Real-world formal documentation

Thomas Tuerk

Independent Scholar, Albert-Otto-Str. 8, 65611 Brechen, Germany
thomas@tuerk-brechen.de

Abstract: In recent years there have been tremendous improvements in interactive theorem proving. Nevertheless, it is hardly used during development of even critical, well funded software projects. One reason is the intrinsic difficulty of formal proofs. However, with advancements in automation and user interfaces this reason becomes less and less important. In my opinion, nowadays preconceptions are a more severe hindrance. As soon as terms like *logic*, *proof* or *formal specification* are used, even very clever, highly skilled software engineers tend to think of some kind of *black magic*: way to complicated for mere mortals and while huge gains are luring, you need to sell your soul to get them. Another reason why interactive theorem proving is not commonly used is – in my experience – that often a large initial investment is needed and progress and benefits are hard to measure.

I believe formal methods and in particular interactive theorem proving are vital to deal with the ever increasing complexity of hard- and software. However, before they get more widely used, the issues discussed above need addressing in my opinion. Therefore I started developing a tool called *Advanced Documentation and Testing Tool* (ADATT). Superficially it looks like yet another functional style programming language accompanied with a compiler and other development tools. Specifications written with ADATT can be exported to common theorem provers for reasoning. Moreover they can be exported to common programming languages for execution. In contrast to similar tools like *Lem* it is also possible to write *partial* specifications. Users can start with completely informal documentation in natural language, which ADATT can use to produce production quality documents. Step by step more formal content can be added. To support common development workflows, there should be an immediate return of invested effort and progress should be easily measurable. A special focus is on using partial specifications for advanced testing.

In this abstract, I will present the ideas behind ADATT. The development is still in a very early stage. There is no working prototype yet. However, I hope by presenting the ideas at an early stage it is possible to start discussions and perhaps find collaborators.

1 Motivation

I'm an interactive theorem prover guy. I have been working with HOL 4, Isabelle/HOL and Coq. I mainly used them to improve trust in real-world systems. Thereby I always followed a rather pragmatic approach. My goal was to find bugs and increase the trustworthiness of systems. Especially when reasoning about real-world systems, there is a non-trivial gap between the real system and the model of the system one can reason about. To gain trust into the real system, one needs to show that the model somehow corresponds to the real system (e. g. via careful code review or conformance testing) and additionally show some interesting properties of the model.

I have been working on this kind of verification projects both in academia and industry. Most recently, I worked in industry and tried to harden a microhypervisor using the Coq proof assistant [1]. In my experience, most problems are found while building a formal model of the computer system. This is due to the fact that building a formal model usually involves a detailed review of the existing system and requires the clarification of ambiguous and missing information. In addition, many bugs are found while proving simple properties about isolated parts the model, as this reveals simple implementation mistakes. Proving deep properties tends to reveal comparably few bugs. These are usually bugs in the design of the whole system.

This means that building a formal model is a very worthwhile activity in itself for hunting bugs. However, once you have a formal model – especially if it is executable – it can be easily used for powerful testing, documentation and automated formal methods. Even better, I believe that no formal methods experts are needed to develop basic formal models. If you present formal specifications as high level programs, domain experts are in my experience willing to read and even write formal specifications. This is especially true, if writing such a formal specification has an immediate benefit.

If the development of formal models can partly be done by domain experts while developing and testing a system, the costs for using interactive theorem proving can be lowered. The communication between domain and formal method experts is simplified and the (partial) formal model is a very good basis for formal method experts to extend and reason about. For enabling this vision of having domain experts develop (partial) formal models, good tool support is essential.

2 Lem

The best tool I know for the purposes stated above is Lem [2]. *"Lem is a lightweight tool for writing, managing,*

and publishing large scale semantic definitions"[1]. It has the look and feel of a functional programming language. Large subsets of Lem specifications can be translated to OCaml as well as definitions for HOL 4, Isabelle/HOL and Coq.

Even before working on Lem, I was fascinated on how much effect the form of presentation of formal methods can have. Programmers are trained to write down precise definitions (that's what a computer program is after all). However, if you ask them to write down a *specification*, the average programmer refuses. It was a revelation to me to see how VeriFast manages to get programmers to specify loops by disguising loop-invariants as programs. This insight grew deeper, while working for Peter Sewell on Lem.

I'm a strong believer in the ideas of Lem. It is vital to bring domain experts and formal verification experts closer together. Moreover, providing an environment that looks like a programming language and supports the normal tools of a programming language is a good choice. It is important to be able to produce human-readable output, executable code and formal specifications from the same input. Even using a functional instead of an imperative language is (while not familiar to many domain experts) a very sensible compromise, since it is comparably straightforward to translate to logic. However Lem does not go far enough in my opinion. Lem is a good tool, if you want to write a complete executable formal specification. It is, however, not suitable to write partial specifications or informal documentation. Moreover, Lem has limited capabilities for testing and measuring progress.

3 ADATT

For these reasons I started developing a tool called *Advanced Documentation and Testing Tool* (ADATT). It is inspired by Lem, but has a different focus. Similar to Lem, ADATT allows writing executable specifications that resemble functional programs and can be translated to interactive theorem provers. However, as the name suggests, ADATT focuses on documentation and testing and considers interactive theorem proving as an extra.

Domain experts should be able to use ADATT to write natural language documentation without any formal content. This should not be much more cumbersome than using other tools. One should be able to produce production-quality documents. Formal content can be added step by step. There should be an immediate benefit for adding formal content. If one – for example – formally declares a function or type, the spelling of it should be checked in the documentation. If you add a type signature, typechecking could take place in the natural language definition.

It is vital that ADATT can deal with *partial* specifications. Partially is supported in multiple ways. Even just declaring a function together with a type signature is a partial specification. You can then add single test cases. These test cases should be easily executable against a real implementation. ADATT aims to provide good support for conformance testing by e. g. providing special code generation. ADATT will support *code contracts* as well as families of executable tests. There is a separate syntactic construct for adding non-executable properties of the function. These cannot be used for testing and are instead intended to be checked by interactive theorem proving. One or more of such non-executable properties can be used as an axiomatic specification for theorem prover backends.

Ideally, however, we would like to end up with executable specifications. I can well imagine that different parts of the specifications are written by different people in different files. A programmer might start with natural language documentation, a function declaration and a few simple test cases. A test engineer might then add code contracts, some more tests and perhaps even a non-executable property. Finally, a formal methods expert might provide an executable specification and add non-executable properties. Different formal method experts might then use theorem provers of their choice to reason about the model, while ADATT keeps track of progress and links developments in various provers.

It is vital to provide some easy measurements of progress in order to integrate ADATT with existing software development processes. This means providing good reports and statistics. (How many functions are declared / specified / executable? Which tests were run when? ...) More interestingly, however, ADATT should be able to measure code-coverage.

4 Conclusion

ADATT is still in its very early stages. There is not even a prototype yet. There is a lot of work still ahead. This is in particular true, since ADATT needs a good user-interface, i. e. integration in commonly used IDEs. However, important design decision have already been made and implementation is well underway. Therefore, I would already value some comments.

References

[1] Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk. *Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor*, pages 69–84. Springer International Publishing, Cham, 2016.

[2] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, 2014. ACM.

[1] citation from `http://www.cl.cam.ac.uk/~pes20/lem`