the update, the assignment statement `X = I:Int;` is removed from the `k` cell, as specified by the local rewrite `X = I:Int; => .`. Recall that the dot "." means nothing, and rewriting something to a dot means removing it. Attribute `structural` means the associated rewrite rule is not counted as an explicit step by $\mathbb{K}$, but an implicit one. It should not affect how $\mathbb{K}$ executes the programs. The empty statement `{}` simply reduces to nothing. The last two rules process the declaration list of program variables and initialize their values to zero.

3 Program Execution and Verification in $\mathbb{K}$

Given the semantics of a programming language, $\mathbb{K}$ provides a variety of language tools, among which the most useful ones include a *parser*, an *interpreter* and a *verifier* for that language. In this section, we use the language IMP as a running example and show how to use these language tools that $\mathbb{K}$ offers, once we feed it the formal semantics (as in Fig. 6). For a more comprehensive introduction about $\mathbb{K}$ and $\mathbb{K}$ tools, we refer to the $\mathbb{K}$ framework website (https://runtimeverification.com/blog/k-framework-an-overview/).

3.1 Program Execution

The most basic tool that is automatically generated by $\mathbb{K}$ is a parser for IMP, based on the formal syntax definition. For example, the parser can parse IMP programs like the one in Fig. 7. Suppose that the $\mathbb{K}$ definition for IMP (Fig. 6) is saved in a file `imp.k`, the command `kompile imp.k` generates an *interpreter* for IMP which is invoked with the `krun` command. Suppose the IMP program `sum` is saved in a file `sum.imp`, then `krun sum.imp` executes the program and yields the final configuration as in Fig. 8. Notice that the `k` cell is empty, meaning that the program was completely executed, or consumed. In the end of its execution, `n` has the value 0 and `s` has the value 5050, which is the total of numbers up to 100, as expected. This execution capability of $\mathbb{K}$ is crucial for testing language semantics, and thus for increasing confidence in the adequacy of a language semantics. The above also illustrates another useful $\mathbb{K}$ tool, which like the parser generator, is used by almost any other tool, the $\mathbb{K}$ *unparser*. Indeed, the above configuration result uses concrete syntax to display the cells and their contents, although internally these are all represented as abstract data types. $\mathbb{K}$ also has the capability to display its results using abstract instead of concrete syntax, which helps users disambiguate in case the concrete syntax is ambiguous.

```
int s, n; n = 100; while(n) { s = s + n; n = n - 1; }
```

Fig. 7. The IMP program `sum` where `n` is initialized to 100.

We should point out that the interpreters automatically generated by $\mathbb{K}$ can be *very efficient*. For example, the formal semantics of the Ethereum Virtual

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

Fig. 8. The final configuration after executing the IMP program sum in Fig. 7.

Machine (EVM) bytecode language, one of the most popular virtual machine languages for the blockchain, yields an EVM interpreter that is only one order of magnitude slower than the reference C++ implementation of the EVM [11,13].

3.2 Program Verification

$\mathbb{K}$ aims to naturally support not only execution, but also full program verification, in an ideal, mathematically grounded manner. Therefore, we require a *fixed* logic with a *fixed* sound and (relatively) complete proof system, where all languages become theories in the logic, about which we can reason using the fixed proof system. In this scenario, program execution is just one particular proof for a certain reachability property (the initial configuration reaches the final configuration). The logic is fixed, so it does not depend on any particular programming language, very much unlike Hoare logic and its variants.

The logical foundation of $\mathbb{K}$'s verification infrastructure is reachability logic [5,6] for dynamic properties, which uses matching logic [19] for static properties. We refer interested readers to the mentioned references for more technical details. Here, we use the sum program as an example, showing how verification can be easily done in $\mathbb{K}$. The first step, of course, is to specify what properties about the program we want to prove. In Hoare logic, such specifications are given in terms of Hoare triples. In reachability logic and $\mathbb{K}$, specifications are written using the already existing $\mathbb{K}$ rule syntax.

```
module SUM_SPEC
  imports IMP

  rule     // invariant spec
      <k> while(n){ s = s + n; n = n - 1; } => .K ... </k>
      <state>
        n |-> (N:Int => 0)
        s |-> (S:Int => S +Int ((N +Int 1) *Int N /Int 2)
      </state>
  requires N >=Int 0

  rule     // main spec
      <k> int n, s; n = N:Int; while(n){ s = s + n; n = n - 1; } => .K </k>
      <state> .Map =>
        n |-> 0
        s |-> ((N +Int 1) *Int N /Int 2)
      </state>
  requires N >=Int 0
endmodule
```

Fig. 9. A functional specification of sum, consisting of two rules: a main one capturing the desired property, and an "invariant" one to be used as a lemma.

Figure 9 shows a specification of the `sum` program. The specification consists of two reachability claims, which follows the keyword `rule`. The second claim is the main specification, which says that the `sum` program (where `n` is now initialized to a symbolic value n, written as a $\mathbb{K}$ variable `N:Int`) will terminate (and thus reaches `.K` in the `k` cell), and when it terminates, the value of `s` equals $n(n+1)/2$. The condition after the keyword `requires` has the similar meaning of a pre-condition in Hoare logic. It asks $\mathbb{K}$ to prove the mentioned reachability claim given that $n \geq 0$. The first claim is provided as a *lemma*, known as the *invariant* of the while-loop, in order for $\mathbb{K}$ to prove the main claim. The invariant claim says that when $n \geq 0$, the while-loop will terminate, and the value of `s` will increased by $n(n+1)/2$.

