Specifying Languages and Verifying Programs with K

http://kframework.org

Grigore Roşu
Department of Computer Science
University of Illinois
Urbana, USA
grosu@illinois.edu

***Abstract*—K is a rewrite-based executable semantic framework for defining languages. The K framework is designed to allow implementing a variety of generic tools that can be used with any language defined in K, such as parsers, interpreters, symbolic execution engines, semantic debuggers, test-case generators, state-space explorers, model checkers, and even deductive program verifiers. The latter are based on matching logic for expressing static properties, and on reachability logic for expressing dynamic properties. Several large languages have been already defined or are being defined in K, including C, Java, Python, Javascript, and LLVM.**

***Keywords*-semantics; verification; formal methods; logic**

I. Introduction

One of the long-lasting dreams of the programming language community is to have one formal semantic definition of a target programming language and from it to derive all the tools needed to execute and analyze programs written in that language: parsers, interpreters, compilers, symbolic execution engines, model checkers, deductive program verifiers, and so on. This is illustrated in Figure 1.

K [1] (`http://kframework.org`) is a rewrite-based executable semantic framework in which 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.

$\mathbb{K}$ is designed to allow implementing a variety of tools that can be used with any language semantics. Currently, it includes the ones mentioned in the abstract. The symbolic execution engine is connected to the Z3 SMT solver, and the deductive program verifier is based on matching logic for expressing static properties, which generalizes separation logic, and on reachability logic for expressing dynamic properties, which generalizes Hoare logic [2]–[7].

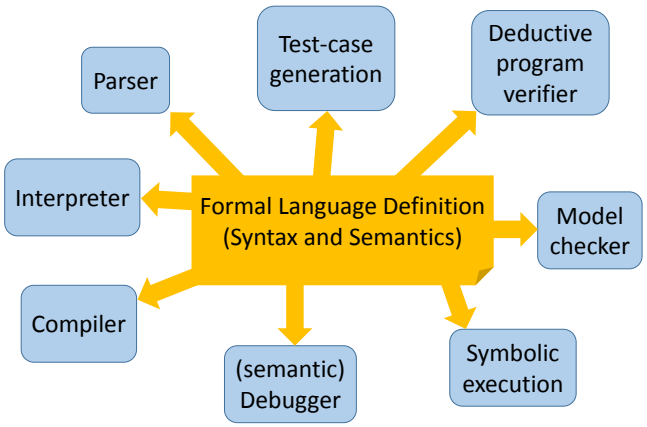

Figure 1. Not a dream anymore

A series of paradigmatic languages have been defined in $\mathbb{K}$ and are used for teaching at several universities. Also, several real-life large languages have been already defined or a still being defined using $\mathbb{K}$. All these can be reached from the URL mentioned above.

II. Why Semantics?

One may wonder why bother defining a language in $\mathbb{K}$, or in any other semantic framework. In fact, the current state of the art is to implement interpreters, compilers, and formal analysis tools specific to each language and not worry at all about any formal semantics for the language. This is not only uneconomical, because most of the tools reimplement the same techniques and algorithms, but it also quite error prone. Consider, for example, the following C program:

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

According to the C standard, this program is undefined. Yet, some compilers (gcc 4, msvc) and formal analysis tools (Havoc, Frama-C) execute or prove that this program evaluates to ... 4. What is wrong here is that these compilers and tools are not based on an explicit and public formal semantics of C, but instead have their own version of the semantics hardwired in their implementation. If an error in the semantics is found, then it needs to be fixed in each tool separately, which is tedious and demotivating.

Compare this to a world where all the tools are derived from a public formal semantics of the language, and have nothing particular to any language, except perhaps for cosmetic syntactic sugar for how to input formal knowledge to the tool. Then fixing an error in the semantics will fix the problem in all the tools. Moreover, tool developers have a much stronger incentive to engineer their tools well, because they become tools for all languages.

III. How $\mathbb{K}$ Works

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, $\mathbb{K}$ uses conventional BNF annotated with $\mathbb{K}$-specific attributes. For example, the syntax of assignment in a simple C-like imperative language 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, $\mathbb{K}$ proposes a cell-based approach. Each cell encapsulates relevant information for the semantics, including other cells that can "float" inside it. For a simple imperative 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 $\mathbb{K}$ 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 a possible K rule for assignment:

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

The ellipses are part of the $\mathbb{K}$ 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 unnecessary 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; its scoping can therefore be controlled using parentheses. 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 $\mathbb{K}$ 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. We do not discuss these options here.

