Computational Logical Frameworks and Generic Program Analysis Technologies

José Meseguer and Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
{meseguer,grosu}@cs.uiuc.edu

1 Motivation

The technologies developed to solve the verifying compiler grand challenge should be *generic*, that is, not tied to a particular language but widely applicable to many languages. Such technologies should also be *semantics-based*, that is, based on a rigorous formal semantics of the languages.

For this, a *computational logical framework* with efficient executability and a spectrum of *meta-tools* can serve as a basis on which to: (1) define the formal semantics of any programming language; and (2) develop *generic program analysis techniques and tools* that can be instantiated to generate powerful analysis tools for each language of interest.

Not all logical frameworks can serve such purposes well. We first list some specific requirements that we think are important to properly address the grand challenge. Then we present our experience with rewriting logic as supported by the Maude system and its formal tool environment. Finally, we discuss some future directions of research.

2 Logical Framework Requirements

Based on experience, current trends, and the basic requirements of the grand challenge problem, we believe that any logical framework serving as a computational infrastructure for the various technologies for solving the grand challenge should have *at least* the following features:

1. good *data representation* capabilities,
2. support for *concurrency and nondeterminism*,
3. *simplicity* of the formalism,
4. *efficient* implementability, and efficient *meta-tools*,
5. support for *reflection*,
6. support for inductive reasoning, preferably with *initial model* semantics,
7. support for generation of *proof objects*, acting as correctness certificates.

While proponents of a framework may claim that it has all these features, in some cases further analysis can show that it either lacks some of them, or can only "simulate" certain features in a quite artificial way. A good example is the simulation/elimination of concurrency in inherently deterministic formalisms by implementing or defining thread/process scheduling algorithms. Another example might be the claim that the lambda calculus has good data representation capabilities because one can encode numbers as Church numerals.

3 The Rewriting Logic/Maude Experience

At UIUC, together with several students, we are developing semantic definitions of programming languages based on *rewriting logic* (RWL) [27]. Rewriting logic meets the requirements mentioned above, and supports semantic definitions of programming languages that combine *algebraic denotational semantics* and *SOS* semantics in a seamless way [29]. Given a language L, its rewriting logic semantics is a rewrite theory

$$\mathcal{R}_L = (\Sigma_L, E_L, R_L),$$

where Σ_L is a *signature* expressing the syntax of L, E_L is a set of *equations* defining the meaning of the sequential features of L together with that of the various state infrastructure operations, and R_L is a set of *rewrite rules* defining the semantics of the concurrent features of L.

3.1 Maude and Its Formal Tools

Rewrite theories are triples (Σ, E, R), with (Σ, E) an equational theory and R a set of rewrite rules. Intuitively, (Σ, E, R) specifies a computational system in which the *states* are specified as elements of the algebraic data type defined by (Σ, E), and the system's *concurrent transitions* are specified by the rewrite rules R. Rewrite theories can be executed in different languages such as CafeOBJ [22], and ELAN [1]. The most general support for the execution of rewrite theories is currently provided by the Maude language [6,7], in which rewrite theories with very general conditional rules, and whose underlying equational theories can be membership equational theories [28], can be specified and can be executed, provided they satisfy some basic executability requirements. Furthermore, Maude provides very efficient support for rewriting *modulo* any combination of associativity, commutativity, and identity axioms. Since an equational theory (Σ, E) can be regarded as a degenerate rewrite theory of the form $(\Sigma, E, \emptyset)$, equational logic is naturally a sublogic of rewriting logic. In Maude this sublogic is supported by *functional modules* [6], which are theories in membership equational logic. When executed in Maude, the RWL formal semantics $\mathcal{R}_L$ of language L automatically becomes an *efficient interpreter* for L: for example, faster than the Linux bc interpreter, and half the speed of the Scheme interpreter.

Besides supporting efficient execution, often in the order of several million rewrites per second, Maude also provides a range of formal tools and algorithms to analyze rewrite theories and verify their properties. These tools can be used almost directly to provide corresponding analysis tools for languages defined as rewrite logic theories. A first useful formal analysis feature is its *breadth-first search* command. Given an initial state of a system (a term), we can search for all reachable states matching a certain pattern and satisfying an equationally-defined semantic condition P. By making $P = \neg Q$, where Q is an invariant, we get in this way a *semi-decision procedure* for finding failures of invariant safety properties. Note that there is no finite-state assumption involved here: any executable rewrite theory can thus be analyzed. For systems where the set of states reachable from an initial state are finite, Maude also provides a linear time temporal logic (LTL) model checker. Maude's is an explicit-state LTL model checker, with performance comparable to that of the SPIN model checker [24] for the benchmarks that we have analyzed [17,18].

