Grand Challenges in Embedded Software

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Reinhard Wilhelm
Saarland University
wilhelm@cs.uni-sb.de

ABSTRACT

This is an introduction to the EMSOFT 2007 Panel on Grand Challenges in Embedded Software.

Categories and Subject Descriptors

C.3 [**Real-time and embedded systems**]

General Terms

Design, Performance, Reliability, Verification

Keywords

Embedded software, challenges, performance, scheduling, verification, analysis, design

Introduction

What are the grand challenges that the embedded software community is facing? The statistics of the submissions to EMSOFT 2007 reveal that embedded software is increasingly recognized as a key factor in embedded systems engineering yet the numbers only provide an unsatisfactory, deeper insight. Leading in the number of submissions are the usual suspects: Scheduling, Analysis and Verification, Synchronous Languages, Software Engineering, Performance Analysis, Model-Based Design, and Models of Computation. The only outlier, ranging just behind Software Engineering, is Systems Software for Flash Memory. This one we could consider a peculiarity of the recent history of EMSOFT, i.e., the fact that EMSOFT 2006 took place in Seoul, and that the South Korean industry is dominating the Flash market.

However, academic researcher's favorite topics seem to trail far behind, e.g. Component-based Design, Predictability, and Mobility. This raises several questions:

- Do we address the right issues?
- How do we know what the right issues are?
- How do we discover newly emerging grand challenges?
- Where do they originate after all?

Looking back, we can at least try to answer the last question. What were the origins of grand challenges that we have already addressed and partially solved?

- New applications, e.g. multimedia, mobile devices,
- Technological developments, ever higher integration causing energy problems,
- "Advances" in computer architecture, e.g. high-performance processors using caches, pipelines, and speculation, as well as in system architecture, e.g. Integrated Modular Avionics, AUTOSAR,
- Pressure of development and maintenance costs leading to component-based and object-oriented designs,
- Safety requirements of many embedded systems leading to reactive languages and model-based design.

The EMSOFT 2007 Panel on Grand Challenges in Embedded Software provides a forum for discussing these questions among the panelists and the embedded software community. For more details on the panel see also the following position statements.

Lothar Thiele

Department Information Technology and Electrical Engineering, Computer Engineering and Networks Lab, ETH Zürich, Switzerland
`thiele@tik.ee.ethz.ch`

Based on the characterization

$$\text{Embedded systems} = \text{computation} + \text{communication} + \text{resource interaction}$$

and the necessity to combine the computational and physical view of embedded software, the following challenges emerge:

Challenge 1 Component models whose interfaces talk about extra-functional properties like time, energy and resource interaction.

Challenge 2 Design methods that lead to timing-predictable *and* efficient embedded systems.

Challenge 3 Design methods that lead to dependable large-scale distributed embedded systems with wireless communication.

Challenge 4 Models of computation that talk about resources and mobility.

Challenge 5 Design and optimization methods that lead to small and extremely resource-constrained embedded systems.

Edward A. Lee

Department of EECS, University of California, Berkeley, Berkeley, CA 94720, USA
eal@eecs.berkeley.edu

Abstractions currently used in computing hide timing properties of software. As a consequence, computer scientists have developed techniques that deliver improved average-case performance and/or design convenience at the expense of timing predictability. For embedded software, which interacts closely with physical processes, timing is usually an essential property. Lack of timing in the core abstractions results is brittle and non-portable designs. Moreover, as embedded software becomes more networked, the prevailing empirical test-based approach to achieving real-time computing becomes inadequate.

I believe it is necessary to reintroduce timing predictability as a first-class property of embedded processor architectures. Architectures currently strive for superior average-case performance that regrettably ignores predictability and repeatability of timing properties. "Correct" execution of a C program has nothing to do with how long it takes to perform any particular action. C says nothing about timing, so timing is not considered part of correctness. Architectures have developed deep pipelines with speculative execution and dynamic dispatch. Memory architectures have developed multi-level caches and TLBs. The performance criterion is simple: faster (on average) is better.

