Open Problems and Challenges

From FSL

(back to Grigore Rosu's webpage)

Here is a list of open problems and challenges that I (Grigore Rosu) am interested in solving, in no particular order. While we are doing our best to keep this list actual, it may well be the case that some of the problems have been solved in the meanwhile or that we have found a different way to approach them. In case you are interested in working on any of these problems, please send me a note at (grosu@illinois.edu (mailto:grosu@illinois.edu)) to make sure that the problem is still actual and nobody is already working on it. The reason I decided to create and maintain this list of challenges is not only that it helps me keep track of them, but more importantly, that is also helps my students understand these topics better and put things at their place in the big picture. There are more problems here than one person can finish in a life-time. If you choose to work on a problem and believe that I can help, please let me know and we may work together on it. If you are a student in my FSL group, then you are actually expected to work together with me, and possibly other students, on one or more of these problems.

Previous versions

The list of problems and challenges below is revised continuously. At the beginning of each year, starting with 2016, we take a snapshot of it and also publish it as a technical report (to serve as a reference for citations). Here are all the snapshots so far:

Open Problems and Challenges 2016

Contents

Here are the short descriptions of all the problems, one per line. Each problem is preceded by a list of major topics that it covers, where: PL means *programming languages* and it includes topics like K, matching logic, reachability logic, language design, language semantics, etc.; RV means *runtime verification* and includes topics like monitoring, monitor synthesis, predictive runtime analysis, etc.; and C mean *coinduction* and includes both algorithmic aspects and interesting applications of coinduction.

[PL] Dynamic matching logic

[PL] Semantics of K

[PL] Sound and relatively complete reachability logic proof system with conditional rules (the challenge is the all-path)

[PL,C] Unifying deductive program verification and (symbolic) model checking

[PL,C] Coinductive program verification

[PL] Separation logic as a fragment of matching logic

[PL,C] Formal relationship between the Circularity proof rule and coinduction

[PL] True concurrency with K

[PL] Rewrite-based parsing

[PL] Narrowing-based parsing

[PL] Fast execution engine

[PL] Semantics-based compilation

[PL] Support full dynamic matching logic in K

[PL] Semantics-based test-case generation

[PL] Symbolic execution framework

[PL] Symbolic model checking

[PL] Invariant/Pattern inference using anti-unification

[PL] Aggressive state/configuration-reduction techniques

[PL] Language-independent infrastructure for program equivalence

[PL] Program portability checking

[PL] Translation validation, preferably for LLVM

[PL] Strategy language for K and reachability logic

[PL] Systematic comparison of K with other operational approaches

[PL] Configuration abstraction

[PL] Defining/Implementing language translators/compilers in K

[PL] Translations from K to other languages or formalisms

[PL] Semantics-based Compiler Generation

[PL,C] Certification of proofs done using the K framework

[PL] K semantics to new real languages

[PL] Verification of real-time languages, WCET verification

[PL] Module system for K

[PL] Semantics-based debugging: from debugging to verification

[PL] Type systems and abstract interpretations in K and reachability logic

[PL] Binders with matching logic

[PL,C] Migrate Circ (circular coinduction) examples/theory to K/Circularity rule

[PL] Define/Implement K in K

[PL,RV] Maximal causality as a configuration space reduction for semantics-based verification

[RV] Playing and replaying program executions

[PL,RV] Runtime verification of matching logic patterns

[PL,RV] Certifiable runtime verification

[PL,RV] Formalizing the JDK API

[RV] Parametric property mining using positive and negative examples

[RV] Offline analysis of large logs

[RV,C] Extended Regular Expression (ERE) Membership Checking

[RV,C] Optimal Monitor generation using coinduction

[RV] Runtime verification of timed properties

[RV] Evolution-aware program analysis

All Problems and Challenges

1. **Dynamic matching logic.**

   Currently (Jan 2016), we are framing matching logic as a *static logic*, that is, as one for reasoning about program configurations at a particular place in the execution of a program:

   Matching Logic --- Extended Abstract
   Grigore Rosu
   RTA'15, Leibniz International Proceedings in Informatics (LIPIcs) 36, pp 5-21. 2015
   PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-29-RTA/2015-06-29-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.4230/LIPIcs.RTA.2015.5) , RTA'15 (http://rdp15.mimuw.edu.pl/index.php?site=rta) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-ref.bib)

   For dynamic properties we use reachability logic:

   All-Path Reachability Logic
   Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu

RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

One-Path Reachability Logic
Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

However, both one-path and all-path reachability rules are particular formulae in a more general *dynamic matching logic*, which extends matching logic with two constructs: a modal *next* construct and a usual fixed-point *mu* construct. The resulting logic is expected to be sound and relatively complete; the relativity comes from the fact that we will want to fix a model of configurations, same like in reachability logic, to enable the use of SMT solvers for domain reasoning. In interested in this topic, you should also check out a draft paper I wrote a couple of years back but never published on how a Godel-Loeb proof rule gives us a complete proof system for finite execution traces; let me know and I can give you the draft. Once defined, then we should be able to prove the reachability logic proof system rules as theorems, thus modulo

From Hoare Logic to Matching Logic Reachability
Grigore Rosu and Andrei Stefanescu
FM'12, LNCS 7436, pp 387-402. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-fm.pdf) , Slides(pptx) (http://fsl.cs.uiuc.edu/pubs/Presentation_FM.pptx) , Slides(pdf) (http://fsl.cs.uiuc.edu/pubs/Presentation_FM.pdf) , Matching Logic (http://matching-logic.org) , FM'12 (http://fm2012.cnam.fr/) , , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-fm.bib.txt)

obtaining that dynamic matching logic generalizes Hoare logic mechanically. We should also be able to prove that dynamic matching logic similarly generalizes dynamic logic (https://en.wikipedia.org/wiki/Dynamic_logic_(modal_logic)) . The point of these generalizations is that Hoare or dynamic logic are basically "design patterns" to be manually crafted for each language separately, while dynamic matching logic is one fixed logic for all languages; each language is a particular set of axioms, which can be used in combination with a language-independent fixed proof system to derive *any* dynamic property for the language. In our view, *that* is how program reasoning/verification should be done, using *one* language-independent and powerful logic, and not to craft a specific logic for each specific language (which is what Hoare/dynamic/separation logic advocate).

2. **Semantics of K.**

K was mostly explained informally until now, calling it a "rewrite-based framework" and relying on reader's intuition and common sense. Semantically, so far we framed K either as a notation within rewrite logic by showing how to translate it to rewrite logic,

An Overview of the K Semantic Framework
Grigore Rosu and Traian Florin Serbanuta
J.LAP, Volume 79(6), pp 397-434. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap.pdf) , Slides(PPTX) (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap-slides-2011-01-14-Iasi.pptx.zip) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap-slides-2011-01-14-Iasi.pdf) , K Tool (http://k-framework.googlecode.com/) , J.LAP (http://dx.doi.org/10.1016/j.jlap.2010.03.012) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap.bib.txt)

(see also the papers under systematic comparison of K with other operational approaches) or in terms of translation to graph rewriting (to show its true concurrency capabilities)

A Truly Concurrent Semantics for the K Framework Based on Graph Transformations
Traian Florin Serbanuta and Grigore Rosu
ICGT'12, LNCS 7562, pp 294-310. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2012-icgt.pdf) , ICGT'12 (http://www.informatik.uni-bremen.de/icgt2012/) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2012-icgt-slides.pdf) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2012-icgt.bib.txt)

More recently, we introduced reachability logic

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

One-Path Reachability Logic
Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

and that allowed us to give a semantics of K that is more faithful to our uses of K.

Yet, the semantics of K is more finer grained than all the above. Consider, for example, the following assignment rule in IMP:

```
rule <k> X = I:Int => I ...</k> <state>... X |-> (_ => I) ...</state>
```

So in one atomic transaction, we rewrite both the assignment construct in the k cell and the value that X is bound to in the state to I. If we regard the above as a rewrite or a reachability rule, then we would need to translate into something like this:

```
rule <k> X = I:Int ~> K </k> <state> S1, X |-> _, S2 </state>
  => <k>         I ~> K </k> <state> S1, X |-> I, S2 </state>
```

The resulting rule above is not only more verbose and uglier than the original K rule, but it is also less concurrent. Indeed, once we add threads to IMP (see, for example, the IMP++ language in the K Tutorial (http://www.kframework.org/index.php/K_Tutorial)), we want K rules like above to apply concurrently, because we want two threads assigning to different variables to proceed concurrently and not interleaved.

Fortunately, dynamic matching logic provides the granularity we need to give semantics to the intended K rewriting. For example, the rule above would become

```
rule <k> (X = I:Int -> o I) ...</k> <state>... X |-> (_ -> o I) ...</state>
```

where "o" is the "next" modal construct of dynamic matching logic, "->" is logical implication and "_" and "..." are universally quantified (anonymous) variables like in K.

Therefore, as soon as the dynamic matching logic challenge is solved, or at the same time, we need to give K a crystal clear dynamic matching logic semantics. And from there on we will call K a *best-effort implementation of dynamic matching logic*, the same way Maude (http://maude.cs.illinois.edu) is a best-effort implementation of rewrite logic.

3. **Sound and relatively complete reachability logic proof system with conditional rules (the challenge is the all-path).**

The original reachability logic proof systems that we proposed and proved sound and relatively complete, namely

Towards a Unified Theory of Operational and Axiomatic Semantics
Grigore Rosu and Andrei Stefanescu
ICALP'12, LNCS 7392, pp 351-363. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-icalp.pdf) , Slides(pptx) (http://fsl.cs.uiuc.edu/pubs/2012-07-12-rosu-stefanescu-ICALP.pptx) , Slides(pdf) (http://fsl.cs.uiuc.edu/pubs/2012-07-12-rosu-stefanescu-ICALP.pdf) , Matching Logic (http://matching-logic.org) , ICALP'12 (http://www2.warwick.ac.uk/fac/cross_fac/dimap/icalp2012/) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-icalp.bib.txt)

From Hoare Logic to Matching Logic Reachability
Grigore Rosu and Andrei Stefanescu
FM'12, LNCS 7436, pp 387-402. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-fm.pdf) , Slides(pptx) (http://fsl.cs.uiuc.edu/pubs/Presentation_FM.pptx) , Slides(pdf) (http://fsl.cs.uiuc.edu/pubs/Presentation_FM.pdf) , Matching Logic (http://matching-logic.org) , FM'12 (http://fm2012.cnam.fr/) , , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-fm.bib.txt)

Checking Reachability using Matching Logic
Grigore Rosu and Andrei Stefanescu
OOPSLA'12, ACM, pp 555-574. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-oopsla.pdf) , Slides(pptx) (http://fsl.cs.uiuc.edu/pubs/2012-10-24-rosu-stefanescu-OOPSLA.pptx) , Slides(pdf) (http://fsl.cs.uiuc.edu/pubs/2012-10-24-rosu-stefanescu-OOPSLA.pdf) , Matching Logic (http://matching-logic.org) , OOPSLA'12 (http://splashcon.org/2012/cfp/378) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-2012-oopsla.bib.txt)

worked only with language semantics defined using *unconditional* reachability rules and deriving *one-path* reachability rules. These are sufficient for deterministic language semantics in several frameworks, including in K. We tried hard to eliminate the "unconditional" and the "one-path" limitations, but we only partially succeeded.

First, we were able to extend our soundness and relative completeness results to allow semantics defined using *conditional* rules, but still deriving only one-path reachability rules:

Reachability Logic
Grigore Rosu, Andrei Stefanescu, Stefan Ciobaca and Brandon Moore
Technical Report http://hdl.handle.net/2142/32952, July 2012
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-stefanescu-ciobaca-moore-2012-tr.pdf) , TR@UIUC (http://hdl.handle.net/2142/32952)

One-Path Reachability Logic
Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

Now thanks to this paper,

A Rewriting Logic Approach to Operational Semantics
Traian Florin Serbanuta, Grigore Rosu and Jose Meseguer
Information and Computation, Volume 207(2), pp 305-340. 2009
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic.pdf) , Experiments (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic-experiments.zip) , J.Inf.&Comp. (http://dx.doi.org/10.1016/j.ic.2008.03.026) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic.bib.txt)

which shows that virtually all operational semantics approaches can be represented using rewriting with conditional rules, at least we have a general language-independent (sound and relatively complete) verification infrastructure that works with any *deterministic* language defined using any operational semantic formalism. We say "deterministic language" above, because for non-deterministic languages we typically want to prove *all-path* reachability.

Second, we were able to extend our results to prove *all-path* reachability, but only when the language semantics is defined using unconditional rules:

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

This is good enough for us in K, because the language semantics that we define in K consist of unconditional reachability rules, so we stopped here with our quest.

However, scientifically speaking, it is very frustrating that we were not able to find a perfect solution to such a beautiful and major problem! It should be possible to obtain a sound and relatively complete proof system for a combined logic with both one-path and all-path conditional reachability rule statements, where the target programming language semantics is nothing but a subset of such rules. Such a proof system would completely eliminate the need for Hoare logic or axiomatic semantics or any other semantics used for program verification, and all the heavy work on proving such semantics sound and relatively complete with respect to a reference model operational semantics of the language, simply because the proof system itself gives you that, for any language, be it concurrent or not, defined using any operational semantic style. This would be a fundamental result in the field, helping future generations of designers and developers of programming languages and program verification tools to do all these better: define a formal operational semantics of your language in any formalism you like, and that's all, because the desired program verification logic for your language comes for free and it is not only sound, but also relatively complete. For all practical reasons we already have that in K, thanks to the last paper above, but the challenge for the perfect result remains.

4. **Unifying deductive program verification and (symbolic) model checking.**

In our approach, there is no distinction between deductive verification and model checking. These are just particular uses of our fixed and language-independent proof system, to derive particular properties about particular languages. Moreover, optimizations made for one can apply to the other. For example, a faster matching logic prover will give us faster deductive reasoning and also faster model checking. Also, a faster checker for already visited configurations/states (better hashing), will obviously give us a faster model checker, but it will also give us faster verifier because the circularity rule will be applied more effectively. I

can personally think of no difference between a "model checker" and a "deductive program verifier" in our approach. But this needs to be spelled out rigorously and empirically. We should implement automated proof search optimizations that give users the feel of a "model checker", at the same time having the system generate a proof object using our proof system, as a checkable proof certificate of what the model checker did. Much of the research in the model checking field goes into developing alogorithms, such as automata-based ones, where the automata encode both the program and the property to check. This approach is particularly useful for explicit-state model checking, but not only. Then there is also much work on using BDDs for symbolic model checking. We should also consider such algorithms, but then we should still be able to generate a proof object as a result of the analysis. My conjecture is the following. The key ingredient that makes our approach particularly suitable for such a grande unification is the *Circularity* rule. See for example this paper, but any of the papers mentioned under the dynamic matching logic challenge works:

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

Check the Circularity proof rule out. It is saying that if you want to prove a reachability property, then assume it and try to prove it, but use it in your proof only after you make one trusted step with the operational semantic rules. This is very similar to how model checkers saturate their state-space search. We should be able to take model checking algorithms and adapt them to work with the general-purpose transition systems that semantics associate to programs, and then take their verification results and translate them into proofs using our proof system, where cycles in the internal graphs or automata maintained by the model checker result in applications of the Circularity rule.

5. **Coinductive program verification.**

Recall that conventional Hoare-style program verification typically proves *partial correctness* of programs, that is, that a program satisfies its property if it terminates. The non-terminating programs satisfy any property under partial correctness; in particular, a non-terminating program satisfies the property *false*. Partial correctness should not be confused with *total correctness*, which means that the program terminates and it satisfies the desired property. Termination is typically handled using different mechanisms (e.g., *variants*). Sometimes researchers craft Hoare logics for total correctness, but those tend to be rather intricate. Now, when partial correctness is sought to be proved, a very nice way to do it is by *coinduction*:

Program Verification by Coinduction
Brandon Moore and Grigore Rosu
Technical Report http://hdl.handle.net/2142/73177, February 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/moore-rosu-2015-tr/moore-rosu-2015-tr-

public.pdf) , Matching Logic (http://matching-logic.org/) , DOI (http://hdl.handle.net/2142/73177) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/moore-rosu-2015-tr/moore-rosu-2015-tr-ref.bib)

The main idea is that the partial correctness properties of a programming language are precisely the coinductive closure of the step relation given by the operational semantics of the programming language. This is not only an interesting theoretical observation, but we believe it can be quite practical. It shares the same belief of an ideal language framework, where the operational semantics is taken as input and tools are generated from it. The tool in this case is more like a workbench, where one can use an interactive theorem prover like Coq or Isabelle with support for coinduction, and do proofs about programs based only on the existing generic mathematical infrastructure and the operational semantics of the programming language defined as a binary (step) relation.

The main challenge here is to make coinductive program verification practical. That means developing proof strategies that *automate* the process. Most likely those will be quite similar to those in the matching logic prover of K; for example "execute symbolic configuration using operational semantics rules, doing case analysis when multiple rules match, and giving priority to claimed circularities (which in this case would be coinductive hypotheses)".

Achieving the above in a satisfactory manner would be quite exciting news for the mechanical verification community. Unfortunately, the state-of-the-art in mechanical verification (Jan 2016) is to formalize two different semantics of the target programming language, and to prove a special relationship between them. One semantics is executable (operational or denotational) and serves as a reference model for the language, because it allows you to run programs and see what they do. The other semantics is axiomatic, essentially a Hoare logic or a verification condition generator based on a hypothetical Hoare logic, and serves for program verification. The axiomatic semantics tends to be quite involved when real programming languages are concerned, so in order to increase confidence in the results of verification, a proof of soundness for the axiomatic semantics wrt the executable semantics is also provided. In our view and experience, defining even *one* formal semantics to a real language (say C or Java) is a huge effort already. Defining two semantics and proving the soundness theorem is just too uneconomical if it can be avoided. And it can! The secret is *coinduction*. Indeed, coinduction with the operational semantics of the language gives you a sound and relatively complete verification infrastructure for any language. And with the right degree of automation, it can be even more practical than the current axiomatic semantics based approached. Why? Because you are actually *executing your program all the time*, so whenever the proof gets stuck or is wrong, which is what happens in most of the situations (you really only prove the program once, all the other attempts until you get there are typically wrong steps or stuck proofs), you know exactly where it happened and why.

You may also want to check the problems under symbolic execution framework and formal relationship between the Circularity proof rule and coinduction for a through understanding of the problem and the state-of-the-art.

There is a disadvantage of the coinductive approach, though. There will be fewer PhD theses on formal semantics of programming languages. With the current state-of-the-art, defining an executable semantics of C is one PhD thesis, defining an axiomatic semantics of C is yet another PhD thesis, and then proving the soundness of the latter semantics in terms of the former semantics is yet another PhD thesis. This last one will have the merit that it will also fix bugs in the axiomatic semantics, because it will be full of bugs as it is not executable and thus not testable. And then all the above will be maintained as C evolves (C99 -> C11 -> ?). Excuse our sarcasm, but the current state-of-the-art is simply unacceptable. Period.