Reflection is a key feature of rewriting logic, and is efficiently supported in the Maude implementation through its `META-LEVEL` module. One important fruit of this is that it becomes quite easy to build new formal tools and to add them to the Maude environment. Indeed, such tools by their very nature manipulate and analyze rewrite theories. By reflection, a rewrite theory $\mathcal{R}$ becomes a *term* $\overline{\mathcal{R}}$ in the universal theory, which can be efficiently manipulated by the descent functions in the `META-LEVEL` module. As a consequence, Maude formal tools have a reflective design and are built in Maude as suitable extensions of the `META-LEVEL` module. They include the following:

– the Maude Church-Rosser Checker, and Knuth-Bendix and Coherence Completion tools [8,15,13,12]
– the Full Maude module composition tool [11,16]
– the Maude Predicate Abstraction tool [34]
– the Maude Inductive Theorem Prover (ITP) [5,8,9]
– the Real-Time Maude tool [33];
– the Maude Sufficient Completeness Checker (SCC) [23]
– the Maude Termination Tool (MTT) [14].

3.2 Unifying SOS and Equational Semantics

For the most part, equational semantics[1] and SOS have lived separate lives. Pragmatic considerations and differences in taste tend to dictate which framework is adopted in each particular case. For concurrent languages SOS is clearly superior and tends to prevail as the formalism of choice, but for deterministic languages equational approaches are also widely used. Of course there are also practical considerations of tool support for both execution and formal reasoning.

In the end, equational semantics and SOS, although each very valuable in its own way, are "single hammer" approaches. Would it be possible to seamlessly *unify* them within a more flexible and general framework? Could their respective limitations be overcome when they are thus unified? Our proposal is that

[1] In equational semantics, formal definitions take the form of *semantic equations*, typically satisfying the *Church-Rosser* property. Both higher-order (denotational semantics) and first-order (algebraic semantics) versions have been shown to be useful formalisms. We use the more neutral term *equational semantics* to emphasize the fact that denotational and algebraic semantics have many common features and can both be viewed as instances of a common equational framework.

rewriting logic does indeed provide one such unifying framework. The key to this unification is what we call rewriting logic's *abstraction knob*. The point is that in equational semantics' model-theoretic approach entities are *identified by the semantic equations*, and have unique *abstract denotations* in the corresponding models. In our knob metaphor this means that in equational semantics the abstraction knob is *always turned all the way up to its maximum position.* By contrast, one of the key features of SOS is providing a very detailed, step-by-step formal description of a language's evaluation mechanisms. As a consequence, most entities —except perhaps for built-in data, stores, and environments, which are typically treated on the side— are *primarily syntactic*, and computations are described in full detail. In our metaphor this means that in SOS the abstraction knob is *always turned down to its minimum position.*

How is the unification and corresponding availability of an abstraction knob achieved? Since a rewrite theory is a triple (Σ, E, R), with (Σ, E) an equational theory with Σ a signature of operations and sorts, and E a set of (possibly conditional) equations, and with R a set of (possibly conditional) rewrite rules, equational semantics is obtained as the special case in which $R = \emptyset$, so we only have the semantic equations E and the abstraction knob is turned up to its maximum position. SOS is obtained as the special case in which $E = \emptyset$, and we only have (possibly conditional) rules R rewriting purely syntactic entities (terms), so that the abstraction knob is turned down to the minimum position.

