Towards the verification of RAISE specifications through Model Checking

Juan I. Perna
Software Engineering Group
Universidad Nacional de San Luis
San Luis – Argentina
jiperna@unsl.edu.ar

and

Chris W. George
International Institute for Software Technology
United Nations University
Macao, SAR – China
cwg@iist.unu.edu

Abstract

Ensuring the correctness of a given software component has become a crucial aspect in Software Engineering and the Model Checking technique provides a fully automated way to achieve this goal. In particular, the usage of Model Checking in formal languages has been reinforced in the last decades because the specifications themselves provide an abstraction of the problem under study (whether created by abstraction from the software or by hand) and the properties validated at the specification level can be warrantied to be preserved until implementation.

In this paper we focus on the main issues for adding Model Checking functionalities to the RAISE specification language and present the most important characteristics of our current approach for doing so. An outline of the main issues and problems faced in the process and possible ways to solve them are also presented.

Keywords: Model Checking, RAISE, formal methods, verification techniques.

Introduction

The utilization of Model Checking techniques for software components verification has been subject of significant research and study during the last decade [1, 2]. This increasing popularity of Model Checking is due to the high level of automation achieved (compared to other verification techniques such as testing or validation by proof) and to the ability of producing counterexamples when a given property is not satisfied.

Regardless of its context of application, the model checking technique is based on the exploration of all possible reachable states by the system under study and the verification of the satisfaction of properties (expressed in a sub logic of the CTL family [3, 2]) on those states. Due to this exhaustive exploration of the state space of the problem, Model Checking suffers from the *state explosion* problem (i.e. the size of the computation increases exponentially with respect to the size of the original problem). As the state explosion problem is a major limitation for the applicability of model checking in *real* problems (due to their complexity and size), several techniques have been developed to cope with this issue. In particular, *symbolic model checking* [4, 1] and *abstraction* [5, 6] are the most used ones.

In the context of formal or rigorous methods for software development, several attempts have been made to incorporate Model Checking techniques given the advantage that software specifications are, essentially, abstractions of the desired system. In particular, there have been several approaches to the incorporation of model checking techniques in order to verify whether a given property is preserved throughout the whole development process or at a certain abstraction stage. Some well known examples of the incorporation of model checking functionalities into formal languages such as Z [7, 8] or process algebras [9] such as CSP [10, 11] can be analyzed from [12, 13, 14].

Regarding RAISE [15], no support for model checking is currently provided. In particular,

RAISE provides several tools regarding verification, such as **code generators** to several languages in order to *run* the specification's code; **test cases** (including test coverage analysis and mutation testing) and a **translator to PVS** so important properties or invariants from the specification can be proved correct. However relatively easy to do, testing is necessarily incomplete and it only allows the user to gain certain confidence about the possible satisfaction of the desired properties. Proofs, on the other hand, provide certainty and completeness but are very hard and time consuming to be done.

This work presents an ongoing project to incorporate the automated functionalities of model checking to the RAISE language and some preliminary results on the development of an automated tool to carry out the verification of PLTL assertions over a given specification.

Model Checking engine

In the context of RAISE[15], software is specified in a step-wise way. In particular, the development process (following the *RAISE development method*[16]) is carried out as a sequence of steps, starting from the specification of the system at a high level of abstraction and progressing by successively adding details towards a more concrete (and thereby closer to the implementation) specification. This way of development (getting closer to the implementation by decreasing the level of abstraction) is particularly suitable for model checking because it allows the verification of properties in early stages in the development process, where an abstract level description is obtained *for free* with the specification (actually, the specification itself is the abstract description of the system under study). Once verified, the RAISE development process warranties the preservation of the properties until the actual implementation of the system.

Given that RAISE specifications can provide a complete yet abstract description of the system under study, a design decision must be made at this point: to develop another model checking engine (customized particularly for RAISE) or to try to use one of the currently available model checking engines.

About developing a brand new model checking engine for RAISE, it has the advantages of eventual better efficiency (compared with other more general model checkers if some particularities from RAISE's language can be used in order to achieve more efficient algorithms) and a closer *fit* of the model checker's language with RAISE's one (no additional syntactic or semantic restrictions but the ones inherent to the model

checking techniques). On the other hand, using an already developed model checker provides the security of a well tested verification engine combined with the possible advantages from future developments of these tools that can be immediately used if backward compatibility is preserved in the model checker.

In order to allow a proper evaluation of the real implications of the usage of a third party model checker, several model checkers were evaluated to analyze the real restrictions that a general purpose model checking engine will impose if used to verify properties in RAISE specifications. Towards this goal, SPIN [17], Symbolic Analysis Laboratory (SAL) [18] and SMV [4] were analysed. The first differs from the others in that it is based in the construction of an explicit state representation of the system, while the others are based on the symbolic approach to model checking.