6. **Separation logic as a fragment of matching logic.**

That matching modulo AC naturally yields spatial separation was hinted even in the first paper on matching logic,

Matching Logic --- Extended Report
Grigore Rosu and Wolfram Schulte
Technical Report UIUCDCS-R-2009-3026, January 2009
TR@UIUC (http://hdl.handle.net/2142/10790) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-schulte-2009-tr.bib.txt)

The paper above also suggests that matching gives separation anywhere, not only in the heap, and shows that common heap patterns in separation logic, such as lists, can be similarly defined using terms and matching. Three years later we showed that separation logic falls as a fragment of matching logic:

Checking Reachability using Matching Logic
Grigore Rosu and Andrei Stefanescu
OOPSLA'12, ACM, pp 555-574. 2012
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2012/rosu-stefanescu-2012-oopsla/rosu-stefanescu-2012-oopsla-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2012/2012-10-24-rosu-stefanescu-OOPSLA.pptx) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2012/2012-10-24-rosu-stefanescu-OOPSLA.pdf) , Matching Logic (http://matching-logic.org) , DOI (http://dl.acm.org/citation.cfm?doid=2384616.2384656) , OOPSLA'12 (http://splashcon.org/2012/cfp/378) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2012/rosu-stefanescu-2012-oopsla/rosu-stefanescu-2012-oopsla-ref.bib)

Back then, matching logic was what we now call *topmost* matching logic, that is, only patterns of top sort, *Configuration*, were allowed. That was sufficient to define a translation from separation logic formulae to matching logic patterns and prove a theorem stating that a separation logic formula is valid if and only if its translation is valid in matching logic. However, it had the problem that all pattern abstractions had to be defined at the top Configuration level, which was verbose. Also, the translation of separation logic formulae modified the original formula, so one might claim that the resulting formula was not as easy to read and reason with as the original one.

We then generalized matching logic to support patterns of any sort, and once we did that, we managed to embed separations logic as a particular matching logic theory within a fixed model, that of maps:

Matching Logic: A Logic for Structural Reasoning
Grigore Rosu
Technical Report http://hdl.handle.net/2142/47004, Jan 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/rosu-2014-tr/rosu-2014-tr-public.pdf) , Matching Logic (http://matching-logic.org) , DOI (http://hdl.handle.net/2142/47004) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/rosu-2014-tr/rosu-2014-tr-ref.bib)

Matching Logic --- Extended Abstract
Grigore Rosu
RTA'15, Leibniz International Proceedings in Informatics (LIPIcs) 36, pp 5-21. 2015

PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-29-RTA/2015-06-29-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.4230/LIPIcs.RTA.2015.5) , RTA'15 (http://rdp15.mimuw.edu.pl/index.php?site=rta) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-ref.bib)

The nice part of the embedding discussed in the papers above is that the original separation logic formula does not change at all! Indeed, we defined a matching logic theory whose syntax of symbols is identical to that used by separation logic, and thus every separation logic formula is a matching logic pattern over that signature as is, without any change. Then it is shown that the separation logic formula is valid in separation logic if an only if it is matched by the particular model of maps in matching logic. The conclusion here is that, through the lenses of matching logic, separation logic is a particular fixed-model fragment for exclusively specifying heap patterns. Matching logic, on the other hand, allows you to specify patterns everywhere in a program configuration, not only in the heap. And you do not have to fix the model either.

So you may ask why we are interested in separation logic at all then. Separation logic proposed a different way to think about program state properties, where the actual structure of the heap matters for what it is and not for how it can be encoded using FOL or other classic logics. Matching logic embraces this view; in fact, it is all about this view, but extended to the entire program configuration, not only the heap. Several interesting theoretical results and automation heuristics have been proposed in the context of separation logic. My conjecture is that many of these results transcend the particularity of separation logic and are just as well applicable in the more general context of matching logic. Doing things more generally may help us better understand the nature of those results, and may turn them into more powerful ones. On the practical side, separation logics have been used to specify program properties in several program verifiers, and several programs have been verified that way. We should be able to reuse all these properties unchanged in matching logic provers. All these indicate that a systematic and thorough study of the various variants of separation logic in the context of matching logic might be more than an exercise. It might be instrumental to obtaining both interesting new theoretical results and practical matching logic program verifiers. I propose the following steps, as a start: (1) Clarify that separation logic is equivalent to first-order logic in the map model directly, without going through matching logic; (2) Show (1) for several variants of separation logic, to make the point crystal clear; (3) Make the point that it is not practical to use FOL, though, because the translations at (2) increase the size of the formula and add quantifiers; (4) Like in the two papers above, show how each variant at (2) can be framed as a matching logic theory over a fixed model of maps, without any formula translation; (5) Implement a matching logic prover and then show how to obtain separation logic provers from it, for each variant at (2).

7. **Formal Relationship between the Circularity proof rule and Coinduction.**

As we claimed in our papers on reachability logic,

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic

(http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

One-Path Reachability Logic
Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

the Circularity proof rule has a coinductive nature. We need to nail down the relationship between the two. Some intuitive connections are discussed in

Program Verification by Coinduction
Brandon Moore and Grigore Rosu
Technical Report http://hdl.handle.net/2142/73177, February 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/moore-rosu-2015-tr/moore-rosu-2015-tr-public.pdf) , Matching Logic (http://matching-logic.org/) , DOI (http://hdl.handle.net/2142/73177) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/moore-rosu-2015-tr/moore-rosu-2015-tr-ref.bib)

but the problem is far from being solved. One we understand the connection well, the next step is to formalize it as a mechanical translator from reachability logic proofs to coinductive proofs. Eventually, this will become part of the K framework, where proofs done using the K prover can be translated into Coq or Isabelle or other similar theorem prover proof scripts, so that the latter can reconstruct the proof in a certifiable manner.

For a thorough understanding of this problem, you may also want to check the symbolic execution framework, coinductive program verification, and language-independent infrastructure for program equivalence problems.

8. **True concurrency with K.**

This is related to the semantics of K challenge. As explained there, K offers finer grained concurrency than conventional rewriting, for example the chemical abstract machine (http://dx.doi.org/10.1145/96709.96717) or rewrite logic (http://dx.doi.org/10.1016/0304-3975(92)90182-F) , because it allows rewrite rules that overlap in the conventional sense apply concurrently. We need to do a systematic study of computational frameworks and/or logics for *true concurrency* and then show how each can be represented in K in a way that gives them exactly the intended truly concurrent semantics. While this challenge naturally combines with the semantics of K challenge, we prefer to study them as separate problems, or at least to publish them as such. Each of them is big and important enough in itself.

We want, in particular, the K concurrent semantics to make it possible to sort an arbitrary permutation of elements 1,2,...,n in *two* concurrent steps with just the transposition rule

```
rule (X => Y)...(Y => X)
```

I have proposed this problem as a "concurrency challenge" to students and colleagues at various occasions, without knowing or even expecting that it can be done in a constant number of concurrent steps! Andrei Stefanescu (http://fsl.cs.illinois.edu/index.php/Andrei_Stefanescu) found a first solution in 2010, with three concurrent steps. Below is a solution in two steps, found by Jérôme Dohrau (http://www.pm.inf.ethz.ch/people/person-detail.html?persid=173070) during the Marktoberdorf Summer School 2016 (https://sites.google.com/site/marktoberdorf16/) . Any permutation can be obtained as a composition of cycles, and each cycle can be sorted independently and in parallel with the other cycles. Moreover, cycles of size 1 are already sorted and cycles of size 2 can be sorted in one step. Without loss of generality, suppose that our permutation is 2,3,4,...,n-1,n,1 for n larger than 2. Apply the transpositions (1,n-1), (2,n-2), ..., ([(n-1)/2],[(n+1)/2]) in one concurrent step. The permutation becomes n,n-1,...,4,3,2,1. Now apply the transpositions (1,n), (2,n-1), ..., ([n/2],[n/2]+1) in another concurrent step, and the permutation becomes 1,2,3,4,...,n-1,n.

Although the transpositions in the problem above happened to be applied in a nested fashion, we want a K semantics that gives us the "maximum possible concurrency" in this example, specifically to allow us to apply the rule above concurrently on any disjoint pairs of elements. For example, we want to be able in one concurrent step to rewrite 1,2,3,4,5 to 4,2,5,1,3 using transpositions (1,4) and (3,5). Unfortunately, the obvious desugaring of K rules into normal rewrite rules to "borrow" the rewrite logic semantics in order to give K a semantics does not work. Indeed, the rule above would desugar to a rule of the form

```
rule X,L,Y => Y,L,X
```

where L ranges over a list of elements, and the rewrite logic semantics does not allow us to rewrite 1,2,3,4,5 to 4,2,5,1,3 in one concurrent step.

There are two parallel semantic avenues to study the problem of true concurrency of K. One is to follow the dynamic matching logic approach to the semantics of K, where K rules are syntactic sugar for particular dynamic matching logic formulae. Another is to follow the reachability logic approach, with a proof system specialized for the particular kind of dynamic matching logic formulae that we call reachability rules. However, in this latter case, we have to extend the notion of reachability rule to allow local and multiple reachability rules into a given context, like the K rules allow; then we should be able to take two such reachability rules which can be applied concurrently and combine them into one reachability rule that collects all the reads/writes of the two combined rules.

Here is a paper where K was used to give semantics to a truly concurrent language, Orc:

Musab A. AlTurki, Omar Alzuhaibi: Towards Formal Verification of Orchestration Computations Using the 핎 Framework. FM 2015: 40-56 http://dx.doi.org/10.1007/978-3-319-19249-9_4

Whatever truly concurrent semantics for K we come with, it would be nice to yield Orc its desired truly concurrent semantics.

9. **Rewrite-based parsing.**

Powerful language frameworks require powerful parsing. Indeed, the first tool generated from a formal language definition is a parser for the language. In fact, several parsers. In addition to the parser for the language, we also want parsers for the semantic rules of the language, in which we want to use concrete instead of abstract syntax, as well as for symbolic program configurations needed for specifying properties to be verified, in which we also want to use concrete syntax. Therefore, the user-defined syntax of the programming language will need to be extended with the syntax of K. Moreover, the syntax of K itself can be changed by the user, for example in order to avoid conflicts or ambiguities with the programming language they define. Even if the user adjusts the syntax of K, syntactically complex languages like C or JavaScript will pose significant parsing challenges. A powerful parsing framework is needed, which should allow users to see, debug and resolve parsing ambiguities.

In K implementations, the parser should be made available to the user using a special KLabel, say `#parse(String,Module[,Sort])`. Also, the implementation of K itself should use that exact same parser to parse its modules, by constructing the right module to be used as input to the parser; that module will combine the user-defined language syntax with the (potentially user-modified) syntax of K. Ideally, we would like the parsing infrastructure to be uniformly integrated with the K framework, at least at its core notation (the front will likely provide parsing specific notations, such as the ">" for constructor priorities). Since K is based on its special rewrite rules, we would like to allow users to disambiguate their syntax using similar rules. Consider the following trivial syntax:

```
syntax Exp ::= Exp "+" Exp
             | Exp "*" Exp
```

The string "1+2*3" is clearly ambiguous. We would like the parser to construct a special K-style configuration making the ambiguity clear, for example something like this:

```
<amb>... <kast> _+_(...) </kast> <kast> _*_(...) </kast> ...<amb>
```

Therefore, an `amb` cell is inserted wherever ambiguities are detected, holding all the ambiguous parse trees as K ASTs (`kast`). Like in the K debugger, the `...` can be unfolded by need using special commands or mouse clicks. Like with any other syntax, the user can also modify the the syntax of parsing configurations; for example, they may replace `amb` with `ambiguity`, and `kast` with `parse-tree`, etc.

Based on many years of experience, we believe that rewrite rules are quite a powerful computational mechanism. We expect them to be equally powerful for parsing disambiguation. For example, to state that _*_ binds tighter than _*_, we can write the following rule:

```
rule <kast> _+_(_:KList) </kast> (<kast> _*_(_:Klist) </kast> => .)
```

That is, if two parser are possible, one with _+_ as root and the other with _*_ as root, then get rid of the latter. Or, alternatively, we can say that a parsing tree with a _+_ immediately under a _*_ is never allowed (one should use parentheses if one wants that, that is, "(1+2)*3"):

```
rule <kast> _*_(_:Klist,_+_(_:Klist),_:Klist) </kast> => .
```

Adding enough rules like the above, eventually all undesired parses are hopefully eliminated. When that happens, a rule like the above can then get rid of the unnecessary `amb`cell:

```
rule <amb> <kast> K </kast> </amb> => <kast> K </kast>
```

Allowing the user to add such rewrite rules to interact with the parser can give them a lot of semantic power. For example, one may use the parsing ambiguities as a means to define non-deterministic languages, where each ambiguity is one possible way to look at the program. Recall, for example, reduction semantics with evaluation contexts, where non-deterministic evaluation strategies are defined using ambiguous evaluation context grammars. Also, this would allow users to *parse in stages*. For example, consider C. A first grammar can parse all the top level declarations (globals and functions), putting the body of each declaration in a string bubble; for example,

```
#parse("int f(int x) {return x+1;}", C-DECLARATIONS-SYNTAX, Pgm)
```

can yield a K AST of the form

```
__(_){_} (int, #token("f","Id"), __(int,#token("x", "Id")), #token("return x+1;", FunctionBodyBubble))
```

then one can enable further parses of bubble with rules of the form

```
rule #token(S, FunctionBodyBubble)) => #parse(S, C-FUNCTION-BODY-SYNTAX)
```

This way, each stage uses a simpler grammar that hopefully generates fewer ambiguities. Moreover, complex parsers can be developed, where a parser state can be maintained and used at later stages; for example, remember that "`a*b;`" in C means a pointer declaration of type `a` when `a` has been declared as a type with `typedef`, and a multiplication otherwise. Also, better error messages can be reported. Not to mention that performance can increase, because the K rewrite engine takes advantage of parallel architectures, so multiple instances of the rule above can be applied in parallel.

It should be easy to implement an *inefficient* parsing infrastructure like above. For example, use an Earley-based algorithm to obtain all the parsings, put them into a configuration like above, and then use the K rewrite infrastructure to rewrite the configuration. The challenge here is how to do this *efficiently*. Essentially, we'd like to apply the disambiguation rules like above at parse time, while the parsing DAG/tree is being generated. Consider, for example, the disambiguation rule

```
rule <kast> _*_(_:Klist,_+_(_:Klist),_:Klist) </kast> => .
```

discussed above. An efficient parser would use it to never even attempt to parse a _+_ underneath a _*_, which can save considerable resources.

An ideal solution to this problem would be a parser generator that takes as input a grammar and a set of disambiguation rules, and generates an efficient parser as output that applies the disambiguation rules on the fly, as the input string is being processed. Moreover, when the grammar falls into fragments for which efficient parser generators are known (e.g., LL(k), LR(k), etc.), we would like the complexity of our generated parser to stay within the same asymptotic complexity.

10. **Narrowing-based parsing.**

Narrowing is rewriting with unification instead of matching. You start with a term `T` with variables and then find rewrite rules whose LHS unifies with `T`. Apply the mgu and rewrite rule to `T` and obtain a new term `T'`. Keep doing that until the term is not unifiable with the LHS of any rule. The overall result is that you iteratively *narrow* the original term by adding structure to its variables so that rules apply and keep applying rules this way until the term cannot be narrowed anymore.

Narrowing is a very powerful computational mechanism. It turns out that we can use narrowing for parsing. The code below show some simple experiments with Maude's narrowing support (v2.7) to flesh out the idea, and it seems to work:

```
in full-maude .

mod TOKENS is
  sorts Token TokenList .
  op nil : -> TokenList .
  op __ : Token TokenList -> TokenList .
  ops a b c d e f g h i j k l m n o p q r s t u v x y z w  0 1 2 3 4 5 6 7 8 9 : -> Token .
  op append : TokenList TokenList -> TokenList .
  var T : Token .  vars Ts Ts' : TokenList .
  rl append(nil,Ts') => Ts' .
  rl append(T Ts, Ts') => T append(Ts,Ts') .
endm

mod KAST is
  sorts KItem K KLabel KList .
  subsorts KItem < K < KList .
  op .KList : -> KList .
  op _,_ : KItem KList -> KList .
  op _`(_`) : KLabel KList -> KItem .
endm

mod GRAMMAR is
  including TOKENS .
  including KAST .

  op [_,_] : K KItem -> TokenList .

***(
  syntax S ::= S A | epsilon
  syntax A ::= a | b
***)

  sorts S A .
  subsorts S A < K .

  ops 'a:->A' 'b:->A' '__:S*A->S' 'epsilon:->S' : -> KLabel .
```

```
  vars S1 S2 : S .  vars A1 A2 : A .  vars K1 K2 : KItem .
  rl [S1, '__:S*A->S'(K1,K2)] => append([S2, K1], [A1, K2]) .
  rl [S1, 'epsilon:->S'(.KList)] => nil .

  rl [A1, 'a:->A'(.KList)] => a nil .
  rl [A1, 'b:->A'(.KList)] => b nil .

endm

select FULL-MAUDE .
loop init .

(search [1,5] in TOKENS :  append(Ts:TokenList, c d nil) ~>* a b c d nil .)

(search [1,7] in GRAMMAR :  [S1:S, AST:K] ~>* nil .)

(search [1,7] in GRAMMAR :  [S1:S, AST:K] ~>* a nil .)

(search [1,8] in GRAMMAR :  [S1:S, AST:K] ~>* a b nil .)
```

The above outputs:

```
search [1,5] in TOKENS : append(Ts:TokenList,c d nil) ~>* a b c d nil .

Solution 1
Ts:TokenList --> a b nil

No more solutions.

search [1,7] in GRAMMAR :[S1:S,AST:K] ~>* nil .

Solution 1
AST:K --> 'epsilon:->S'(.KList)

No more solutions.

search [1,7] in GRAMMAR :[S1:S,AST:K] ~>* a nil .

Solution 1
AST:K --> '__:S*A->S'(('epsilon:->S'(.KList)),'a:->A'(.KList))

No more solutions.

search [1,8] in GRAMMAR :[S1:S,AST:K] ~>* a b nil .

Solution 1
AST:K --> '__:S*A->S'(('__:S*A->S'(('epsilon:->S'(.KList)),'a:->A'(.KList))),'b:->A'(.KList))

No more solutions.

Bye.
```

Unfortunately, it looks like narrowing in Maude only works with Full Maude, and does not have support for associativity, so we had to cripple the definition to use `append` instead of associative lists. And in the end it is quite slow. It takes about 1 minute for the above to terminate on a regular desktop.

So here is the question. We want a powerful narrowing engine for test-case generation and many other reasons anyway in the K framework, which should interact well with strategies and associative lists. Then why not use that for parsing as well? Of course, *performance* is a serious issue, but can't we improve the performance to an extent that it will not be a bottleneck? We want narrowing to be fast anyway.

11. **Fast execution engine.**

    While K rewriting can be mimicked with conventional rewriting, the price to pay may be too high to do so. Our first implementation of K was in Maude

    K-Maude: A Rewriting Based Tool for Semantics of Programming Languages
    Traian Florin Serbanuta and Grigore Rosu
    WRLA'10, LNCS 6381, pp 104-122. 2010
    PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.pdf) , Slides (PDF) (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla-slides.pdf) , K-Maude (http://k-framework.googlecode.com/) , LNCS (http://dx.doi.org/10.1007/978-3-642-16310-4_8) , WRLA'10 (http://wrla10.ifi.uio.no/) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.bib.txt)

    and it worked fine for small languages. However, for larger languages, with tens of semantic components in their configuration, the cost of AC matching started to become unbearable. Note that Maude is indeed a super-fast general purpose rewrite engine, able to perform millions of rewrite steps per second in general. But with complex language configurations, with several layers of nested cells (which are AC soups), the performance of Maude was quickly reduced to only a few thousands steps per second, sometimes a few hundred. To avoid this cost, starting with v3.6 K provides its own rewrite engine, implemented fully in Java. In addition to specialized cell matching to avoid unrestricted AC, the new rewrite engine also provides specialized data-structures, such as hash maps and vectors and soon threads. All these have increased the execution speed of K between one and two orders of magnitude. Also, K has been reorganized to allow various backends. Additionally, Runtime Verification, Inc. (http://runtimeverification.com) has developed an OCAML backend, which is two additional orders of magnitude faster than the Java backend when executing concrete programs.

