AVERTING DENVER AIRPORTS ON A CHIP

Software Engineering Research for Space Applications of MEMS
(position paper)

Kevin J. Sullivan
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

sullivan@virginia.EDU
(804)982-2216

71098

INTRODUCTION

Recent years have seen important advances in our software engineering capabilities. As a result we are now in a more stable environment than in the recent past. *De facto* hardware and software standards are emerging. Work on software architectures and design patterns [GS95,GHJV95] signals a consensus on the importance of early system-level design decisions, and agreement on the uses of certain paradigmatic software structures. We recognize the non-existence of simple solutions to complex software development problems. We now routinely build systems that would have been risky or infeasible a few years ago.

Unfortunately, technological developments threaten to destabilize software design again. Systems designed around novel computing and peripheral devices will spark ambitious new projects that will stress current software design and engineering capabilities. Micro-electro mechanical systems (MEMS) and related technologies provide the physical basis for new systems with the potential to produce this kind of destabilizing effect. Software will, with high probability, be the "pacing," high-risk item for many such systems.

One important response to anticipated software engineering and design difficulties is carefully directed engineering-scientific research. Although no single advance in software engineering will suffice for fast cycle time production of dependable software for future systems, the coordinated application of results from several lines of research should be of substantial value. Recent research has identified promising lines of attack on key problems. Two specific problems meriting substantial research attention are the following:

- We still lack sufficient means to build software systems by generating, extending, specializing, and integrating *large-scale reusable components*.
- We lack adequate computational and analytic tools to extend and aid engineers in maintaining *intellectual control* over complex software designs.

DISCLAIMER

In this paper I elaborate on the claim that advances in MEMS and related technologies will destabilize software design. I offer practical advice on how to deal with the problem. And I discuss research at the University of Virginia that is relevant to the above problem formulations.

My research does not address MEMS *per se*. I am not expert in the area. The positions in this paper are informed opinion. It is impossible even for experts to predict the future impacts of given technological developments, and it is far too easy to be way too optimistic or pessimistic. This position paper is thus an exploration of ideas, not a wager on future outcomes.

While not an expert, I do work in a context in which MEMS is an emerging issue. My research falls within the scope of an *end-to-end systems* research program joint between the electrical engineering and computer science departments. In the next phase of this program, we will target MEMS applications. Work to date has been at the device level: We are designing a sensor to detect chlorine ions to help address the problem of corrosion of the rebar in concrete bridges. The devices will be placed in the concrete. When readings exceed a threshold, electricity will be applied to drive off the chlorine.

RESEARCH OVERVIEW

As this simple device and related devices begin to mature, design and implementation of software to manage them will become a more important issue. As I discuss below, rapid advances in devices will require new work in software engineering. At present, proposed devices are quite limited. The devices we are proposing have a sensor, simple CPU, small memory, and small antenna: hardly a challenge to the best software engineers. These simple devices will be good for initial, small-scale software engineering research for MEMS.

As we scale up the complexity of systems based on new technologies, it will be critical to address the difficult software engineering issues already holding us back in the most demanding current efforts (e.g., advanced, space-based telecommunications systems). We will shortly require major improvements in both software development throughput (productivity), latency (cycle time in response to new requirements), and dependability.

My current work comprises a number of promising attacks on these problems. My primary focus has been on the integration of *independent components* into systems [Sul92,Sul94]. Recent variants focus on integration of *large-scale components*, including, *commercial off-the-shelf components* [Sul95a]. Productivity and dependability demand the reuse of certified, large-scale components; dependability and flexibility demand advanced integration techniques. In the tools area, I focus on the need for rapid development of custom, high-confidence *computational and analytic tools*. Such tools are needed to aid engineers in obtaining and maintaining intellectual control over complex software systems [Sul95b]. Advances in the two broad areas of component-based development and tools promise to significantly benefit future system development projects.

TECHNOLOGICAL DISCONTINUITIES DEMAND RESEARCH

Technological discontinuities demand engineering research. If the technological situation changes by an order of magnitude, you have a whole new set of research problems [Wulf95]. Old solutions to known problems have to be rethought. New problems emerge. Yet, while order-of-magnitude change is costly, it also presents enormous opportunities.