The modelling language provided by the model checkers was also taken into account (the expressiveness of this language has a major impact on the kind of constructs that can be translated into it). In this direction, both SPIN's language (PROMELA) and SMV's one are imperative and not very suitable for modelling applicative descriptions. SAL's language, on the other hand, is similar to PVS and provides an applicative modelling language combined with very powerful constructs to describe transition systems. Another interesting feature in SAL's language is that it is used as a front end to a whole verification tool set [19] (symbolic, infinite-bounded, backwards model checkers, deadlock analyzer, simulator and path finder) that can be used according to the user's needs.

Taking into account that SAL's applicative language provides enough constructs to encode most of RAISE syntactic constructs, that it belongs to the symbolic model checking family (i.e. it can, in many cases, avoid the state explosion problem) and has a good overall performance, it was chosen as translation target for RAISE's specification model checking, discarding in this way the option of the implementation of a new model checker.

Under this development decision, the implementation of the tool to incorporate Model Checking techniques into RAISE would involve acquiring information from the specification and the generation of final code for SAL. In particular, the new tool can obtain the Abstract Syntax Tree (AST) created by the already existing type checker, transform it (probably in more than one pass over it) into another syntax tree closer to SAL syntax and then, unparse it into the final code for the model checker. This process is outlined in figure 1.
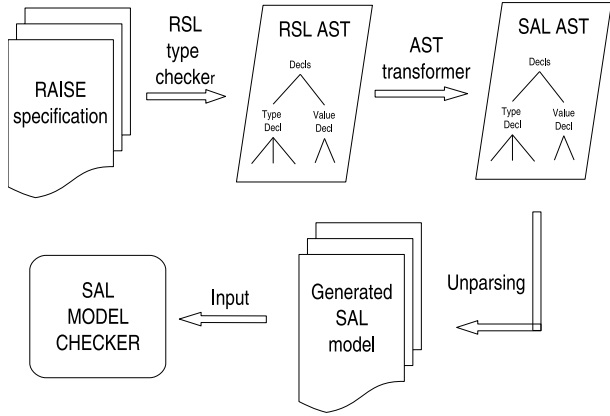
Figure 1: Stages in the translation process

Translation issues

After the choice of using a third-party model checker, several problems in the translation to SAL are expected to arise given the logical differences between the expressive power of RAISE and SAL languages (the former is a whole specification language while the latter is guarded-command-based transition system language enriched with applicative constructs).

Our ongoing research has analysed SAL's syntactic and semantic expressive power looking for possible areas where the translation from RAISE would be problematic.

Following, we present a list of the most important issues found and a brief description of each of them.

- No *define-before-use* rule in RSL and necessary in SAL. This restriction in the target language forces the translator to collect all declarations and sort them according to a *declaration dependency*, where a declaration depends on other declaration if the former refers to the latter on its definition. This relationship among declarations can be iteratively calculated and the declarations sorted with an algorithm based on [20].

- No support for collections in SAL. As collections (sets, maps and lists) are very commonly used data structures in RAISE it is necessary to devise a way to represent them inside SAL's language.

  Given the fact that SAL supports *lambda-functions*, sets, maps and list can be, then, represented with lambda functions with adequate domain and range. However elegant and efficient, this solution may face problems to represent maps or lists defined by complex comprehended expressions (for example, those maps with a pattern of the form $[e1(x) \mapsto e2(x) \mid x : T \cdot p(x)]$ where $e1 : T \to U1$, $e2 : T \to U2$).

- No support for partial functions in SAL. As partial functions can be seen as total functions over a more restricted domain, it would be possible to make the translator use subtypes in function signatures (to make them total) or to keep them using a maximal principle and assume partial functions will never be called with arguments outside the domain that the function is prepared to accept. The latter approach seems to be more efficient and, when this assumption does not hold, the user will receive a run-time error (while model checking the code).

  On the other hand, it would also be possible to modify the translation and the type system in order to model check this assumption of subtype correctness. To do so, every type in the system should be *lifted* into a variant that can hold the type's normal value or a special value **nav** (*Not A Value*) to signal the non-satisfaction of a subtype assumption. As a matter of fact, this approach can be extended and be used to verify not only subtype correctness but also the whole set of confidence conditions (precondition/postcondition satisfaction, correct application of maps and functions, etc.).

- No clear Transition System concept in RAISE. As RAISE was not meant to be a language to specify transition systems, it does not provide constructs for doing so. On the other hand, it is crucial for model checking to have a transition system (or a *kripke transition graph* [1]) to be able to calculate the states reachable by the system.

  Trying to extract the transition system inherent to a RAISE specifications would be the ideal solution but, as specifications are just a set of functions, it would be impossible to deduce it automatically for the general case. Even if this were possible, how to determine the initial state of the *deduced* transition system is also not clear for the general case.

  To solve this issue, it is planned to incorporate a whole new construct into the RAISE language to allow the user to specify the *state variables* of the transition system and the way they change, using guarded commands.

- No clear way to express temporal logic properties in RAISE. Even though RAISE provides a way to express invariants or properties over the whole specification by means of **axiom** declarations, they were not meant to express the temporal properties that PLTL

logic provides. Then, allowing the user to model check properties expressible only by means of axioms would be too restrictive and would not be taking full advantage of model checking verification power.