    While the OCAML backend is orders of magnitude faster than the original Maude backend, there is still room for faster. For example, when executing the C semantics with the OCAML backend, the resulting C interpreter is still about 100,000 slower than if we compiled the program with gcc and then executing the native binary. Two or even three orders of magnitude slower than a compiler is acceptable for an interpreter, but 5 orders is still too much. We believe that we can improve over the OCAML backend another two orders of magnitude. One possible approach could be to translate K to LLVM, and then use LLVM compilers to native code. The challenge here is to implement very efficient indexing algorithms and/or pattern match automata specialized for K, so that we do a minimal amount of computation at each step in order to decide which rule can match.

    Another major challenge is to develop a semantics-based compiler for K.

12. **Semantics-based compilation.**

    An ideal language framework should allow us to generate compilers from language definitions. Specifically, we hope that soon K will have the capability to take a language semantics and a program in that language, and generate an efficient binary for the program. Conceptually, K already provides the ingredients needed for such a semantics-based compiler. One way to approach the problem is to use symbolic execution on all non-recursive non-iterative program fragments and calculate the mathematical summary of the fragment as a

(potentially large) K rule; the left-hand side of that rule will contain the fragment. For example, consider the trivial IMP language and a while loop whose body contains no other while loops. Then we can conceptually replace the while loop body with a new statement, say `stmt17093`, whose semantics is given with a K rule that symbolically accumulates the semantics of the individual statements that composed `stmt17093`. We can do the same for all non-loop fragments of code, until we eventually obtain a program containing only while statements whose bodies are special statement constants like `stmt17093`, each with its own K semantics.

Once the above is achieved, there are two more important components left. One is to translate the while statements into jump statements in the beckend language, say LLVM. In the long term, this should be inferred automatically from the K semantics of the while statement. In the short term, we can get the user involved by asking them to provide a translation for the while loop (yes, this is not nice, but hey, getting a compiler for your language is a big deal). The other component is to translate the K rules for statements like `stmt17093` into efficient target language code. We believe this is an orthogonal issue, which we want to have efficiently implemented in K anyway, as part of our effort on providing a fast execution engine for K.

13. **Support full dynamic matching logic in K.**

    If we can give [[#K-semantics|semantics to K] using [#dml|dynamic matching logic], then why not support the entire dynamic matching logic notation in K? It may be convenient to keep using the existing notational sugar in the frontend, such as the rewrite arrow `=>` instead of "implies next", but it would be nice and uniform to have KORE (https://github.com/kframework/k/wiki/KAST-and-KORE) use directly the dynamic matching logic notation. And of course, since KORE is included in the full K notation, that means to also allow people to use the dynamic matching logic notation in their definitions if they choose to. Note that dynamic matching logic can be computationally expensive, so it is likely that we would not support it in its full generality. But we can support more and more of it. For example, we can start by allowing Boolean connectives, so we can write

```
rule I1 / I2 /\ (I2 =/= 0)@INT => (I1 / I2)@INT
```

instead of

```
rule I1 / I2 => (I1 / I2)@INT requires (I2 =/= 0)@INT
```

or

```
rule <k> nat X; => . ...</k>  <state>... . => (X |-> ?V /\ (?V >= 0)@INT) ...</state>
```

instead of

```
rule <k> nat X; => . ...</k>  <state>... . => (X |-> ?V) ...</state>  ensures (?V >= 0)@INT
```

Or, remember issue #1380 (https://github.com/kframework/k/pull/1380%7CK) , where we wanted to be able to write (comma is AC)

```
rule A,B => A*B \/ A-B \/ A+B \/ A/B
```

instead of four rules

```
rule A,B => A * B
rule A,B => A - B
rule A,B => A + B
rule A,B => A / B
```

14. **Semantics-based test-case generation.**

    Suppose that you have a given program in a given programming language. Suppose also that the program takes inputs, either directly through dedicated parameters or indirectly through interaction with the environment, or the user, or some thread scheduler, and that you would like to test the program according to various test criteria. What this boils down to ultimately is to provide inputs to the program in order to exercise the desired behaviors when executed.

    Nothing in the above process is specific to any particular programming language; no matter whether it is C, or Java, or JavaScript, you would still want to do the same, namely to generate inputs that exercise the various desired behaviors of the program. In fact, the particular semantic details of the programming language can even confuse you wrt what inputs need to be generated. Consider, for example, the simple C expression fragment `a+(*b)`. It is obvious that you need to generate inputs under which `*b` is defined or not, that is, `b` points to a previously allocated and unfreed memory location or not, but it may be easy, if not careful, to miss checking for behaviors where the sum overflows. Now in C overflow is allowed for unsigned integers, but it is disallowed for normal integers, so in one case you test an interesting behavior while in the other you found a bug in the program.

    In the end, if you want to develop a powerful test-case generation tool for your language that misses no dark corners of the program, you must take into account the entire semantics of the programming language. But why waste time and risk to make mistakes implementing your own understanding of the language semantics in your own test-case generation tool? Following the ideal language framework paradigm, we believe that test-case generation should be a framework feature and not a language one, where each language that has a semantics given in the framework can benefit from it. Note that a semantics is unavoidable anyway, as discussed in the simple example above. But instead of manually projecting it into one particular tool for one particular purpose at a significant effort, we should just define it once and for all, in a rigorous and easy to understand notation, and then have all the tools, including the test-case generator, be developed in a language-independent manner that can be automatically instantiated with any language semantics given as input.

    Sometimes doing things generally helps you not only find cleaner solutions to the problem at hand, but also resolve apparently different problems using the same general machinery. Then you realize that all these problems are instances of the same general principle and you do not even see them as different problems anymore. In our case, the general principle is to generate *program configurations*, where each includes both the program and its input, as well as all the other semantic components, possibly dozens of them, such as memory, stacks, I/O buffers, threads, locks held by thread, and so on. Generating program inputs only

amounts to an instance of the general problem where the program in the k cell is fixed, but its inputs are allowed to vary. Here is another instance of the general problem. Suppose that you already have an interpreter or a compiler for a language, or an analysis tool for it, which was not derived using the ideal language framework approach we that advocate, and that you want to test it. In other words, you would like to execute lots and lots of programs that attempt to cover all the dark corners of your language, hopefully multiple times under various combinations of features. Getting such programs is challenging, though. Some newer languages already come with conformance test suites, such as, for example, JavaScript (https://es5conform.codeplex.com/) . But others, like C or C++, in spite of being already quite complex, have no such well-established test suites. There are companies which sell test suites for C and C++, though; for example you can buy them from Plum Hall, Inc. (http://www.plumhall.com/) for thousands of dollars. Well, using the same general machinery described below, we can not only generate inputs to test programs, but also whole programs and inputs for them to test compilers, interpreters, or other language tools.

The general idea of our sought solution is in fact quite simple and the technique well-established: *narrowing*, or rewriting with unification instead of matching. Indeed, starting with a program configuration with (symbolic) variables, which can also be potentially constrained, or in other words a matching logic pattern, attempt to unify it with the left-hand-side of some semantic rule. If that is possible, then rewrite the symbolic configuration and apply the unifying substitution to the result. Keep doing that systematically, with all the rules, until a desired pattern, say one where the result configuration holds the value 0 in the k cell, or until any desired termination criterion is reached. If your objective is to generate inputs to a given program, then start with a configuration holding the concrete program in the k cell and symbolic variables in the configuration where the desired input is to be generated. If your objective is to generate programs, then put a symbolic variable in the k cell. And you can be arbitrarily creative now: you can generate programs making use of say up to three variables in their environment, using at most 10 heap locations and at most 2 stack frames. All you have to do is to start with the desired configuration pattern and to enable the desired strategies to apply the semantics rules. Which brings us to the actual challenge here. The challenge is to come up with appropriate coverage strategies and to implement them efficiently. We want to cover the semantics as thoroughly as possible when generating new configurations, given the initial constraints (both structural and logical constraints). The success of a solution to this problem will be measured by the quality of the generated programs. For example, in the case of C, can we use the ISO C11 semantics in K to generate millions of C programs that can then reveal bugs in C compilers? Or in the case of JavaScript, ca we use the JavaScript semantics in K to generate more comprehensive test suites than the ES5 conformance test suite? Or even better, can we give semantics to ES6, and thus save the committee from coming up with a conformance test suite first place, because they cannot do better than a semantics-driven test suite anyway?

15. **Symbolic execution framework.**

There are many uses of symbolic execution in the literature. Some in the context of test-case generation others in the context of program verification, some static others dynamic, some available to extend others hardwired in tools, and so on, but all instances of symbolic execution that we are aware of end up being specifically instantiated for one particular language. So you end up with a symbolic execution engine for C, one for LLVM, one for Java, one for JavaScript, and so on and so forth, each implemented by a different team, with a high degree of overlapping and similar challenges. Worse, each presents innovations and engineering solutions which could have very well been applied to other symbolic execution engines developed by other teams for other languages. Even worse, because of this fragmentation, we do not even

have a crystal clear definition of what symbolic execution is in the end. People interested in finding bugs will say that it is a method to find which inputs lead to execution paths that have bugs. People interested in program verification will say that it is a method that allows you to systematically explore all the execution paths in a program to prove bug freedom. Some will say that only the program input is symbolic, but not the program environment, nor the program itself (for example you cannot replace a missing function or a missing concurrent thread or a missing code fragment of a given function with a symbolic variable); others say you can. Some will say that values in the heap can be symbolic, but not the stack frames. Some will say that you can only make values that can be assigned to "program variables" symbolic, others that we can go further and have even the program execution/evaluation context symbolic. And what is a "program variable" anyway, generally speaking? It may mean something in some languages, but in others it is not clear (e.g., in Prolog, or in SQL, or in Haskell, etc.). Some will say that symbolic execution is all about using Hoare logic to extract verification conditions and then solve them with an SMT, others will say "no way!", it is all about extending an interpreter of the language to work with symbolic values instead of concrete ones. So the lack of a clear definition leads to mixing possible implementation of symbolic execution to what it actually is. A big mess, in our view.

Since there is no rigorous definition of what symbolic execution is, we propose one which is general enough to include all the instances of the concept that we are aware of, for all programming languages: *reachability logic without Circularity*``. Recall the following papers presenting reachability logic, the former for all-path and the latter for one-path:

All-Path Reachability Logic

Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

One-Path Reachability Logic

Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

Also recall that Circularity is a proof rule for proving circular behaviors. The rest of the rules can be used to derive any one-path or all-path reachability logic property over any finite execution of any programming language. Reachability rules can contain logical variables of any sorts, which act as symbolic variables, and they can occur anywhere in the program configuration, including in the program itself.

The challenge here is to design and implement a *symbolic excution API* to the K framework. Like all the other K tools, it would take a programming language semantics as input and it would provide a suite of functions, likely hooked to K commands in a console or debugging mode, that should allow a user to do anything they need in terms of symbolic execution. Each of the functions/commands would correspond to certain combinations/heuristics/strategies of the reachability logic proof system, except for Circularity. For example, if one is interested in finding a bug in some execution path, then one should be able to drive the operational semantics based one-path reachability logic proof to cover that path, and then one should be able to solve the accumulated semantic constraints to find a concrete input that lead them there. Adding the Circularity proof rule yields a sound and relative complete procedure to verify any property of any program, as shown in the above papers.

The following papers also address this topic, so you may want to make sure you understand them well if you want to work on this problem:

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu: A Generic Framework for Symbolic Execution. SLE 2013: 281-301 http://dx.doi.org/10.1007/978-3-319-02654-1_16

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu: Symbolic execution based on language transformation. Computer Languages, Systems & Structures 44: 48-71 (2015) http://dx.doi.org/10.1016/j.cl.2015.08.004

Dorel Lucanu, Vlad Rusu, Andrei Arusoaie. A Generic Framework for Symbolic Execution: Theory and Applications Journal of Symbolic Computation, Elsevier, 2016, to appear. A preliminary technical report can be found at https://hal.inria.fr/hal-01238696v2

16. **Symbolic model checking.**

This problem is closely related to the unifying deductive verification and model checking and the symbolic execution framework problems, but it focuses in more depth on the model checking aspect. The idea here is to systematically investigate the various approaches used by the symbolic model checking community and to generalize them to work within our semantic framework. After all, model checkers regard a programming language as a transition system generator, one for each program, and then they implement techniques and optimizations to make the analysis of the transition system efficient and automatic. Well, a programming language semantics in a language framework like K is also a transition system generator, one for each program. Therefore, there is no reason not to develop model checking techniques directly at the level of a semantics framework like K, instead of at the level of each particular language. On the contrary, there are significant benefits for doing so. Besides the obvious fact that this way you get model checkers for all languages for which you have formal semantics, there is also a major engineering pragmatic advantage: each feature or optimization implemented for the generic symbolic model of the framework results in optimizations for all the instances for all the languages. Consider, for example, a smart technique to identify equivalent configurations based on graph isomorphism and alpha-equivalence (see also the aggressive state-reduction challenge below); once implemented generically in the framework, it will yield state-reduction benefits to all model-checker instances for all the languages. Compare that with the approach where each team developing a different model checker for a different language re-implements the same idea in their model checker.

To get a better feel for what we expect to achieve here assume some program in some programming language which, when put into some symbolic configuration as a pattern `PHI`, it shows the following behaviors: `PHI => PHI1` and `PHI => PHI2` are all the direct transitions that `PHI` can perform, then `PHI1 => PHI'` is all `PHI1` can perform and we can prove, say using some SMT solver, that `PHI2 -> PHI` is valid (i.e., `PHI2` implies the original `PHI`). This could happen when, for example, `PHI2` is obtained after the execution of a loop with invariant `PHI` and `PHI1` is the exit configuration pattern. In terms of model checking intuitions, we have just proved above that `PHI => PHI'`, because we have exhaustively analyzed all the behaviors of the program and the only way to start with `PHI` and "exit the loop" is to reach `PHI'`. Well, the above can actually very well be framed as a reachability logic proof, where `PHI => PHI'` is used as a circularity to prove itself.

The challenge here is twofold: (1) to frame existing symbolic model checking techniques and algorithms as instances of using the reachability logic proof system and the Circularity rule, in particular, as shown above; and (2) to develop abstraction and automation techniques, based on existing ideas or develop new ones, to come up with patterns like `PHI` above, which manifests the circular behavior needed to apply the Circularity proof rule. This is also related to the invariant inference challenge.

17. **Invariant/Pattern inference using anti-unification.**

    Finding program invariants is and will continue to be one of the major challenges in program verification and analysis. In terms of reachability logic, that is equivalent to finding all the circularities that are needed to derive a reachability rule. Such circularities, if available, would put a finite bound on the space of the reachability logic proof search, reducing the complexity of proving the original rule to domain reasoning, which can hopefully be discharged using SMT solvers.

    We formulate the invariant/circularity inference problem for reachability logic, or K, as follows. Given a set of matching logic patterns `Phi_1`, `Phi_2`, ..., `Phi_n`, find the strongest pattern `Phi` such that `Phi_i -> Phi` for all `i` from `1` to `n` ("strongest" means that `Phi -> Phi'` for any other pattern `Phi'` with `Phi_i -> Phi'` for all `i` from `1` to `n`). The patterns `Phi_1`, `Phi_2`, ..., `Phi_n` can for example be (concrete or symbolic) configuration snapshots at a particular point in a program configuration, which can be extracted by simply executing the program with the semantics. Since in matching logic there is no distinction between terms and predicates, this problem can also be regarded as an *anti-unification*, or *generalization*, problem. Like for other tools and techniques that we plan to port to the generic level of our semantic framework, we should probably start with the state of the art in invariant inference and try to incorporate it in our framework, and then continue to improve it based on other generic tools and algorithms provided by the framework (e.g., unification, narrowing, state-equivalence, etc.).

18. **Aggressive state/configuration-reduction techniques.**

    One of the major engineering challenges when automating deductive program verification and model checking is to identify when a program state is equivalent to a previously discovered state. In matching logic a weaker property suffices, namely when a pattern *implies* another pattern, but that is irrelevant for this problem/challenge. What we'd like to be able to is to efficiently find out when a pattern is semantically equivalent to another one, so that the new pattern can be discarded from the analysis (because it would yield the same behaviors as the previously discovered equivalent one). Same like in object-oriented programming,

where objects can be equivalent for many semantic reasons, there are also various logical reasons for which two patterns can be considered equivalent. For example, two lambda abstractions `lambda x . x` and `lambda y . y` can be considered equivalent due to alpha-equivalence. Or two symbolic integer expressions `x +Int y` and `y +Int x` can be considered equivalent due to the domain commutativity of `+Int`. Or two almost identical program configurations, with the difference that one allocates program variable `x` to location 100 which then holds value 7, and the other allocates `x` to location 17 which also holds value 7, can be considered equivalent in a language where the location of a variable is not available to the program (unlike in C).

We would like to design and implement a generic approach to pattern equivalence, where users should be allowed to define the precise notion of configuration equivalence that they want for their language. As an analogy, object-oriented languages like Java also allows users to implement their own notion of object equality using methods like `isEqual`. The framework should provide default support for at least well-known equivalences like domain equalities, alpha-conversion and graph isomorphism (in the environment/heap), but in its full generality it should allow users to define their own pattern equality.

19. **Language-independent infrastructure for program equivalence.**

Program equivalence is a hard problem, even when the two programs are in the same language. It is a Pi_2^0-complete problem, in fact, which makes it theoretically as hard as the totality problem for Turing machines: given a Turing machine, does it terminate on all inputs? In order to verify program translators within the same language or across different languages, we would like to develop a theoretical foundation with corresponding tool support for program equivalence. And of course, it should all be language-independent. Specifically, given two language semantics and a (symbolic) configuration pattern in each, when are the two equivalent? Like for reachability logic, we would like to have a sound and relatively complete proof system for this problem. The equivalent of Circularities here would be to conjecture more pairs of equivalent configuration patterns. The resulting technique is expected to be akin to bisimulation, or ciricular coinduction. We have already proposed a sound proof system

A Language-Independent Proof System for Mutual Program Equivalence
Stefan Ciobaca and Dorel Lucanu and Vlad Rusu and Grigore Rosu
ICFEM'14, LNCS 8829, pp 75-90. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/ciobaca-lucanu-rusu-rosu-2014-icfem/ciobaca-lucanu-rusu-rosu-2014-icfem-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-11-05-ICFEM.pdf) , Matching Logic (http://matching-logic.org) , DOI (http://dx.doi.org/10.1007/978-3-319-11737-9_6) , ICFEM'14 (http://icfem2014.uni.lu/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/ciobaca-lucanu-rusu-rosu-2014-icfem/ciobaca-lucanu-rusu-rosu-2014-icfem-ref.bib)

and a prototype tool in K has been implemented that worked with some simple examples, but we were not able to prove the relative completeness of the proof system and have not upgraded the tool to work with the latest version of K. Also, some serious applications are needed (see, for example, the C portability and the LLVM correctness problems below.

In addition to a sound and relatively complete proof system, another major challenge is how to automate the process of checking program equivalence. Ideally, we would like the user to only provide a binary relation between "synchronization" points in the two programs, and then have the framework do the rest

automatically. For example, consider a C program with a loop that you want to prove equivalent to its translation to LLVM, where the loop has been replaced with a conditional jump. You want to only say that the beginning of the C loop and the LLVM label that the jump targets are synchronized, and everything else to be automatically inferred. The problem is non-trivial, but we believe it can be done. In order to lift the user-provided light relation to a relation between symbolic program configurations that can be shown closed under the semantic transition relations, we need to add enough semantic structure to the inferred symbolic configurations that we can execute the corresponding cyclic behaviors. We believe that can be done using *narrowing*, basically starting with very abstract symbolic configurations which are just variables, and then adding more structure and constraints to them as the bodies of the corresponding loops are executed. The resulting configuration patterns are expected to be significantly more abstract than the loop invariants would be, because they need not be strong enough to prove the programs correct.

