Rewriting Logic Semantics of Programming Languages: Further Reading

CS 542
Lecture 21
José Meseguer

1. Broadening the Picture: Rewriting Logic Semantics of Hardware Description Languages and Software Modeling Languages

Computational systems can be described at several levels, including, from most concrete to most abstract:

- the hardware on which they run
- the programming language in which they are written
- the software modeling language in which they are designed.

Everybody agrees that having precise formal descriptions at the hardware and programming language levels is of great importance.

Software modeling languages — the so-called model-based software engineering — are quite useful to design systems, and even to generate code from models. However, ~~its~~ their Achilles heel has been that:

1. Many modeling languages lack a precise formal semantics and therefore,
2. Debugging designs at the moding language level is hard to do or impossible, while design errors are orders of magnitude more costly than coding errors,

3. The lack of a precise formal semantics also means that any formal verification is utter nonsense in the absence of such a semantics.

4. Under the above circumstances, code generation from models obviously does something, i.e., produces some code. But there is an utter lack of even any criteria for correctness: the reliability of the resulting code is anybody's guess: it is whatever the code generator spits out.

For the above reasons 1-4, it is actually quite useful to give formal semantics to software modeling languages, ~~in~~ in rewriting logic since then:

1. software models of system designs can be executed and debugged before they are implemented
2. they can also be formally verified to check that they meet formal correctness requirements, for example by model checking, and
3. such a semantics can be used as the basis for correct-by-construction code generation.

The Rewriting Logic Semantics Project is not restricted to programming languages: it has proved very useful also to give semantics to hardware description languages and to software modeling languages, and to build tools to

execute system descriptions at those three levels. The paper:

J. Meseguer and G. Roşu, "The rewriting logic semantics project: A progress report," Information & Computation 231, 38-69, 2013 [in the CS 524 course web page]

gives an overview as of 2013 of the entire rewriting logic semantics project, with special emphasis on both programming languages and software modeling languages.

Also, as of 2012, section 4.4 of the paper "Twenty Years of Rewriting logic" [also in the CS 524 web page] gives a list of references for various software modeling languages that have been given semantics in rewriting logic.

Some examples of such languages include:

- E-LOTOS
- MOF
- UML
- AADL [for real-time and cyber-physical systems]
- Ptolemy [" " " " " "]

for which their formal semantics in Maude allow the prototyping, debugging, and formal analysis advantages (1-3) described above.

For hardware description languages three references worth looking at about rewriting logic semantics include:

- P. O'Neil Meredith, M. Katelman, J. Meseguer, G. Roşu, "A formal executable semantics of Verilog," in Proc. MEMOCODE 2010, 179-188, 2010.

- M. Katelman, "A Meta-Language for Functional Verification", Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, available at UIUC's IDEALS repository.

- M. Katelman, S. Keller, J. Meseguer, "Rewriting semantics of production rule sets," J. Logic & Algebraic Methods in Programming, 81, 929-956, 2012 [semantics of an asynchronous hardware description language].

2. Using the Executable Rewriting Logic Semantics of a Programming, resp. Hardware Description, resp. Software Modeling Language to Formaly Analyze Programs, resp. Hardware Designs, resp. Sofware Models

2.1. Execution, Prototyping, Debugging: Having an executable specification of the semantics allows early exploration of software models, as well as execution and debugging of programs and hardware descriptions.

2.2 Reachability Analysis: Properties such as invariants can be verified [at least up to a depth bound if the number of reachable states is not finite] using the search command [or narrowing search for

symbolic reachability analysis from an infinite set of initial states described by one or more patterns.

2.3 Temporal Logic Model Checking. More sophisticated temporal logic properties in both Linear-Time Temporal Logic (LTL), and the Linear-Time Temporal Logic of Rewriting (LTLR) can be verified of the program/hardware design/software model using Maude's LTL and LTLR model checker; or for real-time programming languages or software models, Real-Time Maude's tool model checker.

2.4 The programs language/hardware description language/software modeling language program or design can also be verified by theorem proving, reasoning over the rewrite theory $\mathcal{R}_{\mathcal{L}}$ giving semantics to $\mathcal{L}$.

This can, and has been, done using Reachability Logic, and Matching Logic, for specifications in both $\mathbb{K}$ and Maude. See, for example:

A. Stefanescu, A. Ciobaca, S. Radu, M. Radu, B. Moore, T. Serbanuta, G. Roşu, "All-Path Reachability Logic," Logical Methods in Computer Science, 15, 2019

S. Skeirik, A. Stefanescu, J. Meseguer, "A Constructor-Based Reachability Logic for Rewrite Theories," Fundamenta Informaticae, 173, 315-382 (2020).