The biggest consequences have been in embedded computing. Avionics offers an extreme example: in "fly by wire" aircraft, where software interprets pilot commands and transports them to actuators through networks, certification of the software is extremely expensive. Regrettably, it is not the software that is certified but the entire system. If a manufacturer expects to produce a plane for 50 years, it needs a 50-year stockpile of fly-by-wire components that are all made from the same mask set on the same production line. Even a slight change or "improvement" might affect timing and require the software to be re-certified. All users of embedded software face less extreme versions of this problem. Upgrading an engine controller in a car to a newer microprocessor, for example, often requires substantial redesign of the software and thorough retesting. Even "bug fixes" in the software can be extremely risky, since they can change timing behavior.

Designers have traditionally covered these failures by finding worst case execution time (WCET) bounds and using real-time operating systems (RTOS's). But these require substantial margins for reliability, and ultimately reliability is (weakly) determined by bench testing of the complete implementation. Moreover, WCET has become an increasingly problematic fiction as processor architectures develop ever more elaborate techniques for dealing stochastically with deep pipelines, memory hierarchy, and parallelism.

The reader may object that there are no true "guarantees" in life, so the correct solution should be to accept timing variability and to build in robustness. However, synchronous digital hardware—the technology on which most computers are built—can deliver astonishingly precise timing behavior with reliability that is unprecedented in any other human-engineered mechanism. Software abstractions, however, discard several orders of magnitude of precision. Compare the nanosecond-scale precision with which hardware can raise an interrupt request to the millisecond-level precision with which software threads can respond.

To fully exploit such timing predictability would require a significant redesign of much of computing technology, including operating systems, programming languages, compilers, and networks. I believe we must start by creating a new generation of processors whose temporal behavior is as easily controlled as their logical function. We call them precision timed (PRET) machines [1]. Our basic argument is that real-time systems, in which temporal behavior is as important as logical function, are an important and growing application; processor architecture needs to follow suit.

Of course, timing precision is easy to achieve if you are willing to forgo performance; the engineering challenge in PRET machines is to deliver both precision and performance. In [1], we argue that the problem should be first tackled from the hardware design perspective, developing precision timed (PRET) machines as soft cores on FPGAs. The near term goal would be that software on PRET machines be integrated with what would traditionally have been purely hardware designs. This provides a starting point for a decades-long revolution that will make timing predictability an essential feature of computing.

References

[1] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference (DAC)*, San Diego, CA, 2007.

Gernot Heiser

NICTA and University of New South Wales and Open Kernel Labs, Sydney, Australia
gernot@nicta.com.au

One of the greatest challenges facing modern embedded systems results from the complexity of their software. Convergence is leading to dramatic increase of functionality offered by a single device, which results in a continued growth of embedded software. Consumer devices, such as mobile phones, already run software that measures upwards of 5,000,000 lines of code.

This complexity is impacting the reliability, safety and security of embedded devices. Even well-engineered code is expected to contain in the order of one bug per thousand lines of code (and most embedded software is not that well engineered), and the defect density in many cases grows with the size of the overall code base. For the foreseeable future, embedded systems will therefore have to live with thousands or tens of thousands of bugs.

At the same time, embedded systems are increasingly used in contexts where they are security critical (e.g. financial transactions via mobile phones), safety critical (e.g. x-by-wire) or mission critical (e.g. business communication infrastructure). This means that the cost of failure of embedded devices is becoming unbearably high. In other words, the effects of the inevitable software faults must be contained, so that the device can recover from them, and continue operating correctly (although possibly with temporarily impaired functionality).

While this is primarily a software-engineering challenge, we argue that it cannot be met without advanced operating-systems technology.

The modern way of managing software complexity is by componentisation. A software system is de-composed into small units which communicate via small interfaces. If those interfaces can be *guaranteed*, meaning that we can ensure that no inter-component interaction happens except via a declared interface, the implementation details are hidden inside the component. More importantly, the effects of a (faulty) component misbehaving can be limited — for example, the component cannot damage any data other than its own. Alternatively, if a component is small enough, it becomes possible to *prove* the correctness of its implementation.