The above was discussed for program equivalence, but a similar approach should also work for program simulation, where one of the programs has only a subset of the behaviors of the other one. This is typically the case in compilers, where the target program may make choices for otherwise non-deterministic behaviors in the source program.

See also the following papers for a better understanding of the problem:

Dorel Lucanu, Vlad Rusu: Program Equivalence by Circular Reasoning. IFM 2013: 362-377 http://dx.doi.org/10.1007/978-3-642-38613-8_25

Dorel Lucanu, Vlad Rusu: Program equivalence by circular reasoning. Formal Asp. Comput. 27(4): 701-726 (2015) http://dx.doi.org/10.1007/s00165-014-0319-6

20. **Program portability checking.**

As an application of the program equivalence problem above, with immense practical relevance, we would like to develop a C portability checker. Specifically, we would like to take a given C program in a particular instance of C, say for 32 bit word machines, and prove the program equivalent to itself under a different semantics of C, say an idealistic one with arbitrarily large integers. So in this special case of the program equivalence problem, the synchronization points between the two programs are obvious, because the two programs are in fact identical. If the check passes and the program under the different semantics has the same behavior, then the program has no behaviors that depend on the particular C instance, so the program is fully portable.

The above is expected to be an idealistic requirement that probably few programs will respect. To allow more programs to be analyzed and the analyses to be finer grained, we should allow the users to annotate the non-portable code, taking responsibility for it. For example, if a while loop iterates from the MAXINT to 0 and its result depends on the number of iterations, then a user annotation is needed. The point is that the checker is not meant to be very intelligent and infer the user's intention. Its purpose is to let the user know where portability might be possibly broken.

The same can be done for C++, LLVM, and various fragments of C (MisraC, JSF C) and of other languages.

21. **Translation validation, preferably for LLVM.**

    Probably the "killer application" of a powerful infrastructure for proving program equivalence/simulation is translation validation. Translation validation means that each translation/compilation instance is validated (as opposed to verifying the translator/compiler itself, which is a much harder problem). The idea is to augment the translator or compiler to produce not only a target program from the source program, but also a witness that can be used to validate the particular translation. For example, the witness can be the synchronization points between the source and the target program. Since LLVM has been invented at UIUC (in Prof. Vikram Adve (http://web.engr.illinois.edu/~vadve) 's group) and we continue to have significant expertise in LLVM as a department at UIUC, in case you are a UIUC student or collaborator it makes sense to pick LLVM as a target compilation framework for this challenge.

22. **Strategy language for K and reachability logic.**

    In K and in reachability logic, the semantic rewrite rules form a set. That is, their order is irrelevant. While there are strong mathematical reasons to keep them that way in a framework for mathematical semantics and reasoning about programs, this may sometimes be inconvenient when writing K language definitions. For example, you may need to add side conditions to rules to prevent their application in cases where you want the "previous" rules in the definition to apply. In such a situation you may want to tell K to attempt to apply the rules in order, and to never apply a rule that appears later in a definition if a rule that appears earlier applies. Or you may want to split the rules in two groups, and to always apply rules from the first group before attempting rules from the second group. For such reasons, and many others, the term rewriting community has developed *strategy languages* to allow users to control the application of rules. Probably Elan (http://elan.loria.fr/elan.html) was the first rewrite engine that provided a language for describing strategies, and probably Stratego (http://strategoxt.org/Stratego/StrategoLanguage) is the rewrite engine with the most advanced strategies.

    We would like to also add strategies to the K framework. However, due to the variety of tools that it incorporates, and due to its semantics and its uses for program reasoning and verification, we would like to maintain K's core semantics as a *set* of reachability rules. So we would like to "desugar" strategies by translating them back to ordinary rules. For example, a strategy can be seen as a monitor telling which rules are allowed to apply. The monitor state can be stored in some cell specially added for this purpose, and the rewrite rules can then match the monitor state and apply only if allowed to. We believe that all this strategy monitoring infrastructure can be generated automatically from some user convenient and intuitive strategy language. We also believe that the above is possible, even for complex strategies, because of the power of reachability logic to take arbitrary matching logic patterns as LHS of rules. For example, for complex strategies you may need to state that none in a set of terms (the LHS's of a group of rules) match, which is impossible to state in a conventional rewrite engine but is easy to state as a matching logic pattern (the conjunction of the complements of all the LHS patterns).

23. **Systematic comparison of K with other operational approaches.**

K evolved from other approaches, systematically analyzing their advantages and disadvantages, and keeping the advantages and eliminating the disadvantages. To systematically analyze other approaches, we have uniformly represented all of them into one meta-formalism, rewrite logic, and implemented them in Maude:

A Rewriting Logic Approach to Operational Semantics
Traian Florin Serbanuta, Grigore Rosu and Jose Meseguer
Information and Computation, Volume 207(2), pp 305-340. 2009
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic.pdf) , Experiments (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic-experiments.zip) , J.Inf.&Comp. (http://dx.doi.org/10.1016/j.ic.2008.03.026) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-meseguer-2007-ic.bib.txt)

A more detailed discussion and comparison appears in my book draft, which I can send you if you want to (send me a message).

A more direct comparison of K with rewrite logic has been studied in depth, and various translators from K to rewrite logic have been implemented in Maude, meant not only to preserve the execution semantics but also other aspects of the original definition, such as its symbolic execution and verification capabilities:

Language Definitions as Rewrite Theories
Vlad Rusu and Dorel Lucanu and Traian Florin Serbanuta and Andrei Arusoaie and Andrei Stefanescu and Grigore Rosu
J.LAMP, Volume 85(1, Part 1), pp 98-120. 2016
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2016/rusu-lucanu-serbanuta-arusoaie-stefanescu-rosu-2016-jlamp/rusu-lucanu-serbanuta-arusoaie-stefanescu-rosu-2016-jlamp-public.pdf) , project (http://fmse.info.uaic.ro/tools/K-3.4) , DOI (http://dx.doi.org/10.1016/j.jlamp.2015.09.001) , J.LAMP (http://www.journals.elsevier.com/journal-of-logical-and-algebraic-methods-in-programming) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2016/rusu-lucanu-serbanuta-arusoaie-stefanescu-rosu-2016-jlamp/rusu-lucanu-serbanuta-arusoaie-stefanescu-rosu-2016-jlamp-ref.bib)

Language Definitions as Rewrite Theories
Andrei Arusoaie (http://andrei.arusoaie.com/andrei/) and Dorel Lucanu (http://fmse.info.uaic.ro/~dorel.lucanu/) and Vlad Rusu (http://researchr.org/profile/vladrusu) and Traian Florin Serbanuta and Andrei Stefanescu and Grigore Rosu
WRLA'14, LNCS 8663, pp 97-112. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/arusoaie-lucanu-rusu-serbanuta-stefanescu-rosu-2014-wrla/arusoaie-lucanu-rusu-serbanuta-stefanescu-rosu-2014-wrla-public.pdf) , K (http://www.kframework.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-12904-4_5) , WRLA'14 (http://users.dsic.upv.es/workshops/wrla2014/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/arusoaie-lucanu-rusu-serbanuta-stefanescu-rosu-2014-wrla/arusoaie-lucanu-rusu-serbanuta-stefanescu-rosu-2014-wrla-ref.bib)

The Rewriting Logic Semantics Project: A Progress Report
Jose Meseguer and Grigore Rosu
Information and Computation, Volume 231(1), pp 38-69. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/meseguer-rosu-2013-ic/meseguer-rosu-2013-ic-public.pdf) , K (http://k-framework.org) , DOI (http://dx.doi.org/10.1016/j.ic.2013.08.004) , Information and Computation (http://iandc.csail.mit.edu/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/meseguer-rosu-2013-ic/meseguer-rosu-2013-ic-ref.bib)

K-Maude: A Rewriting Based Tool for Semantics of Programming Languages
Traian Florin Serbanuta and Grigore Rosu
WRLA'10, LNCS 6381, pp 104-122. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.pdf) , Slides (PDF) (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla-slides.pdf) , K-Maude (http://k-framework.googlecode.com/) , LNCS (http://dx.doi.org/10.1007/978-3-642-16310-4_8) , WRLA'10 (http://wrla10.ifi.uio.no/) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.bib.txt)

Nevertheless, a crystal clear and shorter than in my book comparison of K to other semantic approaches needs to be also written up and published as a conference/journal paper, for wider dissemination. Indeed, we have encountered colleagues who seem to think that K is some new theoretical development based on new principles. Well, sorry to disappoint, but K is an engineering endeavor attempting to get the best of the ideas developed by the formal semantics community over the last four decades, avoiding their limitations. The main novelties of K, besides its unique notation which appears to admit an elegant semantics in dynamic matching logic, are: (1) its concurrent semantics, which allows for true concurrency even in the presence of sharing; and (2) its configuration abstraction mechanism which is the key for K's modularity.

24. **Configuration abstraction.**

Configuration abstraction is one of the key features of K. It allows K language definitions to be be compact and modular. What we mean by modular in this context is that you should not have to modify the semantics of existing features in your language definition in order to add a new, unrelated features. Consider, for example, a naive big-step SOS definition of an IMP-style language. Then suppose that you want to add abrupt termination to your language, say that you want to stop the program when a division-by-zero takes place. To achieve that, you now have to go through each language construct for which you've already given semantics and add at least as many new rules as arguments that construct has, each propagating the abrupt termination signal generated by that argument through the language construct. This is *terrible*. It quickly discourages you to add anything new to your language, not to mention that it is error-prone. Similarly, the configuration of a programming language often changes as you add new features to the language. For example, if you add exceptions to your language, you may want to add an exception stack to the configuration. Or if you want to add threads to your language, you need to re-organize the configuration so that each thread cell holds its own computation and environment, while all the threads share the same store or heap. In K, you do not need to change existing semantic rules when extending the configuration or when adding new unrelated features to your language. For example, consider an environment-based K semantics of IMP, whose configuration looks as follows

```
configuration <T color="yellow">
                <k color="green"> $PGM:Stmt </k>
                <env color="LightSkyBlue"> .Map </env>
                <store color="red"> .Map </store>
              </T>
```

and whose semantic rule for assignment looks as follows, saying that once the assigned expression evaluates to integer `I`, the assignment statement is dissolved from the computation and `I` is written at the location `N` of `X`.

```
rule <k> X = I:Int; => . ...</k>
     <env>... X |-> N ...</env>
     <store>... N |-> (_ => I) ...</store>
```

Later you add several other features to the language, say concurrent threads and I/O which are completely unrelated to the assignment statement, and your configuration becomes:

```
configuration <T color="yellow">
                <threads color="orange">
                  <thread multiplicity="*" color="blue">
                    <k color="green"> $PGM:Stmts </k>
                    <env color="LightSkyBlue"> .Map </env>
                    <id color="black"> 0 </id>
                  </thread>
                </threads>
                <store color="red"> .Map </store>
                <in color="magenta" stream="stdin"> .List </in>
                <out color="Orchid" stream="stdout"> .List </out>
              </T>
```

In a conventional semantic framework you would have to now also change the semantic rule of the assignment to account for the new configuration structure. But not in K. In K, the same rule that we had still works untouched, thanks to configuration abstraction. Indeed, as explained in some of the K papers, such as

K Overview and SIMPLE Case Study
Grigore Rosu and Traian Florin Serbanuta
K'11, ENTCS 304, pp 3-56. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-serbanuta-2013-k/rosu-serbanuta-2013-k-public.pdf) , K (http://kframework.org) , DOI (http://dx.doi.org/10.1016/j.entcs.2014.05.002) , K'11 (http://www.kframework.org/K11/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-serbanuta-2013-k/rosu-serbanuta-2013-k-ref.bib)

An Overview of the K Semantic Framework
Grigore Rosu and Traian Florin Serbanuta
J.LAP, Volume 79(6), pp 397-434. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap.pdf) , Slides(PPTX) (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap-slides-2011-01-14-Iasi.pptx.zip) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap-slides-2011-01-14-Iasi.pdf) , K Tool (http://k-framework.googlecode.com/) , J.LAP (http://dx.doi.org/10.1016/j.jlap.2010.03.012) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-serbanuta-2010-jlap.bib.txt)

there is only one meaningful way to complete the assignment rule above to match the new configuration, process we call *configuration concretization*, which is

```
rule <threads>...
       <thread>...
         <k> X = I:Int; => . ...</k>
         <env>... X |-> N ...</env>
       ...</thread>
     ...</threads>
     <store>... N |-> (_ => I) ...</store>
```

But this is done automatically by the K kompiler, so you do not have to touch any existing, unrelated rules. In our experience with defining semantics for many languages, large and small, modularity of definitions is perhaps the most important aspect of a practical language semantic framework.

This configuration abstraction mechanism has been explained rather informally so far. We need to formalize it and make it an important component of the semantics of K. One way to look at it is as *rewriting modulo structure*, where certain operations symbols, such as the cell constructs, are considered *structural*, and rewrite rules apply modulo them. Another is to define special evaluation contexts that allow to push the rewrite relations that appear in a rule to their place. Both of these possible solutions can be mathematically explained by means of equations, or (pairs of) rules. For example, the effect of an evaluation context C can be defined equationally through heating/cooling equations of the form `C[R] = R ~> C`.

25. **Defining/Implementing language translators/compilers in K.**

So far, K was mostly used to define language semantics or type systems. But in the end, K is about rewriting program configurations, where a program configuration can be almost anything, depending on the application. There is nothing to stop us from implementing complex program translators, or even full compilers, in K. The program configuration in this case can store various types of information that is needed for the translation, such as symbol tables, jump maps, current allocations, resources available, etc. Then rules can match what they need and transform the program accordingly. With this picture in mind, a language semantics can be regarded as an *extreme program transformer*, where the program is transformed iteratively until nothing is left to transform, the result being some value, or some state, or whatever is relevant in the final configuration.

As a case study, it would be nice to do this for LLVM, for the reasons explained in the translation validation challenge. That is, to implement an optimizing LLVM compiler in K. Moreover, to also implement some frontend translators to LLVM, say from C to LLVM, and also some backends translators, say from LLVM to x86.

Besides simplicity and elegance, an addition benefit of doing things this way is that it also gives us a formal basis for verifying such translators or compilers. Indeed, assuming a semantics for the source language and one for the target language, to prove a translator/compiler from the former to the latter correct, we would need to show that whatever syntactic transformation it does, it also preserves/refines the semantics of the original language. This is typically done by extending the translator to consider entire semantic configurations of the first language, not only programs, and then showing that the extended translator preserves the meaning of each semantic rule in the source language semantics. This should be done using reachability logic verification. That is, each semantic rule from the source language translates into a reachability rule in the target language, which then needs to be proved correct using the target language semantics.

Verifying compilers like described above is not easy. A translation validation approach may be preferable, at least as a starting point. Having the translator/compiler defined or implemented in K should also help with the translation validation approach, because we can validate each translation instance as is generated, without leaving the framework.

26. **Translations from K to other languages or formalisms.**

The very first uses of what we call K today were as a definitional methodology in Maude:

CS322 - Programming Language Design: Lecture Notes
Grigore Rosu
Technical Report UIUCDCS-R-2003-2897, December 2003
PDF (http://fsl.cs.uiuc.edu/pubs/UIUCDCS-R-2003-2897.pdf) , TR @ UIUC (http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2003-2897) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2003-cs322.bib.txt)

The Rewriting Logic Semantics Project
Jose Meseguer and Grigore Rosu
J. of TCS, Volume 373(3), pp 213-237. 2007
PDF (http://fsl.cs.uiuc.edu/pubs/meseguer-rosu-2006-tcs.pdf) , J.TCS (http://dx.doi.org/10.1016/j.tcs.2006.12.018) , BIB (http://fsl.cs.uiuc.edu/pubs/meseguer-rosu-2006-tcs.bib.txt)

The fact that we could define rather complex languages without conditional rules, thanks to the then-K-methodology, made us think that in fact conditional rules can be automatically eliminated. We tried hard to do so, first in

From Conditional to Unconditional Rewriting
Grigore Rosu
WADT'04, LNCS 3423, pp 218-233. 2004
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2004-wadt.pdf) , LNCS (http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3423&spage=218) , WADT'04 (http://www.lsi.upc.es/etaps04/wadt2004/index.html) , Experiments (http://fsl.cs.uiuc.edu/pubs/rosu-2004-wadt-experiments.zip) , PDF - Original submission (http://fsl.cs.uiuc.edu/pubs/rosu-2004-wadt-abstract.pdf) , WADT'04 slides (http://fsl.cs.uiuc.edu/pubs/rosu-2004-wadt-slides.ppt) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2004-wadt.bib.txt)

and then in

Computationally Equivalent Elimination of Conditions - extended abstract
Traian Florin Serbanuta and Grigore Rosu
RTA'06, LNCS 4098, pp 19-34. 2006
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2006-rta.pdf) , Slides (PPT) (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2006-rta-slides.ppt) , Experiments (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2006-rta-experiments.zip) , LNCS (http://dx.doi.org/10.1007/11805618_3) , RTA'06 (http://www.easychair.org/FLoC-06/RTA.html) , DBLP (http://ftp.informatik.uni-trier.de/~ley/db/conf/rta/rta2006.html#SerbanutaR06) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2006-rta.bib.txt)

hoping that K could be seen as a general translator from conditional to unconditional rules. Then we ran into quite difficult technical problems following this path, related to associative operators, which didn't even have anything to do with K as we envisioned it at that time, so we gave up that approach. Instead, we started regarding K as a language semantic framework in itself:

K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation
Grigore Rosu
Technical Report UIUCDCS-R-2005-2672, December 2005
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2005-tr.pdf) , Experiments (http://fsl.cs.uiuc.edu/pubs/rosu-2005-tr-experiments.zip) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2005-tr.bib.txt)

K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation
Grigore Rosu
Technical report UIUCDCS-R-2006-2802, December 2006
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2006-tr-c.pdf) , TR@UIUC (http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2006-2802) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2006-tr-c.bib.txt)

K: A Rewriting-Based Framework for Computations -- Preliminary version
Grigore Rosu
Technical report UIUCDCS-R-2007-2926 and UILU-ENG-2007-1827, December 2007
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2007-tr-c.pdf) , ZIP (http://fsl.cs.uiuc.edu/pubs/rosu-2007-tr-c.zip) , TR@UIUC (http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2007-2926) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2007-tr-c.bib.txt)

We had our first implementations of K as translations to Maude:

K-Maude: A Rewriting Based Tool for Semantics of Programming Languages
Traian Florin Serbanuta and Grigore Rosu
WRLA'10, LNCS 6381, pp 104-122. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.pdf) , Slides (PDF) (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla-slides.pdf) , K-Maude (http://k-framework.googlecode.com/) , LNCS (http://dx.doi.org/10.1007/978-3-642-16310-4_8) , WRLA'10 (http://wrla10.ifi.uio.no/) , BIB (http://fsl.cs.uiuc.edu/pubs/serbanuta-rosu-2010-wrla.bib.txt)

The K Primer (version 2.5)
Traian Florin Serbanuta, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu and Grigore Rosu
Technical Report, January 2012
PDF (http://fsl.cs.uiuc.edu/pubs/k-primer-2012-v25.pdf) , K 2.5 (http://k-framework.googlecode.com/svn/tags/v2.5/) , BIB (http://fsl.cs.uiuc.edu/pubs/k-primer-2012-v25.bib.txt)

K Framework Distilled
Dorel Lucanu, Traian Florin Serbanuta and Grigore Rosu
WRLA'12, LNCS 7571, pp 31-53. 2012 ***Invited Paper***
PDF (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-serbanuta-2012-wrla.pdf) , Slides (PDF) (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-serbanuta-2012-wrla-slides.pdf) , WRLA'12 (http://wrla2012.lcc.uma.es/) , LNCS (http://dx.doi.org/10.1007/978-3-642-34005-5_3) , BIB (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-serbanuta-2012-wrla.bib.txt)

Executing Formal Semantics with the K Tool
David Lazar, Andrei Arusoaie, Traian Florin Serbanuta, Chucky Ellison, Radu Mereuta, Dorel Lucanu and Grigore Rosu
FM'12, LNCS 7436, pp 267-271. 2012

PDF (http://fsl.cs.uiuc.edu/pubs/lazar-arusoaie-serbanuta-ellison-mereuta-lucanu-rosu-2012-fm.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/fm-2012-slides.pdf) , FM'12 (http://fm2012.cnam.fr/) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/fm/fm2012.html#LazarASEMLR12) , BIB (http://fsl.cs.uiuc.edu/pubs/lazar-arusoaie-serbanuta-ellison-mereuta-lucanu-rosu-2012-fm.bib.txt)

That gave us an excellent basis for experimentation and for defining K semantics of several languages, but it quickly turned out to be more inefficient than we liked it to, mainly because of the huge cost to pay for nested matching modulo associativity and commutativity as needed for K's configurations. We then implemented a Java rewrite engine customized to K's needs, as well as a modified kompiler that translated the original definition into one that could be handled by our Java backend engine:

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

KJS: A Complete Formal Semantics of JavaScript
Daejun Park and Andrei Stefanescu and Grigore Rosu
PLDI'15, ACM, pp 346-356. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/park-stefanescu-rosu-2015-pldi/park-stefanescu-rosu-2015-pldi-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-16-park-stefanescu-rosu-PLDI.pdf) , Semantics (https://github.com/kframework/javascript-semantics) , DOI (http://dx.doi.org/10.1145/2737924.2737991) , PLDI'15 (http://conf.researchr.org/home/pldi2015) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/park-stefanescu-rosu-2015-pldi/park-stefanescu-rosu-2015-pldi-ref.bib)

Our Java engine is significantly more complex than a conventional rewrite engine, because it is tightly connected to an SMT (Z3) and it also provides support for symbolic execution and reasoning within domains supported by Z3. But because it specializes to K, the resulting K implementation with the Java backend ended up being more than one order of magnitude faster than the previous version with the Maude backend. For execution performance specifically, the RV-Match (https://runtimeverification.com/match) tool implements an OCAML backend to execute the C semantics on ordinary C programs, which consists of a translator from K to OCAML followed by compilation to native code, which turned out to be more than two orders of magnitude faster than the Java backend. The reason the OCAML backend is so muck faster is because it takes advantage of OCAML's efficient implementation of case matching, based on advanced indexing. But even with all this added efficiency thanks to OCAML, the RV-Match tool is still five orders of magnitude slower than the actual C programs compiled with gcc and executed natively.