3. One, Two, a Hundred Thousand Operational Semantics

Since the rewriting logic semantics $\mathcal{R}_{\mathcal{L}}$ has a corresponding category $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$ of concurrent computations, it automatically provides a mathematical semantics, i.e., a mathematical model, for $\mathcal{L}$, namely, the category

$$\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$$

But $\mathcal{R}_{\mathcal{L}}$ is also executable, for example in Maude. And, as such provides an executable operational semantics for $\mathcal{L}$. The executable part is very important. Many operational semantics definitions are what might be called paper semantics. For a complex language this means that:

(1) it is not even clear whether they are correct, and
(2) they cannot be compared with ~~an~~ actual implementations for correctness purposes.

For a large language like C which was never fully specified formally before Ellison and Roşu specified it completely in their K-Maude tool, as reported in their POPL 2012 paper, this is crucial: having an executable semantics in Maude [after translating from K to Maude in K-Maude] meant that the could

compare their C-semantics with the results provided by the GNU C-compiler [they passed more tests than the GNU compilers in its "Torture test suite"]. Furthermore, the uncovered bugs in various C tools, including theorem provers. The current K specification of C is also routinely tested against many such test cases.

But one important question is:

> How do rewriting logic semantic definitions compare with the usual operation semantics definition styles?

The long and short of this question, answered in the paper:
T. Serbanuta, G. Roşu, J. Meseguer, "A rewriting logic approach to operational semantics," Information and Computation, 207, 305-340 [available in the CS 524 web page]
is that all the usual operational semantics definition styles, including:

- Big step operational semantics
- Small step operational semantics
- Modular Operational Semantics (MSOS)
- Reduction semantics
- Continuation-based semantics, and
- Chemical Abstract Machine (CHAM) semantics

can very naturally be expressed in rewriting logic. That is, rewriting logic does not impose a specif style of OS definitions. Instead:

Rewriting Logic provides a Semantic Framework in which:

1. Any definitional style of choice can be used, and
2. New definitional styles can be developed.

Of course, pragmatic issues, such as efficiency, matter in such choices [something invisible at the level of a "paper semantics"]. For example, a continuation-based semantics is orders of magnitude more efficient than a small-step operational semantics.

4. From Programming Language Definitions to Correct-by-Construction Distributed Language Implementations

The rewriting logic semantics $\mathcal{R}_{\mathcal{L}}$ of a language $\mathcal{L}$, when specified in Maude automatically yields a sequential interpreter for $\mathcal{L}$. But if $\mathcal{L}$ is a concurrent language, then $\mathcal{L}$'s concurrency is not exploited: it is only simulated.

The situation may even be worse for a language, like J. Misra's Orc, that is both concurrent and has

real-time features. Because then $\mathcal{R}_{\mathcal{L}}$ will not only simulate concurrency: it will also simulate time, for example, by executing and analyzing programs in $\mathcal{L}$ in the Real-Time Maude tool.

1. So, what can be done to pass from $\mathcal{R}_{\mathcal{L}}$ to an actual distributed and even real-time implementation?
2. And how is the correct semantics defined in $\mathcal{R}_{\mathcal{L}}$ preserved in its actual distributed implementation?

Here are some good news about the answer that can be given to questions (1)-(2) above:

1. Maude has rule-programmable external objects, including (i) I/O TCP/IP sockets, (ii) OS processes, and (iii) I/O files so that an object-based rewrite theory $\mathcal{R}$ can be transformed and deployed into a corresponding distributed implementation $\mathcal{D}(\mathcal{R})$.
2. The theory transformation $\mathcal{R} \mapsto \mathcal{D}(\mathcal{R})$ can be performed inside Maude, and has been proved correct-by-construction is:

S. Liu, A. Sandur, J. Meseguer, P. Ölveczky, Q. Wang, "Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs," Proc. NASA Formal Methods 2020, Springer LNCS, 1229, 22-40, 2020

In fact, the above paper proves that $\mathcal{R}$ and $\mathcal{D}(\mathcal{R})$ are bisimilar systems, and therefore satisfy the same temporal logic properties. That is, $\mathcal{D}(\mathcal{R})$ is correct by construction.

3. This methodology [except for the $\mathcal{R} \mapsto \mathcal{D}(\mathcal{R})$ transformation that is more recent] has already been applied to obtain two distributed implementations of concurrent programming languages, namely:

(3.1) The Distributed KLAIM Implementation, reported in:

[available in CS 524 web page] → J. Eckhard, T. Mühlbauer, J. Meseguer and M. Wirsing, "Semantics, distributed implementation, and formal analysis of KLAIM models in Maude," Science of Computer Programming 99, 24-74 (2015)

(3.2) The distributed and real-time implementation of the Orc language, reported in:

M. AlTurki and J. Meseguer, "Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis", in Proc. 12th International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS'10), EPTCS 36, 26-45, 2010 [also available in the CS 524 web page].