This is a slightly edited version of a paper that appeared at the High Confidence Medical Device Software and Systems (HCMDSS) Workshop, June 2–3, 2005, Philadelphia, PA.

Proofs as a Substrate for Tool Integration Supporting High-Confidence Embedded Software

John Regehr
School of Computing, University of Utah
regehr@cs.utah.edu

Konrad Slind
School of Computing, University of Utah
slind@cs.utah.edu

Elsa Gunter
Department of Computer Science,
University of Illinois,
Urbana-Champaign
egunter@cs.uiuc.edu

1. Problem Statement

As the size and complexity of software in safety-critical embedded systems increases, the ability of programmers to deliver these systems in a timely fashion decreases. Specific difficulties are that embedded software must interact with the physical world in real time and that it must make efficient use of resources such as memory and energy. Our work is driven by the observation that the fundamental scarcity limiting our ability to create high-confidence embedded software is human developer time. A practical and incremental solution to this problem is *tool-rich software development* where software tools such as verifiers, static bug finders, stub generators, and optimizing compilers automate as many development tasks as possible.

A diversity of tools is important because for a given embedded system, some tools will present developers with a good value proposition and others will not. For example, extremely low-power devices with a few KB of main memory can benefit from static stack overflow detection [7] while systems with megabytes of RAM are unlikely to. The current state of the art is that tools are applied independently and sequentially to a software system. This mode of operation has two important drawbacks:

1. Each tool must discover all facts about a system that it requires. This makes it difficult to create new tools that deal with, for example, arbitrary C++ or x86 code. Furthermore there is considerable reinvention of low-level analysis infrastructure across software tools.
2. The correctness of program transformation tools such as compilers, link-time optimizers, and stub generators is often dependent on subtle assumptions about their input. It is difficult to trust a long chain of sophisticated tools, and difficult to debug problems that stem from complex interactions between developer assumptions and multiple tools.

2. Solution Outline

Research-level solutions to the first problem described above exist [1, 4], but the second problem—trusting a collection of useful but certainly buggy tools—remains unsolved and largely unaddressed. *Our thesis is that formal proofs can be used as a semantic substrate for ensuring the soundness of cooperating, independently developed program analysis and transformation tools.* In order to implement this vision, useful tools must emit enough information about their operation that a proof of soundness can be reconstructed. Then, when a final executable system has been generated, the individual proofs can be chained together and checked. By reasoning about individual analyses and transformations, rather than about entire analysis and transformation tools, the probably intractable problem of verifying these tools is avoided.

We do not wish to expose embedded software developers to theorem proving environments. Rather, as in proof-carrying code [6], proofs are an internal mechanism for verifying program properties. Proof-carrying code avoids the need to insert dynamic checks when running untrusted code; our goal is to prevent potentially untrustworthy tools from breaking safety-critical software.

Our insight is that at some level, an analysis or transformation tool must prove to itself that a program property holds before making use of this property. If it can be arranged that such tools can formalize and externalize the steps that comprise these informal and internal proofs, then more tools can be used, with more confidence, in the construction of safety-critical software. We hope that many of the pedantic aspects of dealing with proofs can be contained in proof-manipulation tools that are developed once, by experts, and that are separate from the tools that support the timely production of embedded software. Proofs are a suitable substrate for reasoning about analyses and transformations because theorem provers are a mature, well understood technology, because proofs should be general enough to express any desired program properties, and because they have a clear, generally agreed-on underlying semantics: mathematics.

3. An Example

Consider an ongoing (not yet published) project of ours that compiles higher order logic specifications of algorithms into ARM machine code, with the side effect of automatically producing a proof that the low-level code implements the high-level semantics. This tool performs translation validation [5]. ARM is a particularly good target for this kind of work because it is ubiquitous in embedded systems and because a semantics for ARM has been developed in higher order logic [3].

The binaries emitted by our compiler are unlikely to be especially fast—our goal is not to implement a highly optimizing compiler. Rather, we would like to use the Diablo link-time optimizer for ARM object code [2]. The problem is that Diablo's transformations invalidate the proof of correctness.