We believe we should be able to translate K definitions to any functional language or theorem prover following the same approach as for the OCAML backend, such as to Haskell or Coq. This would give us models of the language defined in K in any of these languages. In the case of the functional languages, the

resulting models can be used for execution or for integration with other tools, while in the case of the theorem provers for mechanized reasoning about the programming language or about programs of it. We believe the same can be done even for LLVM, by customizing and implementing indexing ourselves as part of the translator (instead of relying on the target language's indexing), in which case we can obtain very efficient interpreters. I conjecture that the resulting interpreter can be even two orders of magnitude faster than the current OCAML backend of RV-Match, and thus the C semantics to execute C programs "only" three orders of magnitude slower than if the programs were compiled with gcc and executed natively. To go even faster, I think we need to generate compilers from semantics.

27. **Semantics-based Compiler Generation.**

Generating a language compiler from a language semantics could be one of the most practical applications of a semantic framework. Indeed, it is a lot easier to define a language semantics than to implement a compiler. Besides, since the compiler is generated using the mathematical semantics of the language, the generated compiler can either be correct-by-construction or a proof can be generated for each language instance (in a translation validation style, but for the compiler itself). In other words, we not only would get compilers cheaply, from language semantics, but the compilers we get also would be provably correct. If this can be made to work in a way that the generated compilers are also efficient, it can completely change the way we as a community design and implement programming languages.

But is this really feasible? How can we even start such a project? We actually believe that it is not hard. We expect it to be a nice application of the already existing infrastructure that K provides. Consider a C-like source language and LLVM as a target language. Suppose that we know which instructions in the source language have a circular behavior, and suppose that we already know how to translate those to corresponding instructions in the target language; for example, a while loop in C is translated in a very specific and trivial way to a conditional jump in LLVM. Now we can split the source program into statements with circular behaviors and blocks of other statements in between or inside them. Further, we can regard each such block of instructions as one generalized instruction that accumulates the semantics of all composing instructions. We can express its accumulated semantics as a K reachability rule. Now the challenge is how to generate efficient target language code for a K reachability rule. And we strongly believe that this is indeed possible. Consider for example the following code in a C-like language:

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

The `while` will be translated into an appropriate conditional jump, and the accumulated semantics of the loop body can be expressed with a rule as follows:

```
 rule <k> s+=n;n--; => . ...</k>
      <state>... s |-> (S => S +Int N), n |-> (N => N -Int 1) ...</state>
```

It should not be difficult to translate the effect of this rule on the `state` into LLVM code that does precisely that.

28. **Certification of proofs done using the K framework.**

One of the major features of K is that we can derive deductive program verifiers, model checkers, and possibly other program reasoning tools, all correct-by-construction, where the only trusted base is the operational semantics of the programming language in question. These program reasoning tools are indeed correct-by-construction as supported by their underlying theory, but their implementation can have errors. For example, imagine that you verity a complex C program using the K framework instantiated with the C semantics, and that your program verifies "automatically" and after 15 minutes of waiting you get the much desired *proved* message. Why should you trust that you have indeed verified your program? You have confidence in the C semantics,

Defining the Undefinedness of C
Chris Hathhorn and Chucky Ellison and Grigore Rosu
PLDI'15, ACM, pp 336-345. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/hathhorn-ellison-rosu-2015-pldi/hathhorn-ellison-rosu-2015-pldi-public.pdf) , C Semantics (https://github.com/kframework/c-semantics) , DOI (http://dx.doi.org/10.1145/2813885.2737979) , PLDI'15 (http://conf.researchr.org/home/pldi2015) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/hathhorn-ellison-rosu-2015-pldi/hathhorn-ellison-rosu-2015-pldi-ref.bib)

An Executable Formal Semantics of C with Applications
Chucky Ellison and Grigore Rosu
POPL'12, ACM, pp 533-544. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl-slides.pdf) , Project (http://c-semantics.googlecode.com) , ACM (http://dl.acm.org/citation.cfm?doid=2103656.2103719) , POPL'12 (http://www.cse.psu.edu/popl/12/) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/popl/popl2012.html#EllisonR12) , BIB (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl.bib.txt)

because it was tested on tens of thousands of programs, and you have confidence in the soundness of reachability logic because that has been proved mathematically and mechanically in Coq:

All-Path Reachability Logic
Andrei Stefanescu and Stefan Ciobaca and Radu Mereuta and Brandon Moore and Traian Florin Serbanuta and Grigore Rosu
RTA'14, LNCS 8560, pp 425-440. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-07-16-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.1007/978-3-319-08918-8_29) , RTA'14 (http://vsl2014.at/pages/RTATLCA-cfp.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta/stefanescu-ciobaca-mereuta-moore-serbanuta-rosu-2014-rta-ref.bib)

One-Path Reachability Logic
Grigore Rosu and Andrei Stefanescu and Stefan Ciobaca and Brandon Moore
LICS'13, IEEE, pp 358-367. 2013

PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-27-LICS.pptx) , Reachability Logic (http://fsl.cs.uiuc.edu/RL) , LICS'13 (http://lii.rwth-aachen.de/lics/lics13/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/rosu-stefanescu-ciobaca-moore-2013-lics/rosu-stefanescu-ciobaca-moore-2013-lics-ref.bib)

But how do you know that K, which at the time of this writing (Feb 2016) has about 100k LOC, implements these correctly? In fact, we keep discovering bugs in K, so it is quite likely that K has bugs.

So what can we do to have absolute confidence in a program verified using the K framework? First, note that there is nothing that can be done to have absolute confidence in a language semantics, simply because that acts as the definition of the language, and definitions are correct by definition, if we can say so. You can execute more tests, or even show it equivalent to other semantics for the same language if another one exists, but in the end, no matter what you do, you will have to simply assume that the language semantics is correct. However, we would like the language semantics to be the only thing that we have to trust. Everything that we execute or prove with K is, in the end, a reachability logic proof derivation, specifically a proof derivation using the eight proof rules of reachability logic and the language semantics as axioms. We should be able to augment K to produce not only the final result, but also evidence of how the proof has been constructed. Such evidence, or proof witness, should be a in form that makes it easy to check in a trusted manner, for example by a third-party proof checker. One way to do this is to build upon the relationship between Circularity and coinduction, and to have K produce a proof tactic that a mechanical theorem prover like Coq or Isabelle can use to reconstruct the entire proof. One source of concern with this approach is that we should still trust the translation of the axioms from K to Coq, but hopefully that can be mitigated by re-executing all the tests that were used to validate the K semantics also with the translated Coq semantics. Another source of concern is the gaps in the original proof that are due to invocations of decision procedures, such as matching modulo associativity, commutativity, etc., or SMT solver decisions. Another way is develop a separate proof checker for reachability logic, and then to have K generate proof objects following the syntax expected by the reachability logic proof checker. Since reachability logic has only either proof rules, such a proof checker is not expected to be too complex. This approach would also suffer from the same problem of proof gaps like the other approach. We have done some research on this topic in the context of equational proofs, which we believe can be extended to work for reachability logic:

Certifying and Synthesizing Membership Equational Proofs
Grigore Rosu and Steven Eker and Patrick Lincoln and Jose Meseguer
FME'03, Lecture Notes in Computer Science (LNCS) 2805, pp 359-380. 2003
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2003/rosu-eker-lincoln-meseguer-2003-fme/rosu-eker-lincoln-meseguer-2003-fme-public.pdf) , Slides(PPT) (http://fslweb.cs.illinois.edu/FSL/presentations/2003/2003-09-10-FME.ppt) , Maude (http://maude.cs.illinois.edu/) , DOI (http://dx.doi.org/10.1007/978-3-540-45236-2_21) , FME'03 (http://fm03.isti.cnr.it/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2003/rosu-eker-lincoln-meseguer-2003-fme/rosu-eker-lincoln-meseguer-2003-fme-ref.bib)

Regardless of the approach, the size of the proof object itself can be a problem. The simpler and thus easier to trust the proof checker, the more detailed and thus larger the proof is. The right balance will need to be found to keep certification practical.

29. **K semantics to new real languages.**

If you are practically inclined, defining a K semantics to a new real language is probably the best entry point to the K framework. It is a well-defined problem and can serve as a B.S., M.S, and even a Ph.D. topic, and in our experience it is relatively easy to publish. So far we gave and published semantics to C (ISO C11):

Defining the Undefinedness of C
Chris Hathhorn and Chucky Ellison and Grigore Rosu
PLDI'15, ACM, pp 336-345. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/hathhorn-ellison-rosu-2015-pldi/hathhorn-ellison-rosu-2015-pldi-public.pdf) , C Semantics (https://github.com/kframework/c-semantics) , DOI (http://dx.doi.org/10.1145/2813885.2737979) , PLDI'15 (http://conf.researchr.org/home/pldi2015) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/hathhorn-ellison-rosu-2015-pldi/hathhorn-ellison-rosu-2015-pldi-ref.bib)

An Executable Formal Semantics of C with Applications
Chucky Ellison and Grigore Rosu
POPL'12, ACM, pp 533-544. 2012
PDF (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl-slides.pdf) , Project (http://c-semantics.googlecode.com) , ACM (http://dl.acm.org/citation.cfm?doid=2103656.2103719) , POPL'12 (http://www.cse.psu.edu/popl/12/) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/popl/popl2012.html#EllisonR12) , BIB (http://fsl.cs.uiuc.edu/pubs/ellison-rosu-2012-popl.bib.txt)

to JavaScript (ES5):

KJS: A Complete Formal Semantics of JavaScript
Daejun Park and Andrei Stefanescu and Grigore Rosu
PLDI'15, ACM, pp 346-356. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/park-stefanescu-rosu-2015-pldi/park-stefanescu-rosu-2015-pldi-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-16-park-stefanescu-rosu-PLDI.pdf) , Semantics (https://github.com/kframework/javascript-semantics) , DOI (http://dx.doi.org/10.1145/2737924.2737991) , PLDI'15 (http://conf.researchr.org/home/pldi2015) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/park-stefanescu-rosu-2015-pldi/park-stefanescu-rosu-2015-pldi-ref.bib)

to Java (Java 1.4):

K-Java: A Complete Semantics of Java
Denis Bogdanas and Grigore Rosu
POPL'15, ACM, pp 445-456. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/bogdanas-rosu-2015-popl/bogdanas-rosu-2015-popl-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-01-16-K-Java-POPL/2015-01-16-K-Java-POPL.pdf) , K-Java (https://github.com/kframework/java-semantics) , DOI (http://dx.doi.org/10.1145/2676726.2676982) , POPL'15 (http://popl.mpi-sws.org/2015/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/bogdanas-rosu-2015-popl/bogdanas-rosu-2015-popl-ref.bib)

to Verilog:

A Formal Executable Semantics of Verilog
Patrick Meredith, Michael Katelman, Jose Meseguer and Grigore Rosu
MEMOCODE'10, IEEE, pp 179-188. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/meredith-katelman-meseguer-rosu-2010-memocode.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/memocode_presentation.pdf) , Sources (http://fsl.cs.uiuc.edu/images/c/c7/Verilog_rls.zip) , Verilog Semantics (http://fsl.cs.uiuc.edu/index.php/Verilog_Semantics) , IEEE (http://dx.doi.org/10.1109/MEMCOD.2010.5558634) , MEMOCODE'10 (http://www-memocode2010.imag.fr/) , BIB (http://fsl.cs.uiuc.edu/pubs/meredith-katelman-meseguer-rosu-2010-memocode.bib.txt)

to Scheme:

A K Definition of Scheme
Patrick Meredith, Mark Hills and Grigore Rosu
Technical Report UIUCDCS-R-2007-2907, October 2007
PDF (http://fsl.cs.uiuc.edu/pubs/meredith-hills-rosu-2007-tr-b.pdf) , TR@UIUC (http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2007-2907) , BIB (http://fsl.cs.uiuc.edu/pubs/meredith-hills-rosu-2007-tr-b.bib.txt)

The semantics of Verilog and of Scheme are very old, and back then we used an old version of K that is neither supported anymore nor powerful enough to do the various types of analysis that the new version can do. It would be very nice to upgrade their semantics to the latest version of K, do some experiments with model checking and program verification, and replublish them in some high-impact avenues.

In addition to the above, large to almost complete semantics of Python, LLVM, Haskell and OCaml light have also been defined, but they were not published yet. If interested in a quick paper, please contact me to let you know about the status of these projects and how you can contribute.

Of particular importance are languages related to the web. That's because these languages are in charge of enforcing security policies, and security policies can be easily broken if the language is incorrectly designed or implemented, or if the libraries used in security-critical code are incorrectly implemented. For example, our semantics of JavaScript ES5 above has been used to find bugs in all major browsers and implementations of JavaScript. Ideally, we would like to get involved with the design of future web languages, so that these language start with a formal semantics that can also serve as a reference implementation, instead of with adhoc implementations. We believe the current performance of the OCaml backend of K is good enough, meaning that a formal semantics can already provide a reasonably fast implementation for free, to serve as a language model. Particular languages of interest here are JavaScript ES6 and WASM.

Also of particular importance are unconventional languages for which no adequate verification techniques or tools are available. For example, consider aspect-oriented languages, say AspectJ. There are many tools and systems developed using AspectJ, and it is not at all clear how to formally analyze or verify such systems and there are certainly no available tools for such a task. On the other hand, it should not be hard to give a formal semantics to AspectJ, on top of the existing Java semantics. Through its strategies, K even allows you to do that modularly. Then, once a K formal semantics of AspectJ exists, we should be able to use precisely the same verification and analysis machinery that the K framework provides to verify AspectJ programs.

Finally, we also believe that semantics of low-level languages, like assembler languages for x86 or RISC processors, are low hanging fruits for several reasons. First, these languages are comparatively simple, so it should not be difficult to give them a semantics (provided one does not go into even lower level details, such as caching, which would be needed for certain types of analysis, e.g., WCET). Second, many of these language have no formal semantics, so a semantics for them would be welcome. Third, most of these languages have no program verifiers, or model checkers, or symbolic execution engines, and a K semantics would provide all these tools for these languages. And fourth, a semantics for such a language is indispensable if one wants to verify compilers that target that language.

Here are several K semantics to real languages defined by other groups (please let me know if you want me to add your language/paper here):

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu: Towards a K Semantics for OCL. Electr. Notes Theor. Comput. Sci. 304: 81-96 (2014) http://dx.doi.org/10.1016/j.entcs.2014.05.004

Vlad Rusu, Dorel Lucanu: A K-Based Formal Framework for Domain-Specific Modelling Languages. FoVeOOS 2011: 214-231 http://dx.doi.org/10.1007/978-3-642-31762-0_14

30. **Verification of real-time languages, WCET verification.**

Verification of programs in real-time languages, that is showing not only the functional correctness of the program but also that its time guarantees are met, as well as verification of worst case execution time (WCET), are considered to be very complex topics both from a foundational and from a practical point of view. Indeed, conventional verification techniques work for functional correctness, but they do not apply off-the-shelf for proving time bounds of programs. Existing general purpose verification tools are inapplicable, while tools for verifying such properties tend to be very specialized and obscure.

To verify such properties with our approach, all we need to do is to define an operational semantics for the target programming language that takes time into account. For example, recall the assignment rule in an IMP-like language:

```
rule <k> X = I:Int => I ...</k> <state>... X |-> (_ => I) ...</state>
```

Now suppose that we also want to keep track of time in the semantics, and that the assignment operation takes 3 units (e.g., one unit for reading the value to assign, one unit for writing it to the variable, and another unit for jumping to the next instruction). All we have to do is to add a new cell to the semantics, say called `time`, and then modify the rules to increment the time appropriately, for example:

```
rule <k> X = I:Int => I ...</k> <state>... X |-> (_ => I) ...</state> <time> T => T +Int 3 </time>
```

Now we can refer to the time a program takes in reachability rules, at no additional infrastructure effort.

The bottom line is that we do not need to do anything special for verifying timed properties in our semantics-based verification approach. We have published a workshop paper on this topic, where we considered also interrupts:

Low-Level Program Verification using Matching Logic Reachability
Dwight Guth and Andrei Stefanescu and Grigore Rosu
LOLA'13. 2013
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/guth-stefanescu-rosu-2013-lola/guth-stefanescu-rosu-2013-lola-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2013/2013-06-29-LOLA.pdf) , Matching Logic (http://matching-logic.org/) , LOLA'13 (http://research.microsoft.com/en-us/events/lola2013/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/guth-stefanescu-rosu-2013-lola/guth-stefanescu-rosu-2013-lola-ref.bib)

And we also published a paper specifically on how to do WCET using a semantics-based approach:

Towards Semantics-Based WCET Analysis
Mihail Asavoae and Dorel Lucanu and Grigore Rosu
WCET'11. 2011
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2011/asavoae-lucanu-rosu-2011-wcet/asavoae-lucanu-rosu-2011-wcet-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2011/2011-07-WCET.pdf) , Matching Logic (http://matching-logic.org) , WCET'11 (http://www.artist-embedded.org/artist/Overview,2317.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2011/asavoae-lucanu-rosu-2011-wcet/asavoae-lucanu-rosu-2011-wcet-ref.bib)

But we need a more through study on this topic, investigate the literature, and in the end make a convincing argument that verification of timed properties in real-time programming languages came at no additional complexity besides what is absolutely necessary and unavoidable anyway if mathematical rigor is desired, namely to define a formal semantics of the target language.

