Autocoding from Formal Models

Efficient and Correct-by-Construction

Entalus
Computer Science Labs

www.entalus.com

November 2023

Need. Formal methods in industrial practice are most often used for verifying abstract algorithms and software architectures, which are then hand-coded in latter stages. This process obviously is resource-intensive and extremely error-prone. With the recent PVS2C technology, however, we are now able to bridge this gap by autogenerating efficient, correct-by-construction code from high-level formal specifications.

The PVS specification languages contains a rich executable programming language, which includes conditional expressions, functional abstraction and application, let-binding, record/tuple/function update, and well-founded recursion. Its typing system is much richer than what is usually present in current programming languages in that it admits predicate subtypes, dependent record/tuple/function types, and recursive datatypes, and its “module” system for structuring specifications and programs is unique in that it is based on parametric theories.

The PVS2C [1,2,3] compiler generates verifiably correct and efficient C code from a well-typed PVS program. The generated C code is self-contained in that it does not rely on a run time environment. Type-checking, particularly through the discharging of PVS type correctness obligations, ensures that there are no buffer overflows, null dereferences, uncaught exceptions, division by zero, etc. Altogether, the execution of well-typed PVS expressions is free of runtime exceptions (modulo exhausting heap or stack space). Crucial components of the PVS2C translational steps have been formally verified in PVS itself.

State-of-the-practice industrial autocoding technology is currently often based on Simulink, which essentially is an imperative programming language in the disguise of a graphical language. PVS2C is much more powerful in that it efficiently autogenerates

code in an imperative programming language from a functional specification with a rich typing system.

Our experiments with LLMs for autogenerating code from specifications demonstrate that they often are able to recall intended or useful code fragments for given specifications, but too often these answers are plainly wrong or they are over-interpreting the given specifications (or prompts), and no convincing arguments are provided as to the correctness or efficiency of the proposed solution. We envision, however, that in the near symbolic programming tools such as PVS2C are being integrated as task-specific oracles into more general LLM programming assistants.

Benefits. PVS2C generates verifiably correct C code from verified algorithm, thereby largely automating the coding stage of software development. There is no need to formalizing the complete semantics of C with all its rigmaroles. This approach also supports a clean separation between algorithmic verification and the verification of machine-level properties such as the absence of runtime exceptions.

Engineering Support.

- Creating formal specifications from clients' specifications, which is possibly also formally validated
- Generating efficient C (alternatively, Rust) code from formal specifications,
- Implementing and maintaining client-specific features in PVS2C, including support for further target programming languages
- Conducting case studies on clients' products on the potential benefits of efficient autocoding from specifications with respect to the usual code quality attributes such as correctness, reliability, maintainability, usability, portability, efficiency, security, testability, flexibility, scalability, compatibility, supportability, reusability, and interoperability.
- Certification package for PVS2C, which can be either based on translation validation or on a complete formal correctness proof of PVS2C within PVS
- Working out client-specific roadmaps for implementing efficient and correct-by-construction autocoding capabilities in the context of clients' development environment and processes

Benefits. PVS2C auto-generated code might be directly used as *production code*, as *golden reference models* for possibly more involved implementations, as *runtime monitors*, and as the basis for *generating test cases* from an executable specification.

As such, correct-by-construction code generation with PVS2C has the potential of revolutionizing the current practice of the development of critical software with huge potential cost savings, which typically are on the order of two to three magnitudes, by focusing on early lifecycle programming activities, and the prospect of totally correct production code.

There is the rather common misconception that correct-by-construction code eliminates the need for *unit testing*. But this belief is clearly wrong, *since the* correctness claim of the generated code *is with respect to the specification,* which itself can, in any non-trivial situation, only be approximate with respect to the code's logical and its physical execution environment. But unit testing can at least use the fact that errors might only show up in unspecified situations.

Altogether, the automation of latter stages of the software development process with PVS2C enables one to focus on early-lifecycle software development activities, where the majority of the costly errors are usually being made.

Contact.

Harald Ruess
Principal Partner, Entalus LLC

email: harald.ruess@entalus.com
phone: +1 (941) 312-2144

References.

[1] Shankar, Natarajan. A Brief Introduction to the PVS2C code generator. AFM@ NFM 5 (2017): 109-116.

[2] Courant, Nathanaël, Antoine Séré, and Natarajan Shankar. The correctness of a code generator for a functional language. Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21. Springer International Publishing, 2020.

[3] Férey, Gaspard, and Natarajan Shankar. Code generation using a formal model of reference counting. NASA Formal Methods Symposium. Cham: Springer International Publishing, 2016.