Rewriting logic's "abstraction knob" is precisely its crucial distinction between equations E and rules R in a rewrite theory (Σ, E, R). *States of the computation* are then E-equivalence classes, that is, *abstract elements* in the initial algebra $T_{\Sigma/E}$. A rewrite with a rule in R is understood as a transition $[t] \longrightarrow [t']$ between such abstract states. The knob, however, can be turned up or down. We can turn it *all the way down to its minimum* by converting all equations into rules, transforming (Σ, E, R) into $(\Sigma, \emptyset, R \cup E)$. This gives us the most concrete, SOS-like semantic description possible. Can we turn the knob "all the way up," in the sense of converting all rules into equations? Only if the system we are describing is *deterministic* (for example, the semantic definition of a sequential language) is this a sound procedure. In that case, the equational theory $(\Sigma, R \cup E)$ should be Church-Rosser, and we do indeed obtain a most-abstract-possible, purely equational semantics out of the less abstract specification (Σ, E, R), or even out of the most concrete possible specification $(\Sigma, \emptyset, R \cup E)$. What can we do in general to make a specification *as abstract as possible*? We can identify a subset $R_0 \subseteq R$ such that: (1) $R_0 \cup E$ is Church-Rosser; and (2) R_0 is biggest possible with this property. In actual language specification practice this is not hard to do. Essentially, we can use semantic equations for most of the sequential features of a programming language: only when interactions with memory could lead to nondeterminism (particularly if the language has threads, or they could later be added to the language in an extension) or for intrinsically concurrent features, are rules (as opposed to equations) really needed. In our experience, it is often possible to specify most of the semantic axioms with equations, with relatively few rules needed for truly concurrent or nondeterministic features. For example, the semantics of the JVM described in [21,19] has about 300 equations and 40 rules; and that of Java described in [19] has about 600 equations but only 15 rules. A semantics for an ML-like language with threads given in [30] has only two rules.

This distinction between equations and rules, besides giving to equational semantics and SOS their due in a way not possible for the other alternative if we were to remain within each of these formalisms, has also important practical consequences for *program analysis*; because it affords a massive *state space reduction* which can make formal analyses such as breadth-first search and model checking enormously more efficient. Because of state-space explosion, such analyses could easily become infeasible if we were to use an SOS-like specification in which all computation steps are described with rules. This capacity of dealing with abstract states is a crucial reason why our generic tools, when instantiated to a given programming language definition, tend to result in program analysis tools of competitive performance. Of course, the price to pay in exchange for abstraction is a *coarser level of granularity* in respect to what aspects of a computation are *observable* at that abstraction level. For example, when analyzing a sequential program using a semantics in which most sequential features have been specified with equations, all sequential subcomputations will be abstracted away, and the analysis will focus on memory and thread interactions. If a finer analysis is needed, we can always obtain it by "turning down the abstraction knob" to the right observability level by *converting some equations into rules.* That is, we can regulate the knob to find for each kind of analysis the best possible balance between abstraction and observability.

3.3 Languages Defined in Rewriting Logic

Many languages have already been given semantics in this way using Maude. The language definitions can then be used as interpreters, and —in conjunction with Maude's search command and its LTL model checker— to formally analyze programs in those languages. For example, large fragments of Java and the JVM have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [21,19]. A similar Maude specification of the semantics of Scheme at UIUC yields an interpreter with .75 the speed of the standard Scheme interpreter on average for the benchmarks tested. The specification of a C-like language and the corresponding formal analyses are discussed in detail in [31]. A semantics of an ML-like language with threads was discussed in detail in [30], a modular rewriting logic semantics of CML has been given in [4], and a definition of the Scheme language has been given in [10]. Other language case studies, all specified in Maude, include: BC [2], CCS [43,44,2], CIAO [40], Creol [25], ELOTOS [42], MSR [3,38], PLAN [39,40], and the pi-calculus [41]. In fact, the semantics of large fragments of conventional languages are by now routinely developed by UIUC graduate students as course projects [36] in a few weeks, including, besides the languages already mentioned: Beta, Haskell, Lisp, LLVM, Pict, Python, Ruby, and Smalltalk.

3.4 Formal Analysis

Furthermore, Maude's formal tools, such as its inductive theorem prover, linear temporal logic (LTL) model-checker, and breadth-first search (BFS) capability then become *meta-tools* from which we derive useful program analysis tools for L using $\mathcal{R}_L$.

We are furthermore developing new *generic program analysis technologies* such as, for example, a *generic partial-order reduction* technique [20] than can apply to any language L with threads, and does not require any changes to an underlying model checker.

Correctness of a compiler can and should mean more than just correctness with respect to functional behavior. Depending upon particular applications of interest, certain important safety policies that transcend the basic semantics of the language under consideration may need to be preserved. For example, in an application referring to physical objects, consistency with respect to units of measurement or coordinate systems needs to be assured. We are also developing *domain-specific certifiers*, which are static analysis tools that check conformance of computation with respect to particular but important domains of interest. For example, we developed RWL-based certifiers for conformance with units of measurement [37], and with coordinate frames [26].