31. **Module system for K.**

Large language definitions need to be decomposed in modules. An obvious reason is to better understand them. A more subtle reason is that different modules achieve different tasks and there is no one module that includes all the functionality. Consider, for example, a language like C. In order to parse actual C programs, you need a very specialized syntax, which may even be considered ugly or unsuitable for semantic purposes. For example, you may want to define a left-recursive grammar to efficiently parse comma-separated lists of expressions:

```
module C-SYNTAX-FOR-PARSING-PROGRAMS
...
 syntax NeExps ::= NeExps "," Exp | Exp    // non-empty expressions
 syntax Exps   ::= NeExp | ""              // empty or non-empty expressions
...
```

On the other hand, when giving semantics to C, you would prefer to work with associative lists of expressions, probably something like this:

```
module C-SYNTAX-FOR-SEMANTICS
...
  syntax Exps ::= "." | Exp | Exps "," Exps [assoc id:.]
...
```

This way, in the semantics, you do not need to consider all the cases used for parsing, and you can match anywhere inside a list of expressions. You also want to have another module where you convert the parsed program from the original (abstract) syntax to the semantics (abstract) syntax:

```
module TRANSLATE
  import C-SYNTAX-FOR-PARSING-PROGRAMS
  import C-SYNTAX-FOR-SEMANTICS
  syntax Exps@C-SYNTAX-FOR-SEMANTICS ::= translate(Exps@C-SYNTAX-FOR-PARSING-PROGRAMS)
  rule translate() => .
  rule translate(E:Exp) => E
  rule translate(NeEs,E) => translate(Nes),E
endmodule
```

Now you can define your C semantics module by importing only the desired `C-SYNTAX-FOR-SEMANTICS` module, and making sure that you parse the program you want to execute with the right module and translate it accordingly, without ever polluting the semantics with syntax necessary only for parsing:

```
module C-SEMANTICS
  import C-SYNTAX-FOR-SEMANTICS
  configuration ... <k> #rewrite(TRANSLATE, translate(#parse(C-SYNTAX-FOR-PARSING-PROGRAMS,$PGM))) </k> ...
  ...
endmodule
```

While K currently provides modules, it collapses all the sorts of the imported modules into their union, without qualifying them with their defining modules, so the module TRANSLATE above cannot be defined yet. If qualification is not desired (this is still being debated in our design of K; Maude, for example, does not do it) but we add renaming to K, which we want to do anyway, then we could still solve the problem:

```
module TRANSLATE
  import C-SYNTAX-FOR-PARSING-PROGRAMS * (Exps to ParsingExps)
  import C-SYNTAX-FOR-SEMANTICS
  syntax Exps ::= translate(ParsingExps)
  rule translate() => .
  rule translate(E:Exp) => E
  rule translate(NeEs,E) => translate(Nes),E
endmodule
```

Another issue that we have to discuss and decide is whether we want parametric modules in K. Again, renaming seems to be powerful enough to resolve problems that are typically resolved with parametric modules, so it is not clear.

Take a look at Maude's module system (http://maude.lcc.uma.es/manual/maude-manualch6.html#x41-740006) , especially its parametric modules. Take also a look at some theoretical work we've done in this area, on giving formal semantics to modules and operations on them using category theory:

Abstract Semantics for K Module Composition

Codruta Girlea and Grigore Rosu
K'11, ENTCS 304, pp 127-149. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2013/girlea-rosu-2013-k/girlea-rosu-2013-k-public.pdf) , K (http://kframework.org) , DOI (http://dx.doi.org/10.1016/j.entcs.2014.05.007) , K'11 (http://www.kframework.org/K11/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2013/girlea-rosu-2013-k/girlea-rosu-2013-k-ref.bib)

Towards a Module System for K
Mark Hills and Grigore Rosu
WADT'08, LNCS 5486, pp 187-205. 2009
PDF (http://fsl.cs.uiuc.edu/pubs/hills-rosu-2008-wadt-b.pdf) , LNCS (http://dx.doi.org/10.1007/978-3-642-03429-9_13) , WADT'08 (http://www.di.unipi.it/WADT2008/) , BIB (http://fsl.cs.uiuc.edu/pubs/hills-rosu-2008-wadt-b.bib.txt)

Composing Hidden Information Modules over Inclusive Institutions
Joseph Goguen and Grigore Rosu
Dahl's Festschrift, LNCS 2635, pp 96-123. 2004
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2004/goguen-rosu-2004-dahl/goguen-rosu-2004-dahl-public.pdf) , DOI (http://dx.doi.org/10.1007/978-3-540-39993-3_7) , Dahl's Festschrift (http://dx.doi.org/10.1007/b96089) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2004/goguen-rosu-2004-dahl/goguen-rosu-2004-dahl-ref.bib)

Abstract Semantics for Module Composition
Grigore Rosu
Technical Report http://roger.ucsd.edu/record=b7304233~S9, May 2000
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2000/rosu-2000-tr/rosu-2000-tr-public.pdf) , DOI (http://roger.ucsd.edu/record=b7304233~S9) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2000/rosu-2000-tr/rosu-2000-tr-ref.bib)

A good understanding of the Maude module system and of the papers above should provide a solid basis for approaching this problem.

32. **Semantics-based debugging: from debugging to verification.**

Debugging tools can also be developed to a large extent language-independently. Moreover, in combination with support for symbolic execution and conjecturing (circularity) claims, debugging can be smoothly extended into an interactive program verification.

So suppose that you have a semantics to your programming language, and that you have a program in your language in which we know or you believe that you have a bug. What you typically do with a conventional debugger, you can also do with the language semantics. For example, you may want to execute the program step-by-step and to observe how the state changes at each step. There is better way to see that than by tracing the rewrite-based execution of your semantics. Indeed, this way you can literally see *everything* in the program configuration, not only what a particular, adhoc language-specific debugging tool allows you to see. For example, you can browse through the stack frames of each of the threads alive, or through the output generated so far, or you can check which threads can progress and which are blocked, or what are the next possible execution steps, and so on. And storing the sequence of rewrite rule applications, you can jump back and forth through the execution trace and see how the configuration/state looked like at each moment in

the past, or advance as many steps as you wish in the future. Also, you can obviously set "breakpoints" in the program, by simply inserting an artificial instruction there which has no semantics, so the execution will stop there for you to see the entire configuration. Together with some relatively simple and still language-independent visualization mechanism, such as hiding thinks that you do not care about in the configuration, for example replacing them with "...", we should be able to offer a language-independent debugger in the K framework that offers all the features that conventional debuggers offer.

But we can go a lot further than what a conventional debugger can do! For example, how about executing the program until the configuration matches a certain pattern of interest, where the pattern can be any matching logic formula? With such a feature, you can set *semantic breakpoints*. Here are some examples: breakpoint when program variables `x` and `y` have the same value; or when you reach the 10th function call; or whem function <t>f</tt> is called with value `7`; or when `x` points to a singly linked list in the heap; or then the heap can be organized as a circular list; etc.

And we can go even much further! We can execute the program symbolically, and still do all the above. This allows you to cover a whole bunch of inputs in one run. Moreover, you can symbolically investigate different paths in your program in search for the bug systematically, picking points in the program that you want to cover and then have the generic symbolic execution infrastructure find the constraints under which those points are reachable, and solve the constraints for you to give you the concrete inputs that would get you there.

And we can go even much much further! If you do not want to waste your time waiting for a loop to execute many iterations, or avoid undecidability issues when executing it symbolically in order to find a bug that lurks after the loop, a semantics-based debugging infrastructure based on reachability logic should allow you to simply provide a reachability rule that summarizes the loop in question. Then you do not have to execute the loop, you can apply its corresponding reachability rule and then continue to execute the code after the loop in search for your bug. And if you want to also be absolutely sure that your reachability rule abstraction for the loop is correct, then you can do the same for the loop itself, that is execute it symbolically to check the claimed reachability rule in hopes to find bugs in the loop.

And we can go all the way and even verify the program! Indeed, there is not much left to do on top of the above in order to verify your program. You can start with a set of reachability rules that you want to verify against the program, or to debug the program against them if you like this perspective better, then add some more reachability claims if you want to, such as for some loops and some functions, and then check/debug each of them separately, using any of them as a shortcut whenever possible during the process. If you find any problem in the process, then you either found a bug in the program or you claimed wrong properties, so you need to fix something. If you manage to complete checking all of them, then congratulations, you just verified your program!

So semantics-based debugging is not only possible, but it can go where no other debugger ever dreamed of going.

We have started designing and developing such a debugger, but there is still a lot of work left. You can see our current progress on our K Debugger (https://github.com/kframework/k/wiki/K-Debugger) wiki.

33. **Type systems and abstract interpretations in K and reachability logic.**

As shown in the K tutorial (https://github.com/kframework/k/tree/master/k-distribution/tutorial) , type systems or static semantics can be defined for languages following the same approach that we use for the semantics: rewrite rules. For technical details, check the Technical Reports on K, starting with the 2003 lecture notes:

CS322 - Programming Language Design: Lecture Notes
Grigore Rosu
Technical Report UIUCDCS-R-2003-2897, December 2003
PDF (http://fsl.cs.uiuc.edu/pubs/UIUCDCS-R-2003-2897.pdf) , TR @ UIUC (http://www.cs.uiuc.edu/research/techreports.php?report=UIUCDCS-R-2003-2897) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2003-cs322.bib.txt)

For how to define polymorphic type systems (with type inference) using rewriting, see:

A Rewriting Logic Approach to Type Inference
Chucky Ellison, Traian Florin Serbanuta and Grigore Rosu
WADT'08, LNCS 5486, pp 135-151. 2009
PDF (http://fsl.cs.uiuc.edu/pubs/ellison-serbanuta-rosu-2008-wadt-b.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/ellison-serbanuta-rosu-2008-wadt-slides.pdf) , LNCS (http://dx.doi.org/10.1007/978-3-642-03429-9_10) , WADT'08 (http://www.di.unipi.it/WADT2008/) , BIB (http://fsl.cs.uiuc.edu/pubs/ellison-serbanuta-rosu-2008-wadt-b.bib.txt)

A Rewriting Logic Approach to Type Inference
Chucky Ellison, Traian Florin Serbanuta and Grigore Rosu
Technical report UIUCDCS-R-2008-2934, March 2008
PDF (http://fsl.cs.uiuc.edu/pubs/ellison-serbanuta-rosu-2008-tr.pdf) , TR@UIUC (http://hdl.handle.net/2142/11423) , BIB (http://fsl.cs.uiuc.edu/pubs/ellison-serbanuta-rosu-2008-tr.bib.txt)

For abstract-interpretation style analyses, for the domain of units of measurement, see

Rule-Based Analysis of Dimensional Safety
Feng Chen and Grigore Rosu and Ram Prasad Venkatesan
RTA'03, LNCS 2706, pp197 - 207. 2003.
PDF (http://fsl.cs.uiuc.edu/pubs/chen-rosu-venkatesan-2003-rta.pdf) , LNCS (http://link.springer.de/link/service/series/0558/bibs/2706/27060197.htm) , RTA'03 (http://www.dsic.upv.es/~rdp03/rta/) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/rta/rta2003.html#ChenRV03) , BIB (http://fsl.cs.uiuc.edu/pubs/chen-rosu-venkatesan-2003-rta.bib.txt)

Certifying Measurement Unit Safety Policy
Grigore Rosu and Feng Chen
ASE'03, IEEE, pp. 304 - 309. 2003.
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-chen-2003-ase.pdf) , IEEE (http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240326) , ASE'03 (http://ase.cs.uni-essen.de/ase/past/ase2003/index.htm) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-chen-2003-ASE.bib.txt)

and for the domain of astronomical calculations/geometry see

Certifying Domain-Specific Policies

Michael Lowry and Thomas Pressburger and Grigore Rosu
ASE'01, IEEE, pp 81-90. 2001
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2001/lowry-pressburger-rosu-2001-ase/lowry-pressburger-rosu-2001-ase-public.pdf) , Slides(PPT) (http://fslweb.cs.illinois.edu/FSL/presentations/2001/2001-11-ASEb.ppt) , AutoFilter (https://ti.arc.nasa.gov/tech/rse/synthesis-projects-applications/autofilter/) , DOI (http://dx.doi.org/10.1109/ASE.2001.989793) , ASE'01 (http://ase-conferences.org/ase/past/ase2001/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2001/lowry-pressburger-rosu-2001-ase/lowry-pressburger-rosu-2001-ase-ref.bib)

For an approach to define abstract interpretations using membership equational logic and Maude see

Interpreting Abstract Interpretations in Membership Equational Logic
Bernd Fischer and Grigore Rosu
RULE'01, Electronic Notes in Theoretical Computer Science 59(4), pp 271-285. 2001
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2001/fischer-rosu-2001-rule/fischer-rosu-2001-rule-public.pdf) , Maude (http://maude.cs.illinois.edu) , DOI (http://dx.doi.org/10.1016/S1571-0661(04)00292-0) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2001/fischer-rosu-2001-rule/fischer-rosu-2001-rule-ref.bib)

For a generic framework for adding abstractions to C, see our work on plugable policies for C (https://www.ideals.illinois.edu/handle/2142/11421) .

An obvious work that needs to be done here is to write up and publish the definitions of type systems in the K tutorial (https://github.com/kframework/k/tree/master/k-distribution/tutorial) , especially the Dammas-Milner polymorphic type inferencer. There are some tricky technical mathematical details that need to be clarified there with regards to how meta-variables and sets of them are used. Then we need to develop a foundation for proving such semantic abstractions of a language correct; for example, type preservation and progress style properties. Automating such proofs would be very useful. Can we generalize the nice result by Stump and his students on proving type preservation using confluence (http://dx.doi.org/10.4230/LIPIcs.RTA.2011.345) to any abstractions? That is, take a concrete language semantics L and an abstract semantics of it A; that define correctness of A wrt L as some notion of confluence of the aggregation L+A. It cannot be ordinary confluence, because some terms are irrelevant and possibly non-confluent (e.g., terms corresponding to programs which are undefined or do not type), it will be what is called *relative confluence* in the literature, that is, confluence wrt only some terms. Finally, it would also be quite useful to develop a theory and technique to define abstractions compactly using a specialized notation, and then have the abstract language semantics be generated automatically from it.

34. **Binders with matching logic.**

Matching logic has a general notion of symbol, which captures both operational and predicate symbols as special cases. Symbols together with logical connectives and quantifiers can be used to build patterns, and the meaning of a pattern is that of the set of all the elements that match it. Equality can be defined as a pattern in matching logic, that is, it needs not be axiomatized like we do in FOL with equality. In fact, turns out that matching logic has the same expressiveness as FOL with equality, so in can be regarded as a variant of FOL that allows a more compact notation; indeed, translation to FOL with equality incurs an explosion in

the size of the formula, including adding new quantifiers, so the translation has more of a theoretical than practical relevance. The conventional meaning of operational and predicate symbols can be regained adding patterns, including equality patterns. See this paper to recall matching logic:

Matching Logic --- Extended Abstract
Grigore Rosu
RTA'15, Leibniz International Proceedings in Informatics (LIPIcs) 36, pp 5-21. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-29-RTA/2015-06-29-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.4230/LIPIcs.RTA.2015.5) , RTA'15 (http://rdp15.mimuw.edu.pl/index.php?site=rta) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-ref.bib)

When it gets to defining higher-order language or calculi, matching logic appears to suffer from the same problems as FOL: it provides no explicit support for binders in terms, for bound variable renaming, or for substitution.

On the other hand, we have developed a generalization of FOL that adds generic support for terms with binders, via the more general mechanism of a so-called *term syntax*:

Term-Generic Logic
Andrei Popescu and Grigore Rosu
Journal of Theoretical Computer Science, Volume 577(1), pp 1-24. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/popescu-rosu-2015-jtcs/popescu-rosu-2015-jtcs-public.pdf) , project (http://fsl.cs.illinois.edu/index.php/Generic_First-Order_Logic) , DOI (http://dx.doi.org/10.1016/j.tcs.2015.01.047) , Journal of Theoretical Computer Science (http://www.journals.elsevier.com/theoretical-computer-science/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/popescu-rosu-2015-jtcs/popescu-rosu-2015-jtcs-ref.bib)

The idea is that you start with a generic notion of *term*, which is axiomatized rather than constructed, and then build the entire FOL infrastructure on top of that. Everything works, including Gentzen proof systems and soundness and completeness results that smoothly generalize the classic FOL results.

The challenge here is to find an elegant and powerful mechanism to deal with binders in matching logic, and thus support not only first-order but also higher-order language definitions and calculi. Note that the K implementation of matching logic actually supports all these, but in a rather adhoc manner. We need a solid foundation for this. One obvious thing to do is to try to combine term-generic logic with matching logic. Another is to try to take advantage of the fact that in matching logic we can quantify over terms and we have equality as a pattern with the expected properties, and do things like this:

```
 syntax Exp ::= "prelambda" Var "." Exp  // to always be preceded by an existential quantifier "exists X"
              | "premu" Var . Exp
              | Exp Exp'
// syntactic sugar
 macro lambda X . E = (exists X . prelambda X . E)
 macro mu X . E = (exists X . premu X . E)

 rule (lambda X . E)  E' => E[E'/X]   // the usual matching logic substitution
 rule mu X . E => E[(mu X . E)/X]
```

The use of the substitution is probably unnecessary above. Can we replace it with equality and then rely on the properties of equality (recall that equality is definable in matching logic)?

```
rule (lambda X . E)  E' => E /\ (X = E')
rule mu X . E => E /\ (X = mu X . E)
```

Can we deal with all binders that elegantly? How about other binders, nominal logic, HOAS, etc.? Note that the nominal logic style encoding of `lambda X . E`, namely as `lambda([X] E)` where `[X] E` is a generic version of binder with no specific semantics except that it is a binder, does not seem to work in our context by just replacing `[X] E` with `exists X . E`, because of semantic reasons (thanks to Maribel Fernandez for pointing out this relationship!). Indeed, the matching logic semantics of the existential is the union of all `X` instances of the semantics of `E`; once we compute the union, then the parametricity in `X` is completely lost, so applying a `lambda` on it does not seem to work. On the other hand, with our definition above, the union is calculated over all the semantics of `prelambda X . E`. In other words, we put together all the particular "implementations" of a function. Indeed, patterns like `prelambda X . X` and `prelambda Y . Y` may have different interpretations in a particular model `M` under a particular interpretation `rho`, but the interpretation of `lambda X . X`, that is, of `exists X . prelambda X . X`, will contain both: `rho(prelambda X . X)` and `rho(prelambda Y . Y)` included in `rho(lambda X . X)`. So, in some sense, it is like in the relationship between an NFA and its associated equivalent DFA: a state of the DFA corresponds to the set of equivalent states of the NFA. Similarly, the interpretation of `lambda X . X` is the union of all the equivalent identity functions.

35. **Migrate Circ (circular coinduction) examples/theory to K/Circularity rule.**

Behavioral equivalence, in the sense of indistinguishability under experiments, and the automated circular coinduction technique to prove behavioral equivalence, were at the core of my PhD thesis:

Hidden Logic
Grigore Rosu
PhD Thesis, University of California at San Diego
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2000-phdthesis.pdf) , Thesis@UCSD (http://roger.ucsd.edu/record=b4170032~S9) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2000-phdthesis.bib.txt)

The Circularity rule of reachability logic, and how we automate its application in K, were to a large extent inspired from that early work in my PhD thesis and several subsequent papers discussed below. I believe it is important to unify or reconcile these two approaches to circular reasoning, both from a foundational and from a practical perspective. Such a unification could also let us better understand productivity and generalize it to the level of programming languages and their verification. Maybe novel and powerful automated verification techniques will be discovered in the process.