Component interfaces can be guaranteed either by language means or by hardware. The former requires that all components are implemented in a type-safe language. This is, in general, unfeasible for embedded systems, which typically contain large amounts of code written in non-typesafe languages such as C, C++ and assembler. Furthermore, the run-time systems of type-safe languages are themselves large and must be expected to contain many bugs, which undermines the language approach to componentisation.

The alternative, hardware-enforced component boundaries, works for any language, but imposes strong requirements on the operating system that manages the hardware mechanisms. Specifically, we need:

1. an extremely reliable operating-system substrate, as the encapsulation of components can be only as reliable as the OS itself;
2. a lightweight yet highly flexible component technology that supports the construction of large and complex, yet well-performing software systems that possess the necessary fault-tolerance properties;
3. the ability to ensure non-functional properties of the system, such as timeliness and energy budgets.

The first requirement implies that the operating system is based on a minimal kernel that is small enough to make a formal correctness proof tractable. Such a *microkernel* must be general enough to support the construction of arbitrary systems on top, and must be able to support arbitrary resource management and security policies, while providing low-overhead cross-component communication.

The second requirement is for a carefully-designed software-engineering framework that leverages the operating-system mechanisms to support highly-reliable systems that can detect and recover internal faults, and support formal reasoning about its operation at the system level. In particular, it should be possible to prove correctness of components individually, and prove properties of the system based on formal models of the components.

The third requirements implies that the operating system is fully analysed for its worst-case time and energy consumption, and that it is possible to analyse the timing and energy use of the componentised software system.

The above set of requirements provide a formidable research challenge. Meeting this challenge requires a concerted cross-disciplinary approach combining operating systems, software-engineering and formal methods research[1].

References

[1] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(3), July 2007.

Joseph Sifakis

Verimag, 2 Avenue de Vignate, centre Equation, GIERES, France
`Joseph.Sifakis@imag.fr`

Embedded software design is part of embedded system design, which by its very nature, requires a deep and coherent integration of competencies in software, hardware, and controller design. The scientific challenge is in setting up embedded systems as a new discipline, which systematically and even-handedly marries computation and physicality, performance and robustness. Our aim is not to discuss this grand challenge presented in detail in [1], but rather to identify missing pieces for applying the *formal methods* paradigm to embedded systems design. Formal methods, in particular *formal verification*, have been successfully applied to hardware design, and more recently, to software design. To what extent is it possible to adapt existing methods and tools to embedded systems?

Design is the process of deriving a system that meets given requirements. Correctness can be demonstrated using formal models meeting the requirements and representing a design at some level of abstraction. For some classes of systems, it is possible to derive a design from a model which *by-construction* meets the requirements (e.g. hardware synthesis). For others, a design is obtained as the result of a a creative process using existing algorithms and principles for organizing computation, pre-defined functions, data, components, etc. In this case, correctness may be established by *a posteriori* verification, to show that a model, which is an adequate abstraction of the design, meets the given requirements.

There are two non-trivial obstacles to transposing the formal methods paradigm to embedded systems.

Faithful modeling: Contrary to pure software or hardware, for a given embedded system, we do not know how to derive a model that faithfully represents its behavior at the proper level of abstraction. Embedded systems are *heterogeneous*. They are composed of a large variety of components, each having different characteristics, from a large variety of viewpoints, each highlighting different dimensions of a system.

We need models representing systems at varying degrees of detail and interrelated in an abstraction hierarchy. A key abstraction would relate application software to its implementation on a given platform. Another cause of heterogeneity in abstractions, is the use of different viewpoints e.g. combining computational and analytical models or different extra-functional dimensions such as timing and power consumption. At each level of abstraction, two main sources of heterogeneity may exist. a) Components may be fully synchronized, or asynchronous. Currently, we do not know how to consistently integrate synchronous and asynchronous models. b) There is a large variety of dispersed mechanisms used for coordinating interaction between components, including semaphores, monitors, message passing, remote call, protocols etc.

We need a *unified composition framework* for heterogeneous components. Such a framework should allow system designers to formulate their solutions in terms of tangible, well-founded and organized concepts. It should consider architectures as first-class entities, having their own properties, that can be studied independently of their components' behavior.