Petroski presents the pencil as a paradigm of technological discontinuity [P89]. In 1793, unavailability of English graphite due to the onset of war forced Continental pencil manufacturers to engage in engineering research and development to find alternative ways of making pencil leads [P89]. The French Minister of War, Carnot, commissioned Nicolas-Jacques Conte to develop alternatives. Taking a "deliberately innovative" approach that united "the scientific method... with experience and with the tools and products of craftsmen," Conte developed the modern ceramic method for making pencil leads.

Two hundred years later, the computer revolution—a six-orders-of-magnitude improvement in computing machines over thirty years, produced in large part by "the transformation of computer manufacture from an assembly industry into a process industry [B95]"—drove the need for a significant new engineering research program. As Dijkstra noted, "as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem [D72]." Software design ambitions scale with device capabilities; but software design abilities do not!

The resulting "software crisis," was characterized by many costly engineering failures. As in 1793, so in 1968 the need was countered by an aggressive engineering research program: The NATO Science committee established software engineering as a discipline. Problems with large systems are still a serious concern [G94], but research has produced made much progress toward understanding software system design in the small and large.

DO MEMS BETOKEN A NEW DISCONTINUITY?

The question I ponder in which paper is this: Do MEMS devices betoken a similar technological discontinuity—one that will destabilize software design? Will ambitious future systems based on MEMS and related technologies exceed our software engineering capabilities? If so, what is the proper response?

A case for an affirmative answer includes several points. First, MEMS interact with the physical world, requiring complex real-time behaviors; should MEMS become common, they will raise the average complexity of programs to be designed, increasing demands for scarce good software designers. Second, MEMS do seem destined to become both more common and much more complex, for the same reasons that computers became common and complex: transformation of manufacture from an assembly industry to a process industry. Third, the ability to produce MEMS at low price and in high volumes will encourage the design of systems incorporating many complex devices. This—in the area of large, distributed systems—is where real software engineering difficulties will begin.

The title of this position paper suggests "Denver Airports on a Chip" as a metaphor for the difficulty of programming advanced future systems. As a paradigm of failure owing to software difficulties, one can do little better than the Denver Airport baggage system. That system provides a complexity baseline for assessing potential future software difficulties.

The key problem at Denver was the difficulty of programming a "central nervous system of some 100 computers networked to one another and to 5,000 electric eyes, 400 radio receivers and 56 bar code scanners [G94]." The ambitiousness of the concept outstripped the software engineering capabilities of the developers. My point is *not* to criticize the developers but to pose the question, how do applications now being envisioned compare to Denver; can we use a comparison of physical characteristics (numbers and kinds of devices and interconnects) as a rough way to gauge potential difficulties?

At least some of the applications now being discussed appear to match or surpass "Denver" in complexity. A great deal of the complexity of ambitious future systems will be in the software; and if miniaturization succeeds, we're going to have many more such systems. As a case study in failure due to software engineering difficulties, Denver Airport is invaluable to future system designers: The *key* is to avert software engineering failures.

SOFTWARE ENGINEERING RESEARCH FOR MEMS

How can the risk of software engineering failure be managed? The most important point is to recognize that software engineering difficulties are *a* top risk, and probably *the* top risk, facing advanced technology projects. These risks must be managed aggressively.

A two-pronged approach is needed. First, use best current practices. Take an iterative approach to risk management in which you continually reevaluate risks and address the most serious ones first [Boehm76]. Understand that good management is essential. Hire great software designers [B95]. Recognize there is no silver bullet: No single technique or advance will radically simplify the software problem. Second, understand that we still haven't resolved certain key foundational software engineering research issues (e.g., how to build systems from large-scale reusable components). Progress in these research areas (properly employed) will contribute significantly to success.

Fortunately, in my view, past software engineering research has laid the foundations for new syntheses that will help us to meet the demands for software for future systems with increasing confidence. In the rest of this paper, I discuss my research in two areas. The first attack—building software systems by integrating independent, large-scale components—targets the problem of *throughput, latency, cost in development, and dependability in the resulting product.* The second attack—rapid development of high-confidence analytic and computational software engineering tools—targets the need to help engineers to extend and maintain *intellectual control* over complex software designs.