Besides my PhD thesis above, here are a few papers that might be useful to better understand circular coinduction and how we dealt with circularity and its automation before the Circularity rule in reachability logic and K:

Automating Coinduction with Case Analysis

Eugen-Ioan Goriac, Dorel Lucanu and Grigore Rosu
ICFEM'10, LNCS 6447, pp 220-236. 2010
PDF (http://fsl.cs.uiuc.edu/pubs/goriac-lucanu-rosu-2010-icfem.pdf) , LNCS (http://dx.doi.org/10.1007/978-3-642-16901-4_16) , ICFEM'10 (http://www.sei.ecnu.edu.cn/icfem2010/) , CIRC (http://fsl.cs.uiuc.edu/index.php/Circ) , BIB (http://fsl.cs.uiuc.edu/pubs/goriac-lucanu-rosu-2010-icfem.bib.txt)

Circular Coinduction with Special Contexts
Dorel Lucanu and Grigore Rosu
ICFEM'09, LNCS 5885, pp 639-659. 2009
PDF (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-2009-icfem.pdf) , Slides(PDF) (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-2009-icfem-slides.pdf) , LNCS (http://dx.doi.org/10.1007/978-3-642-10373-5_33) , ICFEM'09 (http://icfem09.inf.puc-rio.br/ICFEM.html) , BIB (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-2009-icfem.bib.txt)

Circular Coinduction: A Proof Theoretical Foundation
Grigore Rosu and Dorel Lucanu
CALCO'09, LNCS 5728, pp 127-144. 2009
Slides (PDF) (http://fsl.cs.uiuc.edu/pubs/rosu-lucanu-2009-calco-slides.pdf) , LNCS (http://dx.doi.org/10.1007/978-3-642-03741-2_10) , CALCO'09 (http://calco09.dimi.uniud.it/) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/calco/calco2009.html#RosuL09) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-lucanu-2009-calco.bib.txt)

CIRC: A Circular Coinductive Prover
Dorel Lucanu and Grigore Rosu
CALCO'07, LNCS 4624, pp 372-378. 2007
PDF (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-2007-calco.pdf) , CIRC webpage (http://fsl.cs.uiuc.edu/CIRC) , CALCO'07 (http://www.ii.uib.no/calco07/) , BIB (http://fsl.cs.uiuc.edu/pubs/lucanu-rosu-2007-calco.bib.txt)

Migrating all the examples in the old Circ prover to K is a good indication that, at least at the practical/automation level, the current approach captures the old one.

Here are two papers that discuss complexity results for behavioral equivalence, and also touch the topic of productivity:

Equality of Streams is a Pi_2^0-Complete Problem
Grigore Rosu
ICFP'06, ACM, 2006
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2006-icfp.pdf) , ICFP'06 Slides (http://fsl.cs.uiuc.edu/pubs/rosu-2006-icfp.ppt) , ACM (http://doi.acm.org/10.1145/1159803.1159827) , ICFP'06 (http://icfp06.cs.uchicago.edu/) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2006-icfp.bib.txt)

Similar results should be generalizable to the level of matching/reachability logic and, as I said above, doing so may reveal new results and definitely a deeper understanding. Here is a paper where we generalized the notion of productivity from streams to arbitrary behavioral theories:

So in case we find a smooth generalization of such general behavioral rewrite systems, then we can also handle productivity as we know it for streams.

Finally, here are some papers taking a different general view at behavioral equivalence:

Behavioral Extensions of Institutions
Andrei Popescu and Grigore Rosu
CALCO'05, LNCS 3629, pp. 331-347. 2005
PDF (http://fsl.cs.uiuc.edu/pubs/popescu-rosu-2005-calco.pdf) , CALCO'05 Slides (http://fsl.cs.uiuc.edu/pubs/popescu-rosu-2005-calco.ppt) , LNCS (http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/11548133_21) , CALCO '05 (http://www.cs.swan.ac.uk/calco/index.php) , DBLP (http://www.informatik.uni-trier.de/~ley/db/conf/calco/calco2005.html#PopescuR05) , BIB (http://fsl.cs.uiuc.edu/pubs/popescu-rosu-2005-calco.bib.txt)

Behavioral Abstraction is Hiding Information
Grigore Rosu
J. of TCS, Volume 327(1-2), pp 197-221. 2004
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2004-jtcs.pdf) , J.TCS (http://dx.doi.org/10.1016/j.tcs.2004.07.027) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2004-jtcs.bib.txt)

Inductive Behavioral Proofs by Unhiding
Grigore Rosu
CMCS'03, ENTCS 82(1). 2003
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2003-cmcs.pdf) , ENTCS (http://dx.doi.org/10.1016/S1571-0661(04)80645-5) , CMCS'03 (http://www.mathematik.uni-marburg.de/~cmcs/) , Experiments (http://fsl.cs.uiuc.edu/pubs/rosu-2003-cmcs-experiments.zip) , DBLP (http://www.informatik.uni-trier.de/~ley/db/journals/entcs/entcs82.html#Rosu03) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2003-cmcs.bib.txt)

Understanding them could be useful for cracking this problem.

36. **Define/Implement K in K.**

A framework which is Turing complete (that is, it can express any Turing machine, or any computable domain) and reflective (that is, it can express itself in itself) can be regarded as an alternative foundation or model of computation, both from a theoretical and from a practical perspective. The Turing completeness is sufficient for the theoretical part, because we can always represent the universal Turing machine in it and by that anything that can be done using conventional computability theory can also be done with the framework. But that may not be practical. We want the framework to be reflective, too, and if possible, elegantly reflective. See the Maude documentation on its reflection (http://maude.lcc.uma.es/manual/maude-manualch11.html#x66-14300011) for a good understanding of what we are after. We would like not only to do the same for K, but we plan on implementing K that way, in fact. Specifically, we want to define a minimal Turing complete and reflective version of K, which we are currently referring to as KORE (https://github.com/kframework/k/wiki/KAST-and-KORE) . KORE will therefore have all the power that we need. But it will miss convenient features, such as defining multiple productions separated with `|`, or parametric modules, etc. Then we can define in KORE all these convenient features that together form what we call K, as an extension of the definition of KORE in KORE.

37. **Maximal causality as a configuration space reduction for semantics-based verification.**

We have shown that we can go way beyond the usual Lamport happens-before in what regards causal dependence between events in a concurrent program execution trace:

Maximal Causal Models for Sequentially Consistent Systems
Traian Florin Serbanuta and Feng Chen and Grigore Rosu
RV'12, LNCS 7687, pp 136-150. 2012
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2012/serbanuta-chen-rosu-2012-rv/serbanuta-chen-rosu-2012-rv-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2012/serbanuta-chen-rosu-2012-rv-slides.pdf) , JPredictor (http://fsl.cs.illinois.edu/index.php/JPredictor) , DOI (http://dx.doi.org/10.1007/978-3-642-35632-2_16) , RV'12 (http://rv2012.ku.edu.tr) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2012/serbanuta-chen-rosu-2012-rv/serbanuta-chen-rosu-2012-rv-ref.bib)

Maximal Sound Predictive Race Detection with Control Flow Abstraction
Jeff Huang and Patrick Meredith and Grigore Rosu
PLDI'14, ACM, pp 337-348. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/huang-meredith-rosu-2014-pldi/huang-meredith-rosu-2014-pldi-public.pdf) , Slides(PPT) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-06-10-PLDI.ppt) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-06-10-PLDI.pdf) , jPredictor (http://fsl.cs.illinois.edu/jPredictor) , DOI (http://dx.doi.org/10.1145/2594291.2594315) , PLDI'14 (http://conferences.inf.ed.ac.uk/pldi2014/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/huang-meredith-rosu-2014-pldi/huang-meredith-rosu-2014-pldi-ref.bib)

Lamport's happens-before is a trivial partial order generated by ordering all the accesses to each shared resource. For example, if in a particular program execution a thread/process reads x before a another thread/process writes x, then we add a less-then relation between the read and the write. Similarly for all concurrent objects (locks, semaphores, monitors, etc.), carefully taking into account their semantics or access protocol. Together with the total execution order on each thread, this gives us a partial order on all the events in the execution trace, called the *causal dependence*, with the intuition that each smaller event has caused a larger one in the observed execution; but two un-ordered events are independent, meaning that they can be permuted and we still get a feasible execution of the same program. In other words, any linearization of Laport's causal dependence partial order is a feasible execution of the original program. Now if any of these linearizations violates a desired property of the program, then you detected a bug in the original program, even if the bug was not directly visible in the original, flat execution trace. This natural idea was the basis of what we called *predictive runtime analysis* (PRA) back in 2001-2002, although note that the "predictions" are always correct; if implemented properly, there should be no false alarms reported. PRA has been mostly used to predict data-races, but there is no conceptual restriction to limit it to data-races.

Some researchers think that Lamport's happens-before is what causal dependence is all about; they even *define* causal dependence that way. Others, including ourselves, have tried hard to extend the notion to include more linearizations. Why? Because this way you can have a better prediction capability in tools, and thus have better bug detection performance still without false positives. There is a series of papers where generalizations of Lamport's happens-before were proposed. For example, you can only relate a write with all the reads following it, and thus move them all together before or after other similar semantic blocks; this is obviously not a partial order anymore, it is a disjunction of partial orders. Similarly, you can permute synchronized blocks provided that there are no data dependencies between them. And you can go even

further if you are not interested in the exact values that are read in variables, but only that a read event has taken place. All these are explained and all the literature discussed in depth in the two papers above. Our major contribution to this area, presented in the papers above, is to show that there is a *maximal* causal model. In other words:

> From one execution trace you can extract a *maximal causal model*, with the property that it comprises exactly all the executions that could be generated by any program that could generate the original trace!

Please read the previous sentence again. Once more, because it is trickier than it may seem. This tells us that any sound dynamic analysis or runtime verification tool, no matter how sophisticated it is, cannot detect more errors from the same execution trace than a tool based on our maximal causal model. The commercial tool RV-Predict (https://runtimeverification.com/predict/) implements the maximal causal model technique, so unless it has implementation errors, it should be the best dynamic race-detection tool that detects no false positives.

We believe that the maximal causal model idea can be significantly generalized, to go well beyond the simplistic model based on variable reads, variable writes and concurrency objects. Specifically, we believe it can be generalized to work with any programming language semantics. Indeed, the K semantic rules have a read-only part, a write-only part, and a don't care part. We should be able to axiomatize the basic principles of semantic execution based on positions in the configuration where each rule reads/writes, the same way we axiomatized the sequentially consistent program execution in our papers above. Then, based on that, to obtain a general notion of maximal causality that applies to any programming language, based entirely on its semantics. To confirm that it is the right notion, when applied to the toy language considered in our papers above it should give the same maximal causality, and when applied to the Java semantics it should give the same model as the one in RV-Predict, so it should detect the same maximal number of data-races.

The above will have at least two major benefits. First, it will yield RV-Predict-like tools for lots of languages, not only for Java. Second, and more importantly, it will yield a very aggressive partial-order reduction mechanism for program verification. Indeed, we can use it in our K-based deductive verification or model checking tools to run the program symbolically in a managed environment, to divide-and-conquer the problem quite elegantly: (1) develop efficient techniques to analyze each causal model generated by each execution; and (2) make sure that at each execution a (maximally) causally different model is observed, i.e., the generated execution does not belong to the maximal causal model generated by any of the previously observed executions. This idea has been discussed in our FSL group for several years by now. Qingzhou Luo has made some interesting and in my view convincing experiments in his PhD thesis showing that this can bring significant benefits. A former member of our group, Jeff Huang, has published similar results in PLDI'15. Make sure you check Qingzhou's and Jeff's works out, too, and even contact them to ask what the current status of their work is. Take also a look at the related play and replay challenge.

38. **Playing and replaying program executions.**

When finding errors in a system using program analysis or verification or even testing tools, you often want to be able to play or replay the erroneous execution so you can better understand the bug. For example, imagine that you are analyzing a program using the maximal causality technique, and that you found that a certain permutation of the events in the observed trace violates the property you are checking. You also know

from theoretical results that the permutation is feasible. There could be, however, millions of events in the predicted execution trace. How can you help the user of your tool pinpoint and understand the bug? The best way to do it in my view would be to steer the program execution to follow the buggy path until the bug is hit. Being able to go back and forth along that execution and see all the state details in some debugger would allow the user to understand the problem well and then fix the bug.

A simplified version of the problem above, which is not trivial either, is just the replay itself. Imagine you execute a Java program and you hit a bug, say a data-race. Then you want to replay the execution to see how the race really happened. Unfortunately, that is a very hard problem with little to no tool support. The program for example can read input from the user, or from sockets, or from files, and then at reply time you want to make sure that the same input is fed to the program, so that it follows that same path. You would need to modify the program (instrument it?) so that when you execute it the first time you log relevant information that allows you to replay afterwards, and then at replay time reuse that information. We started working on this problem, but there is still work to do and nothing was published yet: https://github.com/kheradmand/replaymop

39. **Runtime verification of matching logic patterns.**

One of the most difficult parts of program verification is to come up with the right properties to prove. Not only the global, top-level properties for the code that you want you want to verify, which are unavoidable, but also helping properties (loops invariants, assertions, pre/post conditions, etc.). Some of these may be learned or inferred using techniques like the ones discussed in the invariant inference challenge, but in general you cannot expect to automatically learn or infer all of them. Most of the time, it takes a few iterations to get these right. Whether learned automatically or provided manually, there is a need to quickly check whether they are correct or not. Ideally, you would like to check them by actually verifying the entire program, but that is going to be quite expensive in general. What is significantly cheaper is to execute the program and check them at runtime, or to runtime verify them. However, even that can be quite expensive in some cases. For example, imagine that you have a loop that iterates 1,000,000 times and at each iteration it allocates a new node in a tree represented somehow in the heap, and that the pattern that you want to check is that the tree structure in the heap is maintained. Specifying trees or other heap structures is not a problem at all, as we know from the matching logic paper:

Matching Logic --- Extended Abstract
Grigore Rosu
RTA'15, Leibniz International Proceedings in Informatics (LIPIcs) 36, pp 5-21. 2015
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-06-29-RTA/2015-06-29-RTA.pptx) , Matching Logic (http://matching-logic.org/) , DOI (http://dx.doi.org/10.4230/LIPIcs.RTA.2015.5) , RTA'15 (http://rdp15.mimuw.edu.pl/index.php?site=rta) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/rosu-2015-rta/rosu-2015-rta-ref.bib)

But note that the tree definition has a recursive nature, and naively checking, or matching it against a flat heap each time means re-doing all the work done at the previous iteration of the loop, plus adding the new node to the tree as well. We would like to "cache" all the matching work that we've done at the previous iterations, and only do the delta that is needed to add the effects of the new loop iteration to the already existing work. In some sense, is like dynamic programming used as a way to implement efficiently recursive

algorithms by caching the previously seen results. Edgar Pek has a chapter in his thesis on this topic, but in the context of separation logic. Since matching logic captures separation logic, we hope that we should be able to devise a general efficient runtime monitoring algorithm for matching logic patterns in the same style but working at all levels in the program configuration, not only in the heap.

40. **Certifiable runtime verification**

Semantics-based verification opens the door for novel verification methodologies and for verification of non-conventional programming languages and programming paradigms, because all is needed is an operational semantics of the target language or paradigm, which is considerably simpler to design and define than a verification infrastructure for that language or paradigm. We have already discussed the possibility of verifying AspectJ programs in the semantics of real languages challenge. But we can go further than that. How about Monitoring-Oriented Programming, which can be regarded as one level above aspect-oriented programming? For example, suppose that you have a Java program that includes a class `Resource` with two methods, `use()` and `authenticate()`, and that you want to runtime verify/monitor the property:

> for any resource `r`, method `r.authenticate()` must always be called before `r.use()`; enforce safety/security by calling `r.authenticate()` whenever the property is violated.

This property can be easily formalized and enforced using JavaMOP. What happens under the hood is that JavaMOP generates AspectJ code, which maintains a monitor for each resource for the specified property, and updates it whenever the relevant events take place at the specified pointcuts; if the monitor finds that the property is violated, then it executes the recovery code to call the authentication code, and thus the intended property that authentication precedes usage of the resource is never really violated. Can we do all the above automatically? That is, can we automatically verify that a runtime verified system is correct with respect to the property that runtime verifies? Folklore in the formal methods community goes that if you cannot verify a property then monitor it. But if you really try to do that and prove that the monitored program is correct, you realize that the problem is not as easy as one may think. In fact, it is quite complex. Worse, there are no tools to help you do it in the real world for real programming languages like Java, because complex monitoring of such languages subsumes aspect-oriented programming, and there is no real technology available off-the-shelf to verify aspect-oriented programs. Formally guaranteeing correctness with respect to the runtime verified properties is in my view one of the most important challenges that the runtime verification community needs to overcome. Let's call this problem *certifiable runtime verification*.

41. **Formalizing the JDK API**

One of the biggest challenges of the verification communities, both static and runtime, is the lack of properties to verify. There have been studies on this topic, and many engineers found that writing the properties to check/verify/monitor is the most difficult part for them and often a show stopper. One possibility, and probably the most precise, it to just formalize the properties by hand. Another could be to mine the properties from observing many execution traces. One way or another, we would like to have formal property specifications for commonly user libraries. Those can be used to verify the libraries, or to check programs using the libraries to make sure they are using the libraries correctly, that is, they do not violate the usage protocols of those libraries. For example, in Java, if you create an iterator or an enumerator

over a collection of elements, then you cannot modify the collection by adding or removing elements to it as you are iterating or enumerating it. If you do that, unexpected results can happen and one may even consider that your application is broken.

We have systematically analyzed and categorized the documentation text of four packages of the JDK API, and found that all the properties that were implicit in the text could be formalized as what we call *parametric properties*:

Towards Categorizing and Formalizing the JDK API
Choonghwan Lee and Dongyun Jin and Patrick O'Neil Meredith and Grigore Rosu
Technical Report http://hdl.handle.net/2142/30006, March 2012
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2012/lee-jin-meredith-rosu-2012-tr/lee-jin-meredith-rosu-2012-tr-public.pdf) , Java API (https://github.com/runtimeverification/property-db) , DOI (http://hdl.handle.net/2142/30006) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2012/lee-jin-meredith-rosu-2012-tr/lee-jin-meredith-rosu-2012-tr-ref.bib)

RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties
Qingzhou Luo and Yi Zhang and Choonghwan Lee and Dongyun Jin and Patrick O'Neil Meredith and Traian Florin Serbanuta and Grigore Rosu
RV'14, LNCS 8734, pp 285-300. 2014
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2014/luo-zhang-lee-jin-meredith-serbanuta-rosu-2014-rv/luo-zhang-lee-jin-meredith-serbanuta-rosu-2014-rv-public.pdf) , Slides(PPTX) (http://fslweb.cs.illinois.edu/FSL/presentations/2014/2014-09-25-RV.pptx) , JavaMOP (https://github.com/runtimeverification) , DOI (http://dx.doi.org/10.1007/978-3-319-11164-3_24) , RV'14 (http://rv2014.imag.fr/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2014/luo-zhang-lee-jin-meredith-serbanuta-rosu-2014-rv/luo-zhang-lee-jin-meredith-serbanuta-rosu-2014-rv-ref.bib)

JavaMOP, for example, takes parametric properties as input and automatically generates instrumentation and inlines efficient monitors for them in your application to monitor. Other verification tools can also make use of such properties. You can visualize these properties using a nice graphical interface that lays them out over the JavaDoc of the JDK API: https://runtimeverification.com/monitor/annotated-java-8/ If you want to see the code that is generating that, then check this github repository out: https://github.com/runtimeverification/property-db. I think formalizing the entire JDK API is probably one of the most useful projects that one can carry out. Indeed, Java is one of the most used programming languages, and almost any Java program uses the JDK API. Formalizing the JDK API would give the community a rigorous means to check the correct usage of the JDK in their programs, as well as proof obligations for the libraries themselves (that is, prove that the library implementation satisfies the specifications).