The *cost* of generating tools for a language L this way using its formal semantic definition $\mathcal{R}_L$ is *much lower* (in the order of weeks) than that of building similar language specific analysis tools (man years). For example, it took Feng Chen at UIUC only a few weeks to develop the formal semantics of Java 1.4 (except for its libraries) as a RWL theory $\mathcal{R}_{Java}$ specified in Maude.

Furthermore, the *formal analysis tools* obtained for free from $\mathcal{R}_{Java}$ and $\mathcal{R}_{JVM}$ are *competitive* for some applications with similar language-specific tools such a NASA-Ames' Java Path Finder [45] and Stanford's Java model checker [35]. Similarly, our experiments with the generic partial order reduction technique indicate that it can achieve rates of space and time reduction similar to those of language-specific tools such as SPIN [24].

4 Future Directions Related to the Grand Challenge

Our main point has been to emphasize the need for genericity in approaching the grand challenge; otherwise, an answer to the challenge would have a limited applicability to other languages besides those chosen in the challenge project. For this, we have argued that both a computational logical framework in which to give a precise formal semantics to programming languages, and on which to base generic program analysis tools, would be very useful.

We have also summarized our experience so far with one such logical framework, namely rewriting logic, and for applying the Maude RWL language and its generic tools to formally analyze programs in different programming languages. Our results, although encouraging, are very much *work in progress*; we would like to advance in addition the following directions:

1. Modular programming language definitions in the spirit of MSOS [32]. The goal is to build a database of reusable semantic definitions, with the semantics of each language feature defined in a separate module. It should then be possible to define the semantics of a whole language by just combining the modules for the language features, renaming the syntax of each module to the chosen concrete syntax.
2. Developing various language-generic program analysis tools; we have already mentioned the ongoing work on partial order reduction, which should be further advanced; but generic abstraction tools, and also generic tools for static analysis are two other important areas to advance.
3. Language-generic theorem proving environments, based on an axiomatic semantics that uses the language rewriting semantics as its foundation are also an important direction to investigate.
4. Finally, it would be useful to investigate semantics-preserving translations between languages, in particular the generation of provably correct compilers from the formal semantics $\mathcal{R}_L$ of a language L.

References

1. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E.: ELAN from a rewriting logic point of view. Theoretical Computer Science 285, 155–185 (2002)
2. Braga, C., Meseguer, J.: Modular rewriting semantics in practice. In: Proc. WRLA 2004, ENTCS
3. Cervesato, I., Stehr, M.-O.: Representing the msr cryptoprotocol specification language in an extension of rewriting logic with dependent types. In: Degano, P. (ed.) Proc. Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004), Barcelona, Spain, March 27 - 28, 2004, Elsevier ENTCS, Amsterdam (2004)
4. Chalub, F., Braga, C.: A Modular Rewriting Semantics for CML. Journal of Universal Computer Science 10(7), 789–807 (2004), `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`
5. Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. In: CSLI Publications (2000)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243 (2002)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.0 Manual (June 2003), `http://maude.cs.uiuc.edu`
8. Clavel, M., Durán, F., Eker, S., Meseguer, J.: Building equational proving tools by reflection in rewriting logic. In: CAFE: An Industrial-Strength Algebraic Formal Method, Elsevier, Amsterdam (2000), `http://maude.cs.uiuc.edu`
9. Clavel, M., Palomino, M.: The ITP tool's manual. Universidad Complutense, Madrid (April 2005), `http://maude.sip.ucm.es/itp/`
10. d'Amorim, M., Roşu, G.: An Equational Specification for the Scheme Language. In: Proceedings of the 9th Brazilian Symposium on Programming Languages (SBLP 2005), 2005. Also Technical Report No. UIUCDCS-R-2005-2567 (April 2005) (to appear)