Achieving correctness: Current verification techniques focus mainly on invariance properties by analysis of abstractions. Embedded systems are concurrent, and their correctness is characterized by other classes of properties not preserved by abstractions. This is typically the case for progress properties or time-dependent properties. For instance, we do not have abstractions preserving even simple progress properties such as deadlock-freedom.

Furthermore, existing verification techniques work on flattened global models, whereas for embedded systems, extra-functional properties depend on architectural features. For all these reasons, we believe that *a posteriori* verification by itself is not sufficient for achieving correctness. As discussed in [1], emphasis should be put on results allowing *correctness-by-construction* in two complementary directions: a) Develop *reference architectures* guaranteeing generic properties *by-construction* such as security, robustness, diagnosability, adaptivity. b) Develop results allowing interference-free composition of different architectural solutions. Such results are essential for guaranteeing the stability of properties of integrated components, and are necessary for building reconfigurable systems.

References

[1] T. A. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods 06, LNCS 4085*, pages 1–15, 2006.

Jaejin Lee

School of Computer Science and Engineering, Seoul National University, Seoul, 151-744, Korea
jlee@cse.snu.ac.kr

In the last few years, we have seen heightened interest in the field of mobile devices and consumer electronics equipments, such as smartphones, PDAs, high definition TVs, gaming stations, set-top boxes, etc. Moreover, we have seen landmark developments to strengthen and broaden the scope of the embedded systems to those areas.

One difficulty in developing embedded software is that the software development environment is largely different from that of the enterprise and desktop computing software. In the enterprise and desktop computing, software developers work on the existing hardware/software platforms, for which their product will be finally released. However, in the case of embedded software, the hardware platform is often still under development in parallel with the embedded software due to the pressure of speeding time to market.

On the other hand, demands for computing power and power efficiency in embedded applications have been continuously increased. For example, current video workloads in mobile devices and consumer electronics equipments require much more computing power and power efficiency than those of a few years ago.

Since the performance and power efficiency of the conventional high performance embedded processors may not easily satisfy those demands, multi/many-core embedded processors soon will replace the conventional embedded processors. The number of cores in a single chip is expected to grow exponentially, according to the Moore's law. Furthermore, the multicore embedded processor architecture will show up in all sorts of forms including architectures with symmetric multicores, distributed memory multicores, heterogeneous multicores, etc. with various interconnect architectures and memory hierarchies. For example, the Cell Broadband Engine is one of them.

While the multicore approach may satisfy those demands, its software development cycle becomes much more complex than before. The muticore architecture forces the software developers to write parallel programs explicitly. We arguably believe that the degrees of difficulty in programming and debugging are almost equal in enterprise and desktop computing. However, this is not true for multicore embedded systems because the underlying complexities in the parallel execution of the software are exposed to the programmer during debugging period. Moreover, performance debugging plays a key role in this case. Consequently, embedded software development becomes even more difficult than before.

Among others, to facilitate the development of software for embedded multicores, we address two challenging issues associated with *the ease of programming and debugging* in this position statement. One challenge is *providing embedded software developers with an easy and practical programming model for embedded multicores.* Currently, most of the software implementations for multicore embedded systems (typically with heterogeneous dual cores: RISC and DSP) are based on the traditional programming model, in which a control processor and a compute coprocessor communicates with each other through some specialized communication channels or shared memory. This may not work for the near future multicore architectures where cores are more tightly coupled and interconnect architectures are more complicated in order to meet the performance demand.

The solutions to this challenge could be derived in the form of automatic parallelization based on sequential semantics (possibly with some user hints), a software run-time environment that hides the underlying architecture details yet without loosing performance, a new (possibly parallel) programming language and libraries, or a hybrid of them. For example, a software distributed shared memory for the Cell Broadband Engine is one of the software run-time environments that hide complicated cache coherence and memory consistency issues from the programmer.

The other challenge is *building a fast virtual prototyping system for multicore embedded software.* In the last few years, embedded system developers have exploited instruction level simulators to prototype and debug embedded software in addition to using them to explore design spaces.