Component-based software development

Building software by integrating independent components is critical for at least three reasons. First, it achieves a separation of concerns essential for both intellectual and managerial control of complex systems. Second, it amortizes development costs to the

extent that components are reused. Third, in some sense it permits one to meet the need for exponential demands for software by combining a small number of parts in different ways.

Many component-based approaches have been devised and used, e.g., procedures, pipes and filters, and objects. Structured programming does not support composition of *large-scale* components—e.g., commercial off-the-shelf (COTS) application-sized parts. Unix Pipes-and-filters are inadequate for *interactive* systems. Classical object-oriented approaches do not support very well the integration of visible, stand-alone objects [Sul94].

I explore relevant engineering issues in the context of a specific design approach, called the mediator method [Sul94]. In this method, requirements are mapped to an architecture called a behavioral entity-relationship (ER) model. The nodes in such a model represent independent, visible *behaviors;* the edges represent *behavioral relationships*—potentially complex ways in which the behavioral components are required to interact. The next step is to map the behavioral ER model onto a set of components—such as C++ objects or COTS applications—in a way that preserves the structure of the model. Components that implement behaviors are independent and visible; while separate components (called *mediators*) implement the behavioral relationships.

This approach embodies a view of both the static structure of an integrated system and the way that it evolves. In particular, this design approach is intended to provide an unusual degree of flexibility in composing and evolving integrated systems, overcoming a serious problem with common design methods: that they throw integration and evolution into conflict. The architectural model accommodates evolution by the addition, change and deletion of behaviors and behavioral relationships: one adds, changes, or deletes nodes and edges in the behavioral ER model, with corresponding changes to the implementation. One integrates a new component into a system, for example, by adding the component as an independent part, then adding new mediators to make it work with the existing parts. The existing parts don't have to change to work with the new one. The mediators take care of integration separately from the parts being integrated.

Results to date are encouraging. We have used the approach to develope a radiation treatment planning system for cancer patients [Sul95c] now in clinical use at several large research hospitals. The components—Common Lisp objects—model anatomy, radiation fields, graphical views, etc. The mediators integrate the component so that, for example, a change to a view results in a corresponding change the anatomical model. The approach was key to producing Prism on a modest budget—about eight person years [Sul95c].

I am now exploring the mediator integration of COTS components in a case study involving the design of a commercial-grade fault-tree analysis tool supporting a novel analysis techniques developed by my colleague Joanne Dugan. Components include Visio for drawing, Microsoft Access for storing and generating reports on fault trees, and other large-scale components. The mediators are in Visual Basic. I developed a prototype in about a week, and am now building a complete system. Delegating the interface and bookkeeping functions to volume-priced components will permit us to deliver, for a cost of under $1000, a serious new computational tool in a rich package comprising multiple millions of lines of code. This work is shedding much light on component integration issues.

While the applicability of the detailed mediator work to MEMS applications is not clear, the basic insights and component-integration structures will certainly be valuable. I am continuing research on component integration, pursuing topics including the following:

- identification of useful components and component types,
- techniques for specifying and implementing large-scale reusable components, and for specifying and implementing interconnection structures to integrate them [2,3],
- a discipline of programming with large-scale components, i.e., techniques for mappings application concepts onto integrated sets of components [5], and for characterizing the engineering tradeoffs involved in making the mapping decisions often required by shortcomings in components and integration mechanisms, and
- techniques for integrating components in the context of the heterogenous hardware environments of future systems, at the applications and operating systems levels.

Computational and analytic software engineering tools

Intellectual control is the cornerstone of dependability. This was true in Denver, and it will be true for future systems. Engineers have for centuries used computational and analytic tools to extend their understanding of complex structures. In the absence of such tools, the best choice may be not to build at all. Stephenson decided to employ a tubular bridge instead of a suspension bridge to span the Menai straights because he lacked the tools needed to understand why suspension bridges had failed in the past. His tubular design was an environmental, economic, and aesthetic failure; but he showed excellent engineering judgement by not trying to build a critical system beyond his intellectual reach [Sull95b].

In the future we will increasingly be asked to handle systems that stretch or exceed our intellectual abilities. The need is acute for tools to extend the intellect to help us assure the dependability of future systems.

Current tools exhibit at least two problems. First, it is too difficult to develop custom tools to answer specific but often fairly simple questions about software. Second, it is hard to validly interpret the outputs of many analysis tools. The latter fact is not as appreciated as it ought to be.