11. Durán, F.: A reflective module algebra with applications to the Maude language. Ph.D. Thesis, University of Málaga (1999)
12. Durán, F.: Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International (2000), `http://maude.cs.uiuc.edu/papers`
13. Durán, F.: Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International (2000), `http://maude.cs.uiuc.edu/papers`
14. Durán, F., Lucas, S., Meseguer, J., Marché, C., Urbain, X.: Proving termination of membership equational programs. In: Sestoft, P., Heintze, N. (eds.) Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM 2004, pp. 147–158. ACM Press, New York (2004)
15. Durán, F., Meseguer, J.: A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International (2000), `http://maude.cs.uiuc.edu/papers`
16. Durán, F., Meseguer, J.: On parameterized theories and views in Full Maude 2.0. In: Futatsugi, K. (ed.) Futatsugi, editor, Proc. 3rd. Intl. Workshop on Rewriting Logic and its Applications. ENTCS, Elsevier, Amsterdam (2000)
17. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2002)
18. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 230–234. Springer, Heidelberg (2003)
19. Farzan, A., Cheng, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
20. Farzan, A., Meseguer, J.: Partial order reduction for rewriting semantics of programming languages, Manuscript, submitted for publication (2005)
21. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM code analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
22. Futatsugi, K., Diaconescu, R.: CafeOBJ Report. World Scientific, AMAST Series (1998)
23. Hendrix, J., Meseguer, J., Clavel, M.: A sufficient completeness reasoning tool for partial specifications. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 165–174. Springer, Heidelberg (2005)
24. Holzmann, G.: The Spin Model Checker - Primer and Reference Manual. Addison-Wesley, Reading (2003)
25. Johnsen, E.B., Owe, O., Axelsen, E.W.: A runtime environment for concurrent objects with asynchronous method calls. In: Martí-Oliet, N. (ed.) Proc. 5th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2004)
26. Lowry, M., Pressburger, T., Roşu, G.: Certifying domain-specific policies. In: Proceedings, International Conference on Automated Software Engineering (ASE 2001), pp. 81–90. IEEE, Los Alamitos, San Diego, California (2001)
27. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
28. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)

29. Meseguer, J., Roşu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
30. Meseguer, J., Roşu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
31. Meseguer, J., Roşu, G.: The rewriting logic semantics project. In: Proc. of SOS 2005, ENTCS, Elsevier, Amsterdam (to appear, 2005)
32. Mosses, P.D.: Foundations of modular SOS. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 70–80. Springer, Heidelberg (1999)
33. Ölveczky, P.C., Meseguer, J.: Real-Time Maude 2.1. In: Martí-Oliet, N. (ed.) Proc. 5th Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2004)
34. Palomino, M.: A predicate abstraction tool for Maude. Documentation and tool, `http://maude.sip.ucm.es/~miguelpt/bibliography.html`
35. Park, D.Y.W., Stern, U., Skakkebæk, J.U., Dill, D.L.: Java model checking. In: ASE 2001, pp. 253–256 (2000)
36. Roşu, G.: Programming language classes. Department of Computer Science, University of Illinois at Urbana-Champaign, `http://fsl.cs.uiuc.edu/~grosu/classes/`
37. Roşu, G., Chen, F.: Certifying measurement unit safety policy. In: Automated Software Engineering, 2003. Proc. 18^{th} IEEE Intl. Conference, pp. 304–309 (2003)
38. Stehr, M.-O., Cervesato, I., Reich, S.: An execution environment for the MSR cryptoprotocol specification language, `http://formal.cs.uiuc.edu/stehr/msr.html`
39. Stehr, M.-O., Talcott, C.: PLAN in Maude: Specifying an active network programming language. In: Gadducci, F., Montanari, U. (eds.) Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2002)
40. Stehr, M.-O., Talcott, C.L.: Practical techniques for language design and prototyping. In: Fiadeiro, J.L., Montanari, U., Wirsing, M. (eds.) Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing. 2005, Schloss Dagstuhl, Wadern, Germany (February 20–25, 2005)
41. Thati, P., Sen, K., Martí-Oliet, N.: An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In: Gadducci, F., Montanari, U. (eds.) Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2002)
42. Verdejo, A.: Maude como marco semántico ejecutable. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain (2003)
43. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude. In: In Proc. FORTE/PSTV 2000, vol. 183, pp. 351–366 (2000)
44. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In: Gadducci, F., Montanari, U. (eds.) Proc. 4th. Intl. Workshop on Rewriting Logic and its Applications, ENTCS, Elsevier, Amsterdam (2002)
45. Visser, W., Havelund, K., Brat, G., Park, S.: Java PathFinder - second generation of a Java model checker. In: Proceedings of Post-CAV Workshop on Advances in Verification (2000)

A Discussion on Grigore Rosu's Presentation

Peter Schmitt

I do not want to belittle your work, but the information you gave on formalizing semantics of programming languages in Maude, should be taken with a little grain of salt. We have looked at your Java semantics, the one that your grad students did in three weeks; it is very preliminary. There are lots of Java features that are not covered. There are no exceptions, there is no abrupt program termination. So, it covers some parts, but not too much.

