Balance between Formal and Informal Methods, Engineering and Artistry, Evolution and Rebuild

Edward A. Lee, Professor, UC Berkeley, eal@eecs.berkeley.edu

Technical Memorandum UCB/ERL M04/19

July 4, 2004

Abstract

This paper is the result of a workshop entitled "Software Reliability for FCS" that was organized by the Army Research Office, held on May 18-19, 2004, and hosted by: Institute for Software Integrated Systems (ISIS), Vanderbilt University. I was given the charge of leading one of four topic areas, and was assigned the title. This is my summary of the results of the workshop on this topic.

It may well be that established approaches to software engineering will not be sufficient to avert a software disaster in FCS and similarly ambitious, software-intensive efforts. This topic examines the tension between informal methods, particularly those that focus on the human, creative process of software engineering and the management of that process, and formal methods, specifically those that rely on mathematically rooted systems theories and semantic frameworks. It is arguable that, as these approaches are construed today by their respective (largely disjoint) research communities, neither offers much hope of delivering reliable FCS software. Although certainly these communities have something to offer, the difficulties may be more deeply rooted than either approach can address. In this workshop, we took an aggressive stand that there are problems in software that are intrinsically unsolvable with today's software technology. This stand asserts that no amount of process will fix the problems because the problems are not with the process, and that today's formal techniques cannot solve the problem as long as they remain focused on formalizing today's software technologies. A sea change in the underlying software technology could lead to more effective informal *and* formal methods. What form could that take?

1. Some Objectives of Formal Methods

An early conclusion in the workshop was the dispelling a widely held misconception that formal methods have had little practical impact in software. Type systems are an example of a formal method that is a centerpiece of all modern programming languages. They have a formal structure that has influenced the design of languages and compilers and has proven scalable to extremely large programs. They contribute enormously to software reliability and to the efficiency of the software design process by exposing many programming errors early in the design process.

But type systems represent only the static structure of programs. They do not represent temporal or concurrent behavior, for example. Could the equivalent of strongly typed interfaces be developed to represent these other aspects? While there is research in this direction, it appears inconclusive at this time.

Nonetheless, many formal methods demand a level of skill levels not normally found in software development community to apply.

An examination reveals that formal methods have several objectives, and that depending on the emphasis, the approaches may differ. In particular, formal methods have been proposed to provide:

- semantic grounding for languages,
- precise specification,
- proof of properties,
- proof of correctness,
- improved understanding, and
- reduced need for testing.

It was argued in this workshop that of these, "proof of correctness" was probably the only unattainable objective.

2. Programming Languages

Languages form the medium of expression for software design. In practice, most embedded software is written in C, an ironic choice because of its complete lack of concurrent or temporal semantics. Concurrency and time are essential aspects of software that engages with sensors and actuators. What C does provide is excellent efficiency, access to hardware resources, and familiarity to programmers.

Can new languages help with embedded systems? An interesting case study is the SCADE system marketed by Esterel Technologies. This system is based on the synchronous language Lustre, the formal properties of which strongly influenced the process that led to the certification of the compiler for use in safety critical avionics software. This system is used in practice by Airbus and others for embedded software design.

Another interesting case study is Simulink, from The MathWorks. Simulink has taken hold in several communities, perhaps most notably in the automotive industry where it is widely used to design and deploy embedded control software.

An issue that arises is that the introduction of new programming languages is difficult, expensive, and risky. Even with a strong mandate for many years from DOD, Ada, which has many desirable features for embedded software, has never been completely embraced by the embedded software community. A focus on domain-specific languages and on languages with visual syntaxes (SCADE and Simulink fit both) helps languages gain acceptance, because domain knowledge and style can be built into the languages, and visual syntaxes meet less resistance, presumably because the learning curve appears gentler (although in practice, it may be just as steep).

Yet the success of Simulink and SCADE is the exception, not the rule. Simulink succeeds in part because it is not recognized by engineers as a "language." It is, first and foremost, a modeling tool. It just happens to be extremely convenient that models can be compiled ("code generated") into deployable code. Whereas modeling has traditionally been used as part of the requirements definition process, in this case the requirements turn out to be a compilable implementation. The distinction between "model" and "program" disappears.

Neither Simulink nor SCADE emerged from the mainstream programming languages research community. It was argued in the workshop that language research is stalled in part because language researchers tend to promote "universal" solutions, languages that completely replace their predecessors. Simulink most notably does not do this; it fully embraces C as a mechanism for defining primitive components and as a target for code generation, and therefore offers the key advantages of C, access to hardware resources and code efficiency, but offers them within a framework that has a clean semantic notion of time and concurrency. Simulink also leverages the task scheduling provided by real-time operating systems (RTOS's), but does not expose to the designer the features that are difficult to use correctly, such as priorities. Priorities are used by the code generator (with preemptive multitasking) to synthesize a correct implementation of the Simulink semantics, but what the designer works with is the Simulink semantics, not the abstraction of processes with priorities that RTOS's depend on.

3. Platforms

Describing Simulink as a programming language is a stretch, since the role it plays in design differs somewhat from the role that languages have traditionally played. A better conceptual framework in which to consider design alternatives is to leverage the notion of "platforms." A platform is a set of designs. A programming language (e.g. Java) is a platform (the set of all Java programs). The set is described by describing the syntax and some of the semantics of Java, which defines what it means to be a member of this set. Java byte code is a platform. The Intel x86 architecture is a platform (the set of all x86 programs). A compiler or an interpreter is a translator from a design in one set to a design in another.

Simulink is a platform (the set of all Simulink diagrams). A code generator is a translator that converts a member of this set into a member of the set of C programs. If we change the question from "what programming language(s) should we use?" to "what platform(s) should we use?" then we are likely to get much better answers because we haven't prejudiced the answer with preconceptions about what constitutes a "language" (e.g., it has to have a syntax that can be given in BNF). Moreover, platforms can work in concert at different levels of abstraction (e.g. Simulink with C).

