Bounded Model Checking of Recursive Programs with Pointers in $\mathbb{K}$

Irina Măriuca Asăvoae[1], Frank de Boer[2,3],
Marcello M. Bonsangue[3,2], Dorel Lucanu[1], Jurriaan Rot[3]

[1] Faculty of Computer Science - Alexandru Ioan Cuza University, Romania
[2] Centrum voor Wiskunde en Informatica, The Netherlands
[3] LIACS - Leiden University, The Netherlands

The current work proposes a method, and its implementation, for model checking sequential imperative programs with pointers and recursive procedure calls. To achieve this goal, the following development steps are taken: *(a)* introduce *Shylock*, an abstract programming language with an improved abstract semantics designed to overcome the problem of unbounded pointer allocation; *(b)* use the $\mathbb{K}$ framework [7] to specify Shylock and its semantics, as such obtaining an interpreter for Shylock and the for-free model checking capabilities available in $\mathbb{K}$ via Maude [3]; *(c)* address the limitations imposed in Shylock's verification by Maude's model checker prerequisites by several solutions for defining bounded model checking. In the followings we present in more details and justify the appropriateness of the *(a, b, c)* steps.

(a) Firstly, we introduce Shylock - a simple imperative programming language with pointers and recursive procedures which use local and global variables to store references in the heap. The recursion allows unbounded object allocation, hence the heap size may grow unboundedly making the direct verification of such programs more challenging. We advertise Shylock as a simple but expressive enough language to represent an abstraction for imperative or object oriented programming languages like C or Java. More to the point, we actually promote the abstract interpretation perspective, [2], that advocates applying analysis or verification methods to an abstract, simplified model. The advantage of the abstract model consists in the enhanced effectiveness for verification, or even existence of decidability results in certain cases. Moreover, the verification results obtained on the abstract model automatically have concrete interpretation, based on the relation between the real/concrete programming language and the abstract one. To show the relevance of Shylock, we plan to apply meta-programming techniques for generating suitable Shylock abstract models for C and Java programs.

The definition of the abstract operators in abstract interpretation is, in our settings, tantamount to giving semantics for Shylock. We propose an abstract semantics which introduces a novel mechanism for managing the object identities in the heap. This mechanism resides in a combination of *memory reuse* upon generation of fresh object identities combined with a *renaming scheme* to resolve possible resulting clashes. The memory reuse allows the allocation of any object identity which is unreachable in the context of the current procedure but not necessarily in the context of pending local environments on the stack. The renaming scheme resolves possible resulting name clashes upon changing the context, i.e., upon procedure call/return. The proposed mechanism

significantly simplifies, comparing to the standard approach in [1], the heap transformation operations performed on procedure calls and returns.

(b) Secondly, we faithfully implement the formal semantics of Shylock using $\mathbb{K}$ -Maude [8]. By faithful implementation we understand that the $\mathbb{K}$ specification and the formal semantics of Shylock are bisimilar. The $\mathbb{K}$ specification for Shylock has the following desiderata: (1) to contribute to the pool of languages defined in $\mathbb{K}$ with a fairly elaborated language semantics; (2) to set the grounds for an extensive future work of semi-automatically proving the simulation between the concrete and abstract semantics; (3) most importantly, to use the semantics for verification by model checking techniques.

(1) The quality of solely Shylock's semantics lies in the specification of the rules for fresh object creation (i.e., the memory reuse), and procedure call/return (i.e., the renaming scheme). Each element in this entire mechanism is implemented equationally, i.e., by means of structural $\mathbb{K}$ rules which have equational interpretation when compiled in Maude. Hence, if we interpret Shylock as an abstract model for C or Java, the $\mathbb{K}$ specification for Shylock's abstract semantics renders an equational abstraction.

(2) The $\mathbb{K}$ framework already offers the semantic specification of languages such as C [4] and KOOL. However, it is notoriously difficult and not recommended to alter a concrete semantics for such a purpose as abstraction. Instead, the abstract interpretation standardized approach advocates for the following separation of concerns: the concrete and the abstract models are defined in isolation, the analysis and verification methods are developed in the abstract and are projected in the concrete using the a priori proved relation between concrete and abstract. To convey the same perspective in $\mathbb{K}$, we can consider the C specification as the concrete model, the Shylock specification as the abstract model, and we are left to prove the simulation relation between the two. As such, once established the relation between concrete and abstract, Shylock is yet another witness to the versatility of the equational abstraction methodology [5].