Grigore Rosu

I said fragments of languages, and, actually, we have exceptions defined in other languages. We teach exceptions in that way, which I didn't put in the definition of Java. And this is not about Maude. This is about the methodology to define languages in rewriting logic; you can use another language instead of Maude if you want to. But I think that the methodology is viable, with Maude or without it; that is our point.

Patrick Cousot

I had a comment similar to this one but in a different form. You should wish to have end-users for your tool at the end, because it must correspond to something real, not an approximation of the language and things like that. That was my negative comment. My positive one is: I see a great analogy with the TVLA approach. Have you tried some universal abstractions like in TVLA?

Grigore Rosu

No. So, what was your negative comment?

Patrick Cousot

If you have a definition that is 95% complete, and the 5% that are missing that all the [difficult] problems, then it's vain. And so, you do not have end-users, because it is not real, and that is...

Grigore Rosu (*interrupts)*

Right. So, the aim would be to have complete definitions. We are working with these libraries of features, and we like to have full definitions of the language by putting all these features together. So, not 95% but full definitions, that is our purpose, and that is the language. The definition of the language, that's what the language is.

Kathi Fisler

To what extent are you trying to get interoperability between all those languages that you are defining? Is that at all an advantage you are going to get out of this framework?

Grigore Rosu

We would like to. We have not thought of that. We have not tried anything along those lines, but I think, we should be able to.

Kathi Fisler

But that is not a thing that you could get for free just because you would like it. Wouldn't you have to design that in at the beginning to get this kind of interoperation?

Grigore Rosu

I have to understand what you mean by interoperation. I mean, just to call functions from another language?

Kathi Fisler

To have a program that got fragments written in multiple languages.

Grigore Rosu

I think, that should be quite easy to do. I cannot say "100%, absolutely sure", but my feeling is that this should not be a big problem. We should keep the things disjoint somehow, yes.

Thomas Reps

Because Patrick [Cousot] mentioned TVLA, that prompted me to comment- because actually I wanted to make a negative comment that says something bad about TVLA. I think you're making the same mistake that was made in TVLA, which was to try and be all things to all people. It's a big mistake you've got to accept our transition systems, our transition-system definitions, our logic, etc., as well as our abstractions.

The alternative model is to have well-defined fragments that can be picked up by people who can drop them into Java programs or C++ programs. So, for example, the Parma Polyhedra library is an excellent example of that. There is a well-defined interface based on the primitives that one needs in abstract interpretation, plus mechanisms for defining transformers. I think that is a much better mechanism, and I think that we would do much better in TVLA by packaging up the basic programming abstraction of logical structures, together with a mechanism for defining abstract transformers, and then you do what you want with it in your programs.

There is also another layer at which to consider the issue; let me put in a plug for Stefan Schwoon's Weighted Pushdown System library and the Wisconsin system WPDS++. These are things that handle the question of reachability in weighted pushdown systems: you drop ... different weight domains into them, and then you get static analyzers from them. So, both of these systems are examples of a well-defined fragment with well-defined interfaces to little pieces that can come from others. If you try and be all things to all people, you limit your adoptability.

Grigore Rosu

First of all, I do not think we are making a mistake with what we started. Keep in mind that we keep the syntax separate; we only focus on the semantic definition of the particular features of languages. And, if I can get a chance to show you some definitions, you can see that basically, there is one rule or two at most per language feature. So, if your particular language does not have that, the methodology can do it, if you want to. And here, we are focusing on defining languages, what a language is. It is this logical specification. That is a definition of the language that I am going to use. Otherwise, what is a language? What can I start with? Well, how can I say what it means to verify a program, if I do not have a definition? So, we accept specification languages for requirements, but what is a language to start with? So here, we have a description of the language, it is executable, efficiently executable, and easy to read. So, what else do you want?