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 $\mathbb{K}$ tool distribution, and several others are available from the mentioned URL. Some of these languages have dozens of cells in their configurations and hundreds of semantic rules.

IV. Current Progress, Applications, Further Reading

Besides didactic and prototypical languages (such as the lambda calculus, System F, and Actors), $\mathbb{K}$ has been used to formalize several existing real-life languages and to design and develop analysis and verification tools for them. For example, C [8], Python [9], Java [10] and Scheme [11], as well as various aspects of features of Haskell [12], Javascript [13], X10 [14], a RISC assembly [15], [16], LLVM [17], and a framework for domain specific languages [18], [19].

$\mathbb{K}$'s ability to express truly concurrent computations has been used in researching safe models for concurrency [20], synchronization of agent systems [21], models for P-Systems [22], [23], and for the x86-TSO relaxed memory model [24]. $\mathbb{K}$ has been used for designing type checkers/inferencers [25], for model checking executions with predicate abstraction [26], [27] and heap awareness [28], for symbolic execution [29]–[31], computing worst case execution times [32], [33], studying program equivalence [34], and runtime verification [24], [35]. Additionally, the C definition mentioned above has been used as a program undefinedness checker to analyze C programs [36].

$\mathbb{K}$ served as an inspiration for the design of Reachability Logic [6], [7], a new logic for verification based on matching logic [2], unifying operational and axiomatic semantics [4], generalizing both Hoare logic and separation logic [5], which serves as basis for a new program verification tool for $\mathbb{K}$ definitions using Hoare-like assertions [3].

All these definitions and analysis tools can be found on the $\mathbb{K}$ tool website. Other language definitions and analysis tools developed using the $\mathbb{K}$ technique before the development of the $\mathbb{K}$ tool include early definitions of Java [37] and of Verilog [38], as well as a static policy checker for C [39].

We recommend the reader who wants to learn $\mathbb{K}$ to start by downloading the tool from its website, do the online filmed tutorial, and then do the exercises in the $\mathbb{K}$ distribution. In terms of paper reading, we recommend the recent $\mathbb{K}$ overview [40] for a high-level presentation and a moderate example, and the $\mathbb{K}$ primer [41] for tool-specific details.

V. Conclusion

$\mathbb{K}$ is a framework for defining programming languages. It 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.

Acknowledgment

The development of $\mathbb{K}$ would have not been possible without the enthusiasm and support of many colleagues, students and friends. I would like to particularly thank Dorel Lucanu, who is leading the Romanian team that co-develops the $\mathbb{K}$ tool (together with the team at Urbana, USA), and to Traian Florin Serbanuta, for his leadership of the development team. The research underlying the $\mathbb{K}$ framework was supported in part by the NSF grant CCF-1218605, the DARPA HACMS program as SRI subcontract 19-000222, the Rockwell Collins contract 4504813093, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

[1] G. Rosu and T.-F. Serbanuta, "An overview of the K semantic framework," *J. Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[2] G. Rosu, C. Ellison, and W. Schulte, "Matching logic: An alternative to Hoare/Floyd logic," in *AMAST*, ser. LNCS, vol. 6486, 2010, pp. 142–162.

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

[4] ——, "Towards a unified theory of operational and axiomatic semantics," in *ICALP*, ser. LNCS, vol. 7392, 2012, pp. 351–363.

[5] ——, "From Hoare logic to matching logic reachability," in *FM*, ser. LNCS, vol. 7436, 2012, pp. 387–402.

[6] ——, "Checking reachability using matching logic," in *OOPSLA*. ACM, 2012, pp. 555–574.

[7] G. Roşu, A. Ştefănescu, S. Ciobâcă, and B. M. Moore, "One-path reachability logic," in *LICS'13*. IEEE, 2013.

[8] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *POPL*. ACM, 2012, pp. 533–544.