Ideally, we would like the entire community of Java users and formal methods researchers to contribute to such an important project. We have created the github repository at https://github.com/runtimeverification/property-db exactly for that purpose. Maybe we need to popularize it more? Or to improve the graphical interface that layers the categorization and the current categorization over the JDK Javadoc at https://runtimeverification.com/monitor/annotated-java-8/?

42. **Parametric property mining using positive and negative examples.**

Related to the problem of formalizing the JDK API, it would be great to make use of property learning or mining algorithms. That is, to instrument programs making use of the libraries whose properties we want to learn, and then collect many traces and try to learn from them what properties that API ought to have. Ideally, we would like to do all this automatically. In the paper

Mining Parametric Specifications
Choonghwan Lee, Feng Chen and Grigore Rosu
ICSE'11, ACM, pp 591-600. 2011
PDF (http://fsl.cs.uiuc.edu/pubs/lee-chen-rosu-2011-icse.pdf) , Slides(pptx) (http://fsl.cs.uiuc.edu/pubs/2011-05-27-ICSE.pptx) , ACM (http://dx.doi.org/10.1145/1985793.1985874) , ICSE'11 (http://2011.icse-conferences.org/) , jMiner (http://fsl.cs.uiuc.edu/jMiner) , BIB (http://fsl.cs.uiuc.edu/pubs/lee-chen-rosu-2011-icse.bib.txt)

we proposed a technique that does precisely that. We used the unit tests of the target API packages to see which groups of methods are expected to obey some property, the assumption being that if a tester wrote a test to check an interaction among a certain group of methods, most likely those methods are expected to obey some interaction protocol. Then we instrumented large applications making use of those methods to log all their calls to those methods at runtime, and this way we collected millions of execution traces. We then sliced those execution traces according to various parameter instances, and thus obtained even more traces of non-parametric events. Then using a statistical learning algorithm (k-means) we learned the regular expressions or DFAs that best explained the observed traces. In the end we learned more than 100 parametric properties of the JDK API.

All the above sounds great and it may even work for some cases, but our experience was actually rather negative from a practical perspective. In the end, we ended up throwing away almost all the properties that were mined and we rewrote them by hand, as discussed in the papers discussed in the formalizing the JDK API challenge. Ironically, we had a very hard time to publish the paper about the manually written properties, which took a lot of time to get right, while the paper on automatic mining was easily accepted for publication; well, we all know how random the publication process is these days. However, I actually have some hopes that mining can in fact work in practice, but in order for that to happen we need to learn not only from positive data as we've done above (that is, from correct traces wrt the property), but also from negative data. Search on the internet for "algorithms for learning regular expressions from positive and negative data" and you will find a vast body of literature. I believe that some of those could work if we can produce good negative traces for them. A negative trace would be a trace which should not be accepted by the learned property. How can we generate such negative traces? That would be the challenge here. One can use test-case generation techniques to generate object instances, and then call the methods whose protocol you want to formalize randomly on those objects and mark as negative all the traces where uncaught exceptions are thrown (the hypothesis here being that those sequences of calls should not be part of the property). Or take some positive traces obtained in some large programs, and serialize the object instances on which they were called, save them in a file and then log all the calls to the target methods on them; then load the serialized objects and only call the logged methods on them, in the logged order. This way, you obtain minimal programs that generate positive traces, programs which consist of loading the object parameters followed by sequences of method calls. Well, take these programs and re-order the method calls, then run the resulting programs and if they throw exceptions then mark the way they reordered the methods as negative traces. Maybe other ideas will come as you play with this. Anyway, please use our insight and experience, and stop wasting your time learning parametric properties from only positive traces. If you are interested in mining, spend your time wisely and come up with smart techniques to generate negative traces. Then use the best algorithms that you can find to learn from both positive and negative traces. I believe something useful will

come out of this, which can then be used in some automated manner to mine the properties of the entire JDK API: use unit tests as we did to find groups of methods that are expected to interact and thus obey interaction protocols, then use large applications to collect positive traces, then use some techniques to also collect negative traces, and then learn the regular expressions or DFAs. You can also do this in a loop, until you converge and cannot improve the property you learn.

43. **Offline analysis of large logs.**

Many companies and institutions these days collect huge logs of data for their records, auditing, and in general any kind of analysis. It is not unusual for a web service company to log every single action that any customer takes on their server. There is therefore a need for efficient algorithms and techniques to analyze large logs of data. By large in this context we mean logs of hundreds of GBs or even TBs or PBs of data. Conventional monitoring algorithms can be used to analyze such logs, by simply reading the log entries in order and processing them as if they were generated online. But the advantage of log analysis is that you have random access to all the entries in the log, in any order. Interestingly, there are trace analysis algorithms that are significantly more efficient than online algorithms when they have random access to the entries in the trace. See, for example, an extreme case in these two papers, where a future-time LTL checking algorithm is linear both in the trace and in the formula when it can process the trace backwards:

Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae
Grigore Rosu and Klaus Havelund
Technical Report https://ti.arc.nasa.gov/m/pub-archive/archive/0220.pdf, January 2001
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2001/rosu-havelund-2001-tr/rosu-havelund-2001-tr-public.pdf) , DOI (https://ti.arc.nasa.gov/m/pub-archive/archive/0220.pdf) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2001/rosu-havelund-2001-tr/rosu-havelund-2001-tr-ref.bib)

Rewriting-Based Techniques for Runtime Verification
Grigore Rosu and Klaus Havelund
J.ASE, Volume 12(2), pp 151-197. 2005
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2005/rosu-havelund-2005-jase/rosu-havelund-2005-jase-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1007/s10515-005-6205-y) , J.ASE (http://link.springer.com/journal/10515) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2005/rosu-havelund-2005-jase/rosu-havelund-2005-jase-ref.bib)

It is known that any forwards future-time LTL algorithm requires exponential time and space in the formula, so the fact that the entire trace is available upfront can make a big difference. It is also know that a dynamic programming algorithm similar to the one used above for future-time LTL can be applied to past-time LTL, too, in which case the trace analysis proceeds in a forwards manner:

Efficient Monitoring of Safety Properties
Klaus Havelund and Grigore Rosu
J. of STTT, Volume 6(2), pp 158-173. 2004
PDF (http://fsl.cs.uiuc.edu/pubs/havelun-rosu-2004-sttt.pdf) , J.STTT (http://dx.doi.org/10.1007/s10009-003-0117-6) , BIB (http://fsl.cs.uiuc.edu/pubs/havelund-rosu-2004-sttt.bib.txt)

Synthesizing Monitors for Safety Properties
Klaus Havelund and Grigore Rosu
TACAS'02, LNCS 2280, pp 342-356. 2002
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2002/havelund-rosu-2002-tacas/havelund-rosu-2002-tacas-public.pdf) , Slides(PPT) (http://fslweb.cs.illinois.edu/FSL/presentations/2002/2002-04-TACAS.ppt) , MOP (https://github.com/runtimeverification) , DOI (http://dx.doi.org/10.1007/3-540-46002-0_24) , TACAS'02 (http://www.etaps.org/2002/Tacas/tacas.html) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2002/havelund-rosu-2002-tacas/havelund-rosu-2002-tacas-ref.bib)

And that algorithm can be extended to work with nested call-return events, to monitor stack-related properties without touching the stack:

Synthesizing Monitors for Safety Properties -- This Time With Calls and Returns --
Grigore Rosu, Feng Chen and Thomas Ball
RV'08, LNCS 5289, pp 51-68, 2008
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-chen-ball-2008-rv.pdf) , RV'08 (http://rv08.in.tum.de/) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-chen-ball-2008-rv.bib.txt)

A natural question here is how far we can push these dynamic-programming-style algorithms for off-line log analysis? For example, how rich can the logical formalism to express properties be so that we can have linear or close to linear complexity algorithms for off-line checking? It looks like we should at least be able to support the entire LTL, both past-time and future-time. Can we also add support to call-return events to the logic? How about time, can we add it to the logic as well (see the RV of times properties challenge, too)? Another interesting category of properties is regular expressions, possibly extended with complement (to also allow you to say that certain patterns should not match); see also the ERE membership checking challenge.

I would also include here the development of appropriate finite-trace logics and complete deduction for them. The logs are large but finite, so such logics are appropriate for log analysis. Complete deduction for such logics helps us generate optimal monitors, by statically deriving the formula in all possible ways depending on the next event, and then proving when the obtained formulae are equivalent to previous ones. One of the nicest monitor generation technique that I've worked on does the above in a coinductive style; see the monitor generation using coinduction challenge. Also, for really large logs, it may even be the case that an algorithm that needs to read all the entries in the log may already be too slow, even if it processes each entry in constant, even zero time. Note that the log may not even be stored in one place, it may be distributed across many different machines and even geographical sites. We would like algorithms that split the trace in chunks and then analyze each chunk separately and in parallel with the other chunks, if possible.

44. **Extended Regular Expression (ERE) Membership Checking.**

Checking whether a word belongs to the language of an extended regular expression (ERE) is a problem that challenged me several times, and I still don't have a good solution. The problem starts with the fact that once you extend the language of regular expressions with complement, you have a potential non-elementary explosion in the size of the DFA or NFA monitor. Indeed, to complement an NFA you have to first determinize it, which comes at an exponential cost. Now if you have nested complement operations, then

you have multiple exponential explosions, in the worst case as many as proportional with the size of the ERE. This can simply make the generation of the DFA or NFA monitor impossible, if we follow the usual approach.

To approach the problem from a different angle, we first tried an approach where you do not generate any automata statically. Instead, generate only the path that you are on, dynamically, as you receive events from the monitored system. We do that using derivatives and rewriting, together with simplification rules to keep the size of the ERE bounded:

Testing Extended Regular Language Membership Incrementally by Rewriting
Grigore Rosu and Mahesh Viswanathan
RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2003/rosu-viswanathan-2003-rta/rosu-viswanathan-2003-rta-public.pdf) , Slides(PPT) (http://fslweb.cs.illinois.edu/FSL/presentations/2003/2003-06-RTA.ppt) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1007/3-540-44881-0_35) , RTA'03 (http://users.dsic.upv.es/~rdp03/rta/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2003/rosu-viswanathan-2003-rta/rosu-viswanathan-2003-rta-ref.bib)

Not only that we proved that the ERE size is bounded, but we also proved that its overall size is only exponential, so the non-elementary explosion can be avoided this way. Unfortunately, our proof, which is quite complex, had an error; the error was found by Prasanna Thati, a PhD student at UIUC back in 2004. Frustratingly, that sent us back into a non-elementary complexity of online ERE membership checking. I actually proved in the meanwhile that the complexity is indeed non-elementary, that is, if you want to check online a trace of size `n` against the language of an ERE of size `m`, then you cannot do better that `Omega(n * 2^(2^(2^...)))`, where the exponential tower is as deep as the number of nested complement operations in the ERE; let me know if you want to see the proof, I have it in a book draft which I can send you if interested.

But we nevertheless liked our rewrite-based monitoring algorithm. The problem with it, though, is that you have to pay the price of rewriting the ERE at each event. Since the formula can potentially grow non-elementarily large, that means that the runtime overhead may be potentially quite high in some cases. One thing which can be done is to mimic the rewrite-based monitoring procedure statically and cache all the relevant EREs that it goes through as states, and this way generate a monitor as an automaton whose states correspond to EREs (but relabeled as numbers, for efficiency):

Generating Optimal Monitors for Extended Regular Expressions
Koushik Sen and Grigore Rosu
RV'03, Electronic Notes in Theoretical Computer Science 89(2), pp 226-245. 2003
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-2003-rv/sen-rosu-2003-rv-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1016/S1571-0661(04)81051-X) , RV'03 (https://rtg.cis.upenn.edu/rv2003/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-2003-rv/sen-rosu-2003-rv-ref.bib)

While we know we cannot avoid the non-elementary explosion in the worst case, at least we hope that this way we avoid intermediate explosions that you might have with the conventional approach which do not result in an explosion in the final monitor automaton. For example, consider an ERE of the form `~(~(~(~(~(~(~(~r))))))`. This is equivalent to `r` and our monitor generation technique above will completely avoid the unnecessary exponential explosion that the naive algorithm suffers from. It would be nice to implement the

above very efficiently and then do some experiments to compare its efficiency with the conventional approach. Maybe even turn it into a library that languages using regexes can incorporate. See also the monitor generation using coinduction challenge.

But what if the entire trace is available, like in the off-line analysis of large logs challenge? Can we come up with better algorithms for ERE membership checking if we have random access to the trace? We managed to improve a bit the best known lower bounds:

An Effective Algorithm for the Membership Problem for Extended Regular Expressions
Grigore Rosu
FOSSACS'07, LNCS 4423, pp 332-345, 2007
PDF (http://fsl.cs.uiuc.edu/pubs/rosu-2007-fossacs.pdf) , FOSSACS'07 (http://www2.in.tum.de/~seidl/fossacs07/) , BIB (http://fsl.cs.uiuc.edu/pubs/rosu-2007-fossacs.bib.txt)

We showed that if we have a trace of size `n` and an ERE of size `m`, then we can check the membership of the trace to the language of the ERE in time `O(n^2 * (log n + m) * 2^m)`, while the previous best algorithms had complexity `Omega(n^3)`. So only exponential in `m`, which means that this algorithm may work in situations where the online algorithm cannot generate or store the monitor for the ERE. However, it is still annoying that you have to pay such a high `n^2 * log n` cost in the size of the execution trace. Can we push that down to linear complexity or `O(n * log n)` in the execution trace? If not, then prove so.

45. **Optimal Monitor generation using coinduction.**

There is a nice relationship between monitor generation and circular coinduction. We used it for generating optimal monitors for extended regular expressions and for linear temporal logic:

Generating Optimal Monitors for Extended Regular Expressions
Koushik Sen and Grigore Rosu
RV'03, Electronic Notes in Theoretical Computer Science 89(2), pp 226-245. 2003
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-2003-rv/sen-rosu-2003-rv-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1016/S1571-0661(04)81051-X) , RV'03 (https://rtg.cis.upenn.edu/rv2003/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-2003-rv/sen-rosu-2003-rv-ref.bib)

Generating Optimal Linear Temporal Logic Monitors by Coinduction
Koushik Sen and Grigore Rosu and Gul Agha
ASIAN'03, Lecture Notes in Computer Science (LNCS) 2896, pp 260-275. 2003
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-agha-2003-asian/sen-rosu-agha-2003-asian-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1007/978-3-540-40965-6_17) , ASIAN'03 (http://dx.doi.org/10.1007/b94667) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2003/sen-rosu-agha-2003-asian/sen-rosu-agha-2003-asian-ref.bib)

The idea is to start with the formula/pattern/specification for which you want to generate an optimal monitor, and then derive it in all possible ways, one step, depending on what the next even can be. That is, considering a formula `PHI` and an event `e`, the derived formula `PHI{e}` is a formula `PHI'` with the property that `PHI'` holds on a given trace `t` if and only if `PHI` holds on the trace `e t`. Each time a new formula is derived, check if it is equivalent to some formula that has already been seen. If yes, then discard

the new formula. If not, then add the new formula to the pool and keep deriving. Now the interesting part is that in order to check for equivalence by circular coinduction, you also derive the two involved formulae. As you derive them, it makes sense to cache all the formulae that you reach in the process, because they will likely be needed again. The challenge is how to do this efficiently. Our previous solutions only showed that the concept makes sense, but we have not really put much effort into making that process efficient. Ideally, we would like to combine the derivation process that unveils the optimal monitor with the circular coinductive checking, and apply them both synchronously making sure that no work is repeated. Another challenge is to understand how general this monitor generation process can be. For example, what "API" should a given logical formalism provide in order to apply this coinductive technique to it? It should obviously provide a derivative operation that takes a formula and an event and returns another formula. It would likely also provide some formula simplification procedure. Anything else needed?

46. **Runtime verification of timed properties.**

Runtime verification and monitoring of times properties raises many interesting and challenging problems. For example, who keeps track of the time? Does the monitor have its own clock, or do the events come already time stamped? Does the monitor generate timeouts, or the event sequence somehow magically is assumed to provide a special timeout event? When are the timeouts generated and by whom? Now supposing that somehow the monitor is made aware of the time, what would the best formalism for specifying timed properties be? We know that timed automata have some complexity challenges, and we often do not need their full generality. Besides, in many cases users prefer a more declarative formalism, such as variants of temporal logic. We have investigated the problem of monitoring metric temporal logic properties, where the events were assumed to arrive time-stamped:

Monitoring Algorithms for Metric Temporal Logic
Prasanna Thati and Grigore Rosu
RV'05, Electronic Notes in Theoretical Computer Science 113, pp 145-162. 2005
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2005/thati-rosu-2005-rv/thati-rosu-2005-rv-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1016/j.entcs.2004.01.029) , RV'05 (https://www.react.uni-saarland.de/rv2005/) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2005/thati-rosu-2005-rv/thati-rosu-2005-rv-ref.bib)

We employed a rewrite-based approach, following the approach in

Rewriting-Based Techniques for Runtime Verification
Grigore Rosu and Klaus Havelund
J.ASE, Volume 12(2), pp 151-197. 2005
PDF (http://fslweb.cs.illinois.edu/FSL/papers/2005/rosu-havelund-2005-jase/rosu-havelund-2005-jase-public.pdf) , MOP (http://fsl.cs.illinois.edu/mop) , DOI (http://dx.doi.org/10.1007/s10515-005-6205-y) , J.ASE (http://link.springer.com/journal/10515) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2005/rosu-havelund-2005-jase/rosu-havelund-2005-jase-ref.bib)

but it would be nice to generate some sort of timed automata from such mtl properties, maybe using the coinductive approach.

47. **Evolution-aware program analysis.**

    Program analysis tends to be very expensive, no matter whether we are talking about static of dynamic analysis. On the other hand, software to be analyzed evolves incrementally, in small steps. Ideally, we would like to be able to reuse analysis efforts from one version of the software to the next. A trivial example is when you verify a function and that function does not change from one version to the next; then obviously we do not need to verify that function again. But how about another function which changes that calls the function we verified? We may or may not need to verify it fully, depending on what changed and on the property.

    The challenge here is to develop a general infrastructure for evolution-aware program analysis. We started working on this in the context of runtime monitoring of MOP properties:

    Evolution-Aware Monitoring-Oriented Programming
    Owolabi Legunsen and Darko Marinov and Grigore Rosu
    ICSE NIER'15, ACM, pp 615-618. 2015
    PDF (http://fslweb.cs.illinois.edu/FSL/papers/2015/legunsen-marinov-rosu-2015-icse/legunsen-marinov-rosu-2015-icse-public.pdf) , Slides(PDF) (http://fslweb.cs.illinois.edu/FSL/presentations/2015/2015-05-21-legunsen-marinov-rosu-ICSE.pdf) , JavaMOP (http://fsl.cs.illinois.edu/index.php/JavaMOP) , DOI (http://dx.doi.org/10.1109/ICSE.2015.206) , ICSE NIER'15 (http://2015.icse-conferences.org/NIER) , BIB (http://fslweb.cs.illinois.edu/FSL/papers/2015/legunsen-marinov-rosu-2015-icse/legunsen-marinov-rosu-2015-icse-ref.bib)

    But this is only the beginning, there is a lot of work to be done. I am particularly interested in a general approach to state what a software analysis tools does, and how the code changes interfere with it. In other words, how does a delta on your code impact the analysis? What properties need not be verified/checked because they are not impacted? Among those you need to verify, do you need to verify them fully or only partially? In the case of runtime analysis you may only need to instrument some paths, and in the case of symbolic execution you may only need to analyze some paths, all depending on the changes in the code and on the property to check.