To make Diablo emit proofs, we observe that it is structured as an abstract interpreter. Recent work by Seo et al. [8] has shown how to turn abstract interpretation results into proofs. This technique, coupled with proofs of soundness and monotonicity of abstract transfer functions for object code that we are currently working on, will lead to automated generation of proofs that particular abstract interpretations were sound. Subsequently, proving the soundness of simple optimizations such as global constant propagation (from which Diablo derives much of its benefit) is not difficult.

We believe that chaining together two proof-generating tools—our compiler and a modified version of Diablo—will result in the creation of verified binaries that also make efficient use of resources. Furthermore, we will achieve these goals with less effort, and in a more modular fashion, than would other means to the same end, such as adding aggressive global optimization passes to our HOL→ARM translator.

Higher order logic implementations such as PVS, HOL, and Isabelle/HOL can be seen as providing semantic platforms upon which to stitch together proofs generated by a variety of tools. For example, they already provide mathematical theories needed for precise specification of medical software, such as number systems (integers, reals, fixed and floating point), temporal logics (CTL, LTL), set theory, etc. Proof tools such as simplifiers, first order logic proof search, and model-checkers can be programmed in such environments, but our main focus is on adapting existing program analysis tools, such as Diablo, to generate proofs and work together soundly.

4. Questions and Answers

Our work addresses problems in infrastructure, resource management, and verification/validation for medical device software and systems (MDSS).

What are the three most important research challenges?

The overall challenge is to start with an assortment of tools dealing with MDSS at various levels of abstraction—specifications, source code, object code, and hardware—and augment them with the capacity to argue for the soundness of their analyses and transformations through proofs.

Second, if proofs are to be used as a substrate for tool integration, then proof-manipulation tools must improve dramatically. Rather than expecting tools (and tool developers) to generate actual proofs, ways should be devised to conveniently create proof certificates for existing analysis domains.

Third, partial program correctness is being attacked from various angles: there exist proofs that real-time deadlines are met, proofs of type-safety, proofs of deadlock freedom, etc. These proofs usually exist in isolation; it is often unclear that they compose gracefully. Researchers should strive to apply various partial correctness results to an agreed-upon formal program semantics such as the HOL model of the ARM ISA. This will be particularly challenging for proofs that deal with very abstract execution models such as those that are typical in the real-time scheduling literature.

What are the three most important information technology research needs?

First, research on tools for embedded software is impossible without access to source code. If the MDSS industry is unwilling to make source code available then the next best course of action is to create a DARPA-style open experimental platform (OEP) for each major class of medical device. Funding agencies should allocate resources that will be used to create and disseminate high-quality OEPs that include requirements, specifications, documentation, hardware designs, simulators, and the embedded software itself. Research based on these OEPs will be far more likely to solve actual problems faced by industry, and furthermore, different research efforts that use a common platform can be directly compared and evaluated.

Second, open infrastructure for software tools is critical to lowering barriers to entry for validation and verification research. Bradley et al. [1] make this point nicely.

Third, formal (and hopefully executable) semantics for MDSS platforms should be developed. Fox's ARM semantics [3] is a good start. Models are also needed for other embedded processor architectures, for sensors and actuators, for multiprocessor devices, for configurable logic devices, for real-time operating systems, for embedded middleware, and for the environments in which MDSS devices operate.

What is a possible roadmap for the next 5–10 years?

In the short term, the next three years, we want to focus on the technical challenges outlined above. In the longer term, incentives will need to be created to convince tool vendors to generate proofs.

References

[1] Aaron R. Bradley, Henny B. Sipma, Sarah Solter, and Zohar Manna. Integrating tools for practical software analysis. In *Proc. of the 2004 CUE Workshop*, Vienna, Austria, October 2004.

[2] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of ARM binaries. In *Proc. of the 2004 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 211–220, Washington, DC, June 2004.

[3] Anthony Fox. Formal specification and verification of ARM6. In *Proc. of the 16th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 25–40, Rome, Italy, September 2003.

[4] Institute for Software Integrated Systems, Vanderbilt University. Analysis Interchange Format v1.5, March 2004.

[5] George C. Necula. Translation validation for an optimizing compiler. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 83–94, Vancouver, Canada, June 2000.

[6] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[7] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.

[8] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *Proc. of the 1st Asian Symp. on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.