4. Actor Orientation

It was argued in the workshop that concurrency and time play central roles in embedded software, and yet are almost entirely absent in the semantics of prevailing programming abstractions. When present, as in the threading model of Java, they are reflections of very old and very low-level mechanisms. Java's threads and monitors date back to the 1960's, and as a concurrency model, are actually extremely difficult to use reliably.

Many flaws in software are ultimately due to concurrency errors, and these flaws are difficult to find. They manifest themselves rarely in an execution, so verification based on testing often fails to find them. Code can be exercised in deployed form for years before a design flaw appears. Static analysis techniques can help (e.g. Sun Microsystems' LockLint), but these methods are often thwarted by conservative approximations and/or false positives.

Worse, programs that use threads and monitors can be extremely difficult for programmers to understand. It was argued at the workshop that if a program is incomprehensible, then no amount of process improvement or schedule extensions will make it reliable. In fact, schedule extensions

are as likely to degrade the reliability of programs that are difficult to understand as they are to improve it.

Formal methods can help detect flaws, and in the process can improve the understanding that a designer has of the behavior of a complex program. But if the basic mechanisms fundamentally lead to programs that are difficult to understand, then these improvements will fall short of delivering reliable software.

Simulink and SCADE both offer concurrency models that are much easier to understand than threads or processes that interact via monitors and semaphores. Both are based on a synchronous abstraction, where components conceptually execute simultaneously, aligned with one or more interlocked clocks. SCADE relies on an abstraction where components appears to execute instantaneously, whereas Simulink is more explicit about the passage of time and supports definition of tasks that take time to execute and execute concurrently with other tasks. In both cases, every (correctly) compiled version of the program will execute identically, in that if it is given the same inputs, it will produce the same outputs. In particular, the execution does not depend on extraneous factors such as processor speed. Even this modest objective is often hard to achieve using threads and monitors directly.

Simulink and SCADE both offer a software component model that is significantly different from the object-oriented component model. Whereas in Java and C++ components interact with one another primarily through method calls, in Simulink and SCADE they are concurrent components that send messages via ports. This style of component interaction has been called *actor oriented*, and it can complement and co-exist with object-oriented components. The key feature of actor-oriented models is that they emphasize concurrency, and typically offer concurrency mechanisms that are easier to understand than threads.

Many actor-oriented languages also offer a notion of time built-in to their semantics. Imperative languages (C, C++, Java) abstract away the notion of time, and temporal properties have to be specified indirectly by invoking operating system features (such as setting priorities). Simulink models, for example, explicitly specify temporal behavior, and any (correct) implementation of the Simulink model must conform to that specification.

There is much discussion of integrating “non functional” and “quality of service” aspects into program specifications. Time is a key one of these aspects. However, much of this work approaches the problem by adding expressiveness through APIs to object-oriented languages. It was proposed at the workshop that together with adding expressiveness, it is also necessary to create constraints. A clean semantics for time and concurrency cannot emerge simply as a design pattern in languages that fundamentally lack time or concurrency in their semantics.

Actor-oriented modeling is an active, albeit somewhat immature, area of research. Computer Science, as a discipline, has had only modest and sporadic interest in domain-specific languages, and most actor-oriented languages in use today are domain specific. Investment in research in this area (such as the DARPA MoBIES program) can strongly affect the level of activity in the research community.

It was argued at the workshop that actor-oriented design has the potential for impact on the scale that object-oriented has had. But much more work is needed, for example in modularity

techniques (classes, inheritance, interfaces, type systems, aspects), models of computation, and visual notations.

5. Model Transformations and Multi-View Modeling

Modeling has always played a role in software design, but it has its pitfalls. Models can diverge from an implementation over time, and they are frequently at a higher level of abstraction. Introducing details later can introduce bugs and complexity. Moreover, models can be incorrect and code synthesizers can be incorrect. Interaction with legacy or handwritten code can introduce errors.

These problems are mitigated (but not eliminated) by blurring the distinction between the model and the program. A Simulink model, for example, is both an (abstracted) model of a control system and the source code for the embedded software.

With the distinction between models and programs becoming blurred, it becomes useful to have multiple models/programs for the same design. This fact is well recognized in the object-oriented programming community, where the various UML languages are used in concert to complement source code specification and to describe, for example, static structure and sequential behavior. Actor-oriented technique could similarly benefit from multi-view modeling. Research is needed however in how to maintain coherence and consistency, how to integrate code generators, how to weave specifications of diverse aspects, and how to leverage descriptions of the modeling paradigms themselves (so called “meta modeling”).

6. Visual Notations

Visual notations have a checkered history in computer science, but have always played a role in design. In mainstream design today, the various UML visual languages are often extensively used to complement textual programs and specifications. Highly concurrent models seem to benefit particularly from visual notations (e.g., Simulink). But there are questions about scaling and about expressiveness that need to be addressed. For example, is the prevalent use of hierarchy as the principle (or only) abstraction mechanism sufficient?

Visual notations should be used to express aspects of design that are not well expressed by text, such as static structure and concurrency. They should not be used to replace text where text does well, as in flowcharts, and certain elements of executable UML.

7. Conclusion

Bad design can be done in any language. No amount of formal analysis will turn a bad design into a good design. No amount of schedule slippage or process planning will turn an incomprehensible design into a reliable one. Bad designs evolve into worse designs, never into good designs.

Artistry is the essence of good design. Languages are the medium of expression, and hence greatly affect the product. High quality medium is essential for durable art. Tools (formal and informal) are just tools, and high quality tools facilitate but do not guarantee artistry.