What is interesting is how $\mathbb{K}$ establishes the invariant claim via a *circular proof*, based on reachability logic proof system. $\mathbb{K}$ starts with the configuration with a while-loop in the `k` cell and `n` mapping to n and `s` mapping to s, as required by the left-hand side of the claim. Then, $\mathbb{K}$ rewrites the configuration *symbolically*, following the semantics rules we defined in Fig. 6, so the while-loop will be de-sugared to an if-statement, and the two assignments are resolved accordingly, too. After that, $\mathbb{K}$ reaches a configuration which contains exactly the same while-loop in the `k` cell, but in the `state` cell `n` maps to $n-1$ and `s` maps to $s+n$. For clarity, let us denote *that* configuration as γ and let $n' = n-1$ and $s' = s+n$. At this point, the (Circularity) proof rule of the reachability logic proof system (see, e.g., [6]) takes effect, and the invariant claim itself becomes a regular *axiom* which can be used in further proofs. Therefore, we can *instantiate* the variables n and s in the invariant claim by n' and s', yielding exactly the configuration γ, and the invariant claim immediately tells us that γ will terminate at a state where `n` maps to 0 and `s` maps to $s' + n'(n'+1)/s$. And this tells us that the initial configuration, with `n` mapping to n and `s` mapping to s, can reach γ and then terminate at the same state. Finally, $\mathbb{K}$ calls SMT solvers (such as Z3 [8]) to prove that $s' + n'(n'+1)/2$ equals $s + n(n+1)/2$, and concludes the proof successfully.

4 Towards Language-Independent Runtime Verification

Runtime verification is a system analysis technique that extracts execution information from a running system and uses it to detect and react to observed behaviors satisfying or violating certain properties [7]. As it avoids complex traditional formal verification techniques and analyzes only a few system execution traces, runtime verification tools have good scalability on real-world projects and practical codebase, and thus has gained significant interest from the research community.

Typically speaking, runtime verification tools take a target system as input together with event specifications and desired properties, and yield as output a modified "monitored" system which checks the desired properties during execution and reacts in case of property violation. At present, a suite of runtime verification tools are available for many real-world languages, including RV-Match that checks undefined behavior of C programs [10], RV-Predict that checks data race for Java and C/C++ programs [4], and RV-Monitor that checks and enforces properties of Java and C programs [15], just to name a few.

Given the existing positive results that we have achieved in language-independent program execution and verification with the $\mathbb{K}$ framework, we propose a new promising direction towards *language-independent runtime verification*, where event specifications and desired properties are formally defined in the semantics and programs, and monitors are automatically generated in a correct-by-construction manner.

5 Conclusion

The $\mathbb{K}$ Framework was born from our firm belief that an ideal language framework is possible, where programming languages must have formal semantics, and that language tools such as parsers, interpreters, and deductive program verifiers are derived from just one reference formal definition of the language, at no additional cost specific to that language. $\mathbb{K}$ provides a user-friendly frontend (the meta-programming language) with which a variety of programming languages can be defined, while in its backend, a fixed language-independent logic powers $\mathbb{K}$'s deductive program verification. $\mathbb{K}$ may not be the final answer to this quest, but it proves that it is possible to have a language-independent program verification framework.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hahnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification-The KeY Book. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
3. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). https://doi.org/10.1007/10722010_8
4. Şerbănuţă, T.F., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent systems. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 136–150. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_16
5. Ştefănescu, A., Ciobâcă, Ş., Mereuţă, R., Moore, B.M., Şerbănută, T.F., Roşu, G.: All-path reachability logic. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 425–440. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_29
6. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics based program verifers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2016), pp. 74–91. ACM, November 2016

7. Daian, P., et al.: Runtime verification at work: a tutorial. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 46–67. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_5
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
10. Guth, D., Hathhorn, C., Saxena, M., Roşu, G.: RV-Match: practical semantics-based program analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 447–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_24
11. Hildenbrandt, E., et al.: KEVM: a complete semantics of the Ethereum virtual machine. In: Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF 2018). IEEE (2018). http://jellopaper.org
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
13. KEVM Team: KEVM: Semantics of EVM in K (2017). https://github.com/kframework/evm-semantics
14. Lamport, L.: The 'Hoare logic' of concurrent programs. Acta Inform. **14**(1), 21–37 (1980)
15. Luo, Q., et al.: RV-Monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_24
16. The Coq development team. The Coq proof assistant reference manual. LogiCal Project (2004)
17. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_17
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE (2002)
19. Roşu, G.: Matching logic. Log. Meth. Comput. Sci. **13**(4), 1–61 (2017)
20. The Isabelle development team: Isabelle (2018). https://isabelle.in.tum.de/
21. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. In: Concurrency and Computation: Practice and Experience, pp. 1173–1214 (2001)
22. Zhao, L., Wang, S., Liu, Z.: Graph-based object-oriented Hoare logic. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 374–393. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_23