Taking this into account, a new construct to express assertions is being added to RAISE and LTL temporal operators will be available for the user within its scope.

Limitations

Encoding a whole specification language (which is devised with the idea of expressiveness in mind) into a model checking language (which is designed for transition system description and analysis) will, inevitably, end up in some areas of the specification language being left outside of the translation. In the case of this work, the limitations on the translation are mainly regarding recursion and implicitly defined values.

Recursive types

The only data declaration in SAL that has the syntactic expressiveness necessary to allow recursive type definitions is the `datatype` that will be used to encode RSL variant types during the translation. However syntactically possible, none of the SAL tools allow the usage of this construct if it involves some kind of recursion.

The reason for this restriction is that there is no way to statically (i.e. during compilation time) resolve the recursion associated with the structure in order to calculate the set of possible values in the type. It is easy to realize that this is a serious shortcoming that can not be overcome if it is taken into account that a finite representation must be available to every type if model checking techniques are going to be applied. In particular, symbolic model checking theory relies on the representation of the system as strings of binary (boolean in the general case) values and it is impossible to find a codification for the state of the system if one of the types that comprises it has an unknown/undefined size.

Recursive functions

Recursive functions, on the other hand, do not constitute a serious limitation for model checking techniques but, in general, a way to determine the depth of the recursion or the so called *measure* of the function is required. In the SAL case, according to the authors in [18], the SAL type checker was supposed to be able to automatically calculate the measure of a function (provided that the language was initially devised to

be very simple). This assertion proved to not to be true in the general case and providing the user with means to state recursive function measures is regarded as future work for the SAL development team.

Implicitly defined values

Implicitly defined values are a very valuable resource when developing abstract specifications because they allow the introduction of components that will be properly defined in successive refinement steps. On a more conceptual level, implicitly defined values can be seen as a reference to functionality with underspecified behaviour at the current level of abstraction.

On the other hand, values are used in in the model checking paradigm as a mean to define how the system under study must evolve. In this context, the introduction of undefined values in a model checking transition system is not conceivable if it is taken into account that undefined behaviour would be introduced in the system under study's evolution.

Conclusions and future work

We have briefly presented our approach for the inclusion of model checking functionalities to the RAISE specification language. In particular, the main translation problems have been presented, the possible solutions briefly outlined as well as the main limitations of the current approach.

Regarding future work, finishing the current version of the tool and the implementation of a lifted type system in order to allow confidence condition verification are our major concerns.

It will also be interesting to explore SAL's construct `implements` that allows the usage of model checking exhaustive state exploration in order to verify if there is a refinement/abstraction relationship between two modules. With this construct, it will be possible to actually verify (without the need of a proof) specifications that are derived from each other.

References

[1] Edmund M. Clarke Jr., Orna Grungberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[2] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A.Petit, L. Petrucci, Ph. Schnoebelen, and P. Mackenzie. *Systems and Software Verification, Model Checking Techniques and Tools*. Springer-Verlag, 1998.

[3] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, England, 2000.

[4] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[5] S. Graf. Verification of a distributed cache memory by using abstractions. In *Computer Aided Verification*, volume 697 of *Lectures Notes in Computer Science*. 5th International Conference in Computer Aided Verification, Springer, 1994.

[6] Jurgen Dingel and Thomas Filkorn. Model cheking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Pierre Wolper, editor, *Computer Aided Verification*, volume 939 of *Lectures Notes in Computer Science*. 7th International Conference in Computer Aided Verification (CAV '95), Springer, 1995.

[7] J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs: An Advanced Course*, pages 343–410. Cambridge University Press, 1980.

[8] J. P. Bowen, R. B. Gimson, and S. Topp-Jørgensen. Specifying system implementations in Z. Technical Monograph PRG-63, February 1988.

[9] Wan Fokkink. *Introduction to Process Algebra.* Springer-Verlag, Berlin, Germany, 2000.

[10] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[11] Jonathan P. Bowen and Michael G. Hinchey. *High-Integrity System Specification and Design.* Springer Verlag, 1999.

[12] Graeme Smith and Luke Wildman. Model Checking Z Specifications Using SAL. In *ZB 2005*, pages 85–103. International Conference of Z and B Users, Springer, 2005.

[13] Formal Systems (Europe) Ltd. Failures-divergence refinement – FDR 2 user manual, 1997.

[14] Michael Leuschel, Thierry Massart, and Andrew Currie. How to make FDR spin LTL model checking of CSP by refinement. *Lecture Notes in Computer Science*, 2021:99+, 2001.

[15] The RAISE Language Group. *The RAISE Specification Language.* Prentice Hall International (UK), 1992.

[16] The RAISE Method Group. *The RAISE Development Method.* Prentice Hall International (UK), 1995.

[17] Gerard J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2003.

[18] *The SAL Language Manual.* http://sal.csl.sri.com/doc/language-report.pdf.

[19] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[20] Aristides Dasso and Chris George. Translating RSL into PVS. Technical report, International Institute for Software Technology - United Nations University, 2002.