(3) We employ the $\mathbb{K}$ specification of Shylock to derive verification capabilities provided for free by the Maude LTL model checker. The atomic properties are defined as regular expressions for the heap structure exploration. Maude's model checker provides a substantial expressivity w.r.t. atomic properties in comparison with other model checking tools. This feature justifies the appropriateness of using $\mathbb{K}$ -Maude for Shylock in order to consequently model check Shylock programs. However, due to the prerequisites imposed by the Maude's model checker, we can successfully verify only a restricted class of Shylock programs. We address this issue in step *(c)*.

(c) Our renaming scheme defined for resolving name clashes in the context of memory reuse is based on the concept of *cut points* as introduced in [6]. Cut points are objects in the heap that are referred to from both local and global variables, and as such, are subject to modifications during a procedure call. Recording cut points in extra logical variables allows for a sound return in the calling procedure, enabling a precise abstract execution w.r.t. object identities. Hence, under the assumption of a bounded visible heap, the programs can be represented by finitary structures, namely *finite pushdown systems*. Hence, the $\mathbb{K}$ specification for Shylock programs is bisimilar with a finite state pushdown system and compiles in Maude into a rewriting system. Obviously, in the presence of recursive procedures, the stack in the pushdown system grows unboundedly

and, even if the abstraction ensures a finite state space, the equivalent transition system is infinite and so is the associated rewriting system. To overcome this drawback we approach two solutions: (i) apply a further abstraction to enforce the finiteness of the equivalent transition system; (ii) apply collecting semantics over the current abstraction to specify model checking algorithms for pushdown systems.

(i) In order to use Maude's model checker for Shylock programs with recursive procedure calls we need to employ another abstraction over the current one. Basically, we add to the semantics a computation bounding mechanism which cuts the recursion after a predefined number of recursive calls, hence semantically transforming the model checking into bounded model checking. This composition of abstractions is naturally expressed in $\mathbb{K}$ by adding new cells containing the bound for the recursion depth and the current depth for each recursive call, and by transforming the procedure call rewrite rule into a conditional rewrite rule. It is relatively easy to prove that there is a stuttering simulation between Shylock and recursion-bounded Shylock. Using this solution we obviously lose completeness of the model checking procedure for Shylock.

(ii) The value of Shylock abstract semantics for model checking concerns its pushdown system characterization for which there are standardized model checking algorithms [9]. Hence, we propose embedding the $\mathbb{K}$ specification for Shylock abstract semantics into a $\mathbb{K}$ specification for Shylock collecting semantics. The Shylock collecting semantics renders an exhaustive execution of Shylock's abstract semantics, is terminating, and produces the reachable state space and/or the reachability automaton. Using the reachable state space we can model check heap invariants (defined by the atomic propositions), while the reachability automaton can be coupled with the the LTL property to obtain the the full power LTL model checking capabilities. Consequently, this second solution preserves the completeness of the model checking for Shylock. This is currently ongoing work available for presentation in the near future.

References

1. A. Bouajjani, S. Fratani, and S. Qadeer. Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In *CAV 2007*, vol. 4590 of *LNCS*, Springer 2007.
2. P. Cousot, and R. Cousot. Abstract Interpretation and Application to Logic Programs, *Journal of Logic Programming*, 13(2–3), pp. 103–179, 1992.
3. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *WRLA 2002*, vol. 71 of *ENTCS*, pp. 162–187, 2002.
4. C. Ellison, and G. Roşu. An Executable Formal Semantics of C with Applications. In *POPL 2012*, to appear.
5. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In *Theoretical Computer Science*, vol. 403(2-3), pp. 239–264, 2008.
6. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL 2005*, pp. 296–309, 2005.
7. G. Roşu, and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
8. G. Roşu, and T. F. Şerbănuţă. K-Maude: A Rewriting Based Tool for Semantics of Programming Languages. *WRLA 2010*, vol. 6381of *LNCS*, pp.104–122, Springer 2010.
9. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.