In a recent empirical study [MNL95], Murphy, Notkin and Lan report on significant divergences in the outputs of tools that compute static call graphs from C programs. Not only are the outputs different, but there is little or no indication that they should be; the input-output specifications are not clearly spelled out; and it is hard to infer or deduce what the specifications are. In the absence of such information, it is hard for the engineer to have much confidence in decisions made on the basis of tool results. That is, the tools give an impression of intellectual control, but they don't give real intellectual control—dangerous.

Figure 1 depicts a simple, prototype framework for the rapid development of custom program analysis tools. The "probe" is a commercial compiler front end plus an abstract "visitor class" in C++. Specializing the visitor enables extraction of selected information from the front end-generated intermediate representation (as indicated in the Figure by specializations for extraction of the includes hierarchy, call graphs, and global variable uses). The front end, owned by the Edison Design Group [EDG95], can be configured to parse many important variants of C and C++, and is highly optimized.

I typically define visitor subclasses to format and output Prolog facts [CM94] about the program to be analyzed—e.g., whether there is a call from procedure P to Q. The output is piped to the European Community Research Centre's Common Logic Programming System [E95]. This Prolog system includes a transactional relational database useful for storing large databases of facts; and it supports logic programming-based inference. After massaging the data produced by the probe, Eclipse constructs a "dot" program, which it then pipes to the "dot" program. "Dot" is a sophisticated, programmable graph layout system developed at ATT. The framework supports rapid generation of simple, custom analysis and structure visualization tools.

Instantiating the framework with less than 100 lines of C++ code and a few lines of Prolog enabled me to implement a static call graph extraction program. In contrast to current programs (whose outputs differ substantially and whose precise input-output specifications are unclear), my program realizes the following simple specification:

- if there is an arc in the extracted call graph from a node P to a node Q then there is a call instruction in procedure P with target procedure Q, and that if that instruction is executed there will be a runtime call from P to Q, and
- if there is no arc in the graph from P to Q then there is no possibility of a runtime call from P to Q, unless P is annotated to indicate that P contains indirect calls through pointers, in which case further (possibly manual) investigation is needed.

The function is not complex, and indeed it is simpler than the function of more sophisticated tools that perform more complex semantic analysis to refine the call graph. But no matter how sophisticated, if the engineer doesn't really understand how to interpret the tool output, the sophistication is not very useful. The key, of course, is not this particular tool or its component parts, but in the ability to rapidly generate families of customized tools to answer a range of questions about industrial systems. As I take this relatively new research forward, I am particularly emphasizing the following objectives:

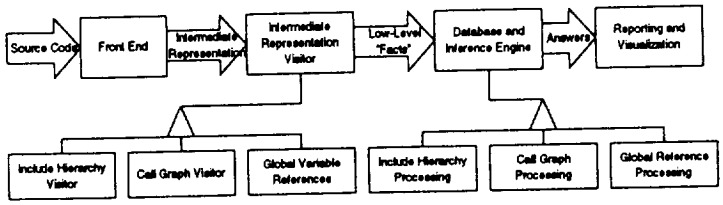

Fig. 1. - Architecture of the Tool Framework.

- To develop *probes* to gather signals from software systems (measurements, both static and dynamic), much as oscilloscope probes sample signals in electrical circuits. An example is a probe that extracts static call graphs from programs.
- To develop *analysis and presentation facilities* to prepare and display information derived from the captured signals. For example, I use a Prolog inference engine to draw conclusions from the raw outputs produced by probes.
- To develop architectural approaches that support rapid construction and operation of customized tools through *integration of large-scale reusable components*.
- To elaborate on the concepts of *evaluatability of tool results* as a key quality criterion, and of actual evaluation of tool results as a key responsibility of an engineer.

Again, the direct applicability of the specific work to MEMS-based applications is not clear. The point is that the computational and analytic tool situation for software engineering in general is woefully inadequate. The consequence is unnecessary limitation of our intellectual control over complex (or even just plain large) systems. Research and development in this area are important as we move into a domain of very ambitious projects.