A successful virtual prototyping system for multicore embedded software requires full-fledged whole system simulation technologies. It should be capable of performing cycle-accurate simulations of the multicore architecture, operating system, and peripherals to facilitate performance debugging. It should be fast enough to reduce time spent on each edit-test-debug cycle in the embedded software development process. Furthermore, it should be capable of detecting and reporting bugs caused by parallel execution and synchronization. We do not believe that such capabilities come in handy in the currently existing virtual prototyping systems for embedded software.

The solutions to the two challenges addressed in this statement will synergistically facilitate the software development process for embedded multicores.

Gilbert Edelin

Research Group on Information Science and Technologies, THALES Research and Technologies,Palaiseau, France
`Gilbert.Edelin@thalesgroup.com`

Embedded applications are extremely diverse: from information systems to aerospace on board navigation systems, or systems on chips. In addition to common scientific challenges [1], this raises a lot of practical challenges in defining the research directions, in selecting and exploiting the best of class solutions. At the Group level one can mention eg

- Applicability of new software Engineering and architecting solutions to a specific domain: specific properties (Critical real-time constraints, performance, dependability, etc) suitability/efficiency of standard solutions: theoretic issues (languages, semantic gaps, computational models, composability,etc), cost effectiveness.
- Addressing legacy to preserve the software investment while implementing new generations of tools
- Validate the maturity of the technologies,
- Develop training and cultural changes: embedded software is a new discipline which should result in important improvements in computer science curricula
- Measure the ROI (return on investment) before ensuring a large scale deployment

Key trends in hard real-time and highly constrained Systems from a THALES perspective have been recently summarized this way [2] in the Aerospace field: growing automation, increased real-time media/interaction demanding Quality of Service (video, positioning, etc.), increasing need to process more info while preserving response time, continued limited resources (communications, processing, etc.), increased cooperation between systems, evolution during life cycle, extensive safety constraints, increased vulnerability (due to complexity and openness) along with a growing request for security etc.

This means systems harder and harder to engineer and design to manage response quality in an uncertain world: the platforms themselves, when only inspired by IT solutions (servers), are hugely increasing the difficulty. Moreover these systems must sustain lifecycles as long as 50 years or more [3].

In the context of these types of Embedded Systems, from a user perspective, essential examples of challenges appear to be found in eg:

- Combining functional and non functional properties: a standard top down design flow is not suitable here but a Y chart coupling software engineering and hardware engineering techniques and giving to the programmer a exploitable view of the complexity at the execution level should give better results
- For ensuring long lifecycle stability of the application, providing a stable reference model completed by formally proved transformation techniques: software synthesis, co-design, use of higher reference levels than source code, libraries are likely to be improved. Engineering for uncertainty to find the best compromise/ optimum should result in weaving and reconciling all models in a proven solution
- Breakthroughs in performance and timing predictability are also expected: they could come from new composition techniques for non functional properties
- New platforms for computational intensive applications: parallel, with a new programming model, less RT uncertainties and more robustness to failures, with their related programming tools and formalisms (beyond C) should take part to the solution.
- Breakthroughs in productivity with model driven techniques providing quality and composability (even for non functional properties), but also analysis and simulation tools which are essential to the massive extension of these component oriented techniques in the engineering of Embedded Systems. This means also a dramatic reduction in verification and certification lead-times. This covers bugs and moreover timing properties correctness. Incremental certification is also a track for improvement.

The goal of a large Company involved mainly in mission critical systems is not specifically to develop tools but we need a diversified offer for the thousands of software designers and developers. But with an industry of editors very fragmented and unstable [3], the industrial challenge of providing state of the art, stable, low cost and sustainable solutions is high, so internal development and investment in research is necessary. This is also required to handle the difficult and long lasting insertion between best in class scientifically proved solutions and their suitability in the industrial context: the operational transition (scale factor, legacy issues, curricula of the software engineers, training, etc. are huge obstacles to the rapid diffusion in large industrial companies.

References

[1] T. A. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Formal Methods 06, LNCS 4085*, pages 1–15, 2006.

[2] J.L. Voirin. Some future challenges in hard real-time and highly constrained systems. *Artemis annual conference*, Berlin, 4-5 June 2007.

[3] S. Robert. New trends and needs for avionics systems. *Artemis annual conference*, Berlin, 4-5 June 2007.