Proving Partial Correctness Beyond Programs

Vlad Rusu Vlad.Rusu@inria.fr

Inria, Lille, France

Partial correctness is perhaps the most important functional property of algorithmic programs. It can be broadly stated as: on all terminating executions, a given relation holds between a program's inputs and outputs. It has been formalised in several logics, from, e.g, Hoare logics [1] to temporal logics [2].

Partial correctness is also a relevant property for any class of specification that has a notion of terminating execution. For example, communication protocols have both nonterminating executions (all messages are forever lost and re-sent) and terminating executions (all messages sent are properly received). Here, partial correctness may, for instance, require that on all terminating executions, the set of messages corresponding to the transmission of a given file are received *and* the reception of every message is acknowledged.

How can one naturally specify such *generic* partial-correctness properties, and how can one verify them in a *maximally trustworthy* manner? One possibility would be to use Hoare logics, but that solution is not optimal because Hoare logics intrinsically require *programs* (as their deduction rules focus on how program-instructions modify logical predicates), and we are targeting systems specified in formalisms that are *not* programs but more abstract *models*, e.g., one more naturally specifies communication protocols in some version of state-transition systems. Another possibility is to state partial-correctness properties in temporal logic and to use a model checker to prove the temporal formula on a state-transition system specification. This is a better solution, since it stays at a model-abstraction level; however, as we are aiming at trustworthy verification, this is not satisfactory as even in case of a successful verification, one's trust in the result is limited as one does not get independently-checkable verification certificates. Moreover, model checkers are limited to essentially finite-state systems (perhaps up to some data abstraction), a limitation we want to avoid.

Contribution We thus propose a generic approach implemented in the Coq proof assistant [24], a system trustworthy enough to be widely regarded as a *certification* tool, in the sense that Coq proofs are independently machine-checkable certificates. We define and implement therein a notion of *Abstract Symbolic Execution* (hereafter, ASE) to capture a generic notion of symbolic execution for otherwise arbitrary specifications. As property-specification language we adapt *Reachability Logic* [3–7] (hereafter, RL) to any system for which ASE is defined. We propose a new deductive system for this version of RL and prove its soundness both on paper and in the Coq proof assistant [24]; the latter provides us with a Coq-certified RL deductive system. We also prove a relative completeness result for our deductive system, which, although theoretical in nature, also has a practical value, since it amounts to a strategy for applying the proof system that does

succeed on valid RL formulas. Initial examples (proving the Needham-Schroeder-Lowe protocol [11][1]) suggest that the approach is applicable in practice.

Related Work Reachability Logic [3–7] is a formalism initially designed for expressing the operational semantics of programming languages and for specifying programs in the languages in question. Languages whose operational semantics is specified in (an implementation of) RL include those defined in the $\mathbb{K}$ framework [8], e.g., Java and C. Once a language is formally defined in this manner, programs in the language can be formally specified using RL formulas; the typical properties expressible in RL are partial-correctness properties. The verification of such formally-specified programs is performed by means of a sound deductive system, which is also complete relative to an oracle deciding certain first-order theories. Recently, it has been noted that RL is also relevant for other classes of systems, i.e., rewriting-logic specifications [9, 10]. In this paper we adapt RL to an even broader class of specifications - essentially, any specification for which an abstract notion of symbolic execution is defined.

The papers [3–7, 10], which describe several variants of RL (earlier known as *matching logic*[2]). The version of RL that we are here adapting is the *all-paths* version [6], suitable for concurrent nondeterministic systems, in contrast with the *one-path* version [5] for sequential programs.

We note that Coq soundness proofs have also been achieved for various proof systems for RL [5, 6]. Those proofs did not grow into practically usable RL interactive provers, however, because the resulting Coq frameworks require too much work in order to be instantiated even on the simplest programming languages[3]. By contrast, our ambition is to obtain a practically usable, certified RL prover within Coq, directly applicable to formalisms more abstract than programs.

Our approach is based on a generic notion of symbolic execution, an old technique that consists in executing programs with symbolic values rather than concrete ones [13]. Symbolic execution has more recently received renewed interest due to its applications in program analysis, testing, and verification [14–19]. Symbolic execution-like techniques have also been developped for rewriting logic, including rewriting modulo SMT [20] and narrowing-based analyses [21, 22].

Finally, abstract interpretation [23] provides us with a useful terminology (abstract and concrete states, abstract and concrete executions, simulations, etc) which we found most convenient for defining abstract symbolic execution.

References

1. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12 (10): 576580, 1969.

[1] Extended paper and Coq code is available at `https://project.inria.fr/rlase`.

[2] Matching logic now designates a logic for reasoning on program configurations [12]. These changes in terminology are a side effect of the field being relatively recent.

[3] To our best understanding, obtaining certified RL interactive provers was not the goal of our colleagues; rather, they implemented automatic, non-certified provers.

2. Z. Manna and A. Pnueli The temporal logic of reactive and concurrent systems - specification Springer Verlag, 1992.
3. G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP 2012*, Springer LNCS 7392, pages 351–363.
4. G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA 2012*, ACM, pages 555–574.
5. G. Roşu, A. Ştefănescu, Ş. Ciobâcă, and B. Moore. One-path reachability logic. In *LICS 2013*, IEEE, pages 358–367
6. A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. Moore, T. F. Şerbănuţă, and G. Roşu. All-path reachability logic. In *RTA 2014*, Springer LNCS 8560, pages 425–440.
7. A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91
8. The $\mathbb{K}$ semantic framework. `http://www.kframework.org`.
9. D. Lucanu, V. Rusu, A. Arusoaie, and D. Nowak. Verifying reachability-logic properties on rewriting-logic specifications. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer*, Springer LNCS 9200, pages 451–474.
10. Stephen Skeirik, Andrei Stefanescu, and Jose Meseguer. A constructor-based reachability logic for rewrite theories. Technical report, University of Illinois at Urbana-Champaign, 2017. Available at `http://hdl.handle.net/2142/95770`.
11. Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. *Software - Concepts and Tools*, 17(3):93–102, 1996.
12. G. Roşu. Matching logic. In *RTA 2015*, LIPICS volume 36, pages 5–21.
13. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
14. J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution tool for verification. In *CAV, 2012*, Springer LNCS 7358, pages 758–766.
15. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Software Engineering Notes*, 26(5):142–151, 2001.
16. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security 2006*, pages 322–335.
17. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE 2005*, ACM, pages 263–272,
18. C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE 2010*, ACM, pages 179–180.
19. D. Lucanu, V. Rusu, and A. Arusoaie. A generic framework for symbolic execution: A coinductive approach. *J. Symb. Comput.*, 80:125–163, 2017.
20. C. Rocha, J. Meseguer and C. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Meth. Program*, 86(1):269-297, 2017.
21. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. Higher-Order and Symbolic Computation 20(1-2):23–160, 2007.
22. K. Bae, S. Escobar and J. Meseguer, Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. *RTA 2013*, LIPICS volume 21, pages 81–96.
23. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, ACM, pages 238–252.
24. The Coq proof assistant reference manual. `http://coq.inria.fr`.