CONCLUSION

Technology advances in both evolutionary and revolutionary ways. Most change is evolutionary, and the impacts of change are often less dramatic than predicted. Nevertheless, rapid advances can revolutionize the circumstances in which design is done, often requiring significant, new engineering research and development activities. The development of MEMS devices and related technologies seem likely to do this by sparking ambitious application concepts that will stress our software engineering capabilities. Best practices can help; but inadequacies in the foundations of software engineering demand that we also engage in aggressive software engineering research as a major part of our strategy to manage considerable software-related risks. The good news is that the basis for substantial progress has been laid by past research. The potential for technological change to catalyze major advances in software engineering is quite exciting.

The French had understood the risk of depletion of English graphite for many years before the war finally and quickly shut off the strategic material. Crisis finally prompted decisive research. We can anticipate and brace for the problems likely to be produced by new MEMS technologies, while ensuring that resources are not wasted on problems that don't materialize, with significant investments in software engineering research directed at relevant foundational issues. Mechanisms should be established to assure appropriation of the benefits of such research. Carefully directed investment in emerging MEMS-*specific* software engineering research issues is also warranted.

I have discussed my research in two areas: component-based software development (attacking cost, dependability, throughput and latency); and rapid construction of computational and analytic tools (attacking intellectual control). The technological change underlying this work—carefully developed, validated, and implemented, and thoughtfully combined with related research results—will help to avert "Denver Airports on a Chip."

ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation under grant number CCR-9502029. Some of the work described in this document is joint with John Knight and Joanne Dugan.

REFERENCES

[B76] Boehm, B., "A Spiral Model of Software Development and Enhancement," Computer, May, 1988, pp. 61-72.

[B95] Brooks, F., *The Mythical Man Month (Anniversary Edition),* (Reading, Massachusetts: Addison-Wesley), 1995.

[CM94] Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer Verlag, 1994.

[C90] Chen, Y.F. et al., "The C Information Abstraction System," *IEEE Transactions on Software Engineering,* SE-16(3):325-334, March 1990.

[D72] Dijkstra, E., *Programming Considered as a Human Activity,* in Classics of Software Engineering.

[E95] European Community Research Centre (ECRC), The Eclipse Common Logic Programming System, February, 1995.

[G94] Gibbs, W., "Software's Chronic Crisis," *Scientific American,* Sept., 1994.

[GS95] Garlan, D. and Shaw M., "An Introduction to Software Architecture," in V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering,* pp. 1-39, Singapore, 1993.

[GHJV95] Gamma, E. et al., *Design Patterns,* (Reading, Massachusetts: Addison-Wesley), 1995.

[MNL95] Murphy, G., Notkin, D., and Lan, E., "An Empirical Study of Static Call Graph Extractors," University of Washington Department of Computer Science and Engineering Technical Report UW-CSE-TR-95-08-01, August, 1995 (to appear in the Proceedings of ICSE-18, Berlin, March, 1996).

[P94] Petroski, H., *Design Paradigms,* (Cambridge: Cambridge University Press), 1994.

[P94] Petroski, H., *The Pencil: A History of Design and Circumstance*, (New York: Knopf), 1990.

[SN92] Sullivan, K.J., and D. Notkin, "Reconciling environment integration and software evolution", *ACM Trans. on Software Engineering and Methodology*, 1(3), July 1992.

[Sul94] Sullivan, K.J., "Mediators: Easing the design and evolution of integrated systems", Ph.D. dissertation and technical report CSE-TR 94-08-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1994.

[Sul95a] Sullivan, K.J. and Knight, J.C., "Assessment of an Architectural Approach to Large-Scale, Systematic Reuse," University of Virginia Department of Computer Science Technical Report, To Appear in Proceedings of ICSE18, Berlin, 1996.

[Sul95b] Sullivan, K.J., "Rapid Development of Simple, Customized Program Analysis

Tools," University of Virginia Department of Computer Science Technical Report CS-95-45, 1995 (submitted to the ICSE18 workshop on program comprehension).

[Sul95c] Sullivan, K.J., Kalet, I.J. and Notkin, D. "Mediators in a Radiation Treatment Planning Environment," University of Virginia Department of Computer Science Technical Report CS-95-29, 1995 (submitted to IEEE Transactions on Software Engineering).

[Wulf95] Wulf, W., Personal Communication, University of Virginia, 1995.