[9] 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

[10] D. Bogdanas, "K definition of Java 1.4," 2013. [Online]. Available: https://github.com/kframework/java-semantics

[11] P. Meredith, M. Hills, and G. Rosu, "An executable rewriting logic semantics of K-Scheme," in *SCHEME*, D. Dubé, Ed. Laval University TR DIUL-RT-0701, 2007, pp. 91–103. [Online]. Available: http://www.schemeworkshop.org/2007/procFront.pdf

[12] D. Lazar, "K definition of Haskell'98," 2012. [Online]. Available: https://github.com/davidlazar/haskell-semantics

[13] D. Park, "K definition of Javascript," 2013. [Online]. Available: https://github.com/kframework/javascript-semantics

[14] 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.

[15] M. Asavoae, "K semantics for assembly languages : A case study," in *K'11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., 2013, in this issue.

[16] ——, "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

[17] C. Ellison and D. Lazar, "K definition of the LLVM assembly language," 2012. [Online]. Available: https://github.com/davidlazar/llvm-semantics

[18] V. Rusu and D. Lucanu, "A K-based formal framework for domain-specific modelling languages," in *FoVeOOS*, ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, and D. Gurov, Eds., vol. 7421. Springer, 2011, pp. 214–231.

[19] ——, "K semantics for OCL—a proposal for a formal definition for OCL," in *K'11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., 2013, in this issue.

[20] 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.

[21] P. Dinges and G. Agha, "Scoped synchronization constraints for large scale actor systems," in *COORDINATION*, ser. Lecture Notes in Computer Science, M. Sirjani, Ed., vol. 7274. Springer, 2012, pp. 89–103.

[22] T. F. Şerbănuţă, G. Ştefănescu, and G. Roşu, "Defining and executing P systems with structured data in K," in *WMC*, ser. Lecture Notes in Computer Science, D. W. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, Eds., vol. 5391. Springer, 2008, pp. 374–393.

[23] C. Chira, T.-F. Serbanuta, and G. Stefanescu, "P systems with control nuclei: The concept," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 326–333, 2010.

[24] 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.

[25] C. Ellison, T. F. Serbanuta, and G. Rosu, "A rewriting logic approach to type inference," in *WADT*, ser. Lecture Notes in Computer Science, A. Corradini and U. Montanari, Eds., vol. 5486. Springer, 2008, pp. 135–151.

[26] I. M. Asavoae and M. Asavoae, "Collecting semantics under predicate abstraction in the K framework," in *WRLA*, ser. Lecture Notes in Computer Science, P. C. Ölveczky, Ed., vol. 6381. Springer, 2010, pp. 123–139.

[27] I. M. Asavoae, "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

[28] J. Rot, I. M. Asavoae, 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.

[29] I. M. Asavoae, M. Asavoae, 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.

[30] I. M. Asavoae, "Abstract semantics for alias analysis in K," in *K'11*, ser. Electronic Notes in Theoretical Computer Science, M. Hills, Ed., 2013, in this issue.

[31] 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.

[32] M. Asăvoae, D. Lucanu, and G. Roşu, "Towards semantics-based WCET analysis," in *WCET*, ser. Austian Computer Society (OCG), C. Healy, Ed., 2011.

[33] M. Asavoae, I. M. Asavoae, and D. Lucanu, "On abstractions for timing analysis in the K framework," in *FOPARA*, ser. Lecture Notes in Computer Science, 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

[34] D. Lucanu and V. Rusu, "Program Equivalence by Circular Reasoning," INRIA, Tech. Rep. RR-8116, October 2012. [Online]. Available: http://hal.inria.fr/hal-00744374

[35] G. Rosu, W. Schulte, and T.-F. Serbanuta, "Runtime verification of C memory safety," in *RV*, 2009, pp. 132–151.

[36] 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.

[37] A. Farzan, F. Chen, J. Meseguer, and G. Rosu, "Formal analysis of Java programs in JavaFAN," in *CAV'04*, ser. LNCS, vol. 3114, pp. 501–505.

[38] P. O. Meredith, M. Katelman, J. Meseguer, and G. Rosu, "A formal executable semantics of Verilog," in *MEMOCODE'10*. IEEE, 2010, pp. 179–188.

[39] M. Hills, F. Chen, and G. Rosu, "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.

[40] G. Rosu and T. F. Serbanuta, "K overview and simple case study," in *Proceedings of International K Workshop (K'11)*, ser. ENTCS. Elsevier, 2013, To appear.

[41] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu, "The k primer (version 3.3)," in *Proceedings of International K Workshop (K'11)*, ser. ENTCS. Elsevier, 2013, To appear.