Thoughts on System Architecture Research & Education

Philip Koopman, *koopman@cmu.edu*
April 22, 2000

Our society has an insatiable appetite for complex systems, and such systems are increasingly becoming critical to our daily lives. Designers routinely attempt to implement systems of overly ambitious complexity. Sometimes such attempts fail, resulting in mere economic loss. Sometimes projects succeed (by luck and brute force), but yield short-lived systems that must be replaced quickly. And, in arguably the worst case, sometimes unfit systems are declared successes and adopted for use in critical applications even when it is inappropriate to do so. One does not have to look far to see examples of all three classes of outcomes today. In the future, the stakes will be raised dramatically because of the increasing connectedness of our world's people, economy and infrastructure.

Traditionally, the role of engineers has often been to build systems that are as ambitious as possible, get them mostly working, and then work on patches and extensions after the fact. True, many systems are limited enough in complexity that they can be built well most of the time. But, because of the thirst for ever-increasing complexity, a system architect often perceives these tractable systems as components (things that we know how to get right) rather than systems (things that we have to think about how to build).

Research and education in the face of the complexity escalation dilemma

There is a central paradox to research and education in system architecture. It is human nature to reach beyond our grasp. So, if we simply provide better tools and techniques for creating more complex systems, the eventual result will be that overly complex system designs will still be attempted, but at a higher level of ambition than before. This is an undesirable result, since attempting to improve the situation will simply have raised the stakes in the game without resolving any problems.

I see the role of educational institutions as finding and implementing a way past the dilemma of how to improve the abilities of system architects in the face of this paradox. A set of activities to support this are listed below (and I truly look forward to discussions on how to improve or augment these ideas!). A list of core competencies for system architects is provided in the next section.

Research. One way to resolve the system architecture research paradox is to create a dual-prong approach that combines *measurement* with improved *capabilities*. Suppose (and this is no small task) we can create quantitative ways to understand which designs are feasible for a given system architecture skill maturity level. Such metrics might take the form of an estimate for development time/cost that approaches infinity for a particular situation. We could then provide a powerful way for engineers to know that they are attempting something unwise, and a way to justify to management why it is infeasible. If we further create improved system architecting capabilities that are based on a core foundation of measurement, we will provide not only the incentive of demonstrable engineering process/product improvement, but also a set of checks and balances to reduce the incidence of abuse of these capabilities via escalation of system complexity. Is this an overly ambitious research agenda? Perhaps. But it does neatly resolve the dilemma while providing the satisfaction of improving the maturity of the system architecting process. (Of course nothing can prevent people from attempting the inadvisable, but at least with this approach there is little excuse for ignorance of the consequences.)

Practitioner education. Some aspects of system architecture can only be understood by experienced practitioners, and thus can only be taught some years after entry into the work force. It is essential to reach out to practitioners both to deliver research results as well as to maintain the researchers' foundation in reality. This sort of technology transfer can take place via consulting, summer schools, books, mainstream articles, and the like.

Graduate education. PhD-level education should, of course, be intimately tied with system architecture research. Graduate students are more mature than undergraduates, and have the luxury of focusing on narrow areas when appropriate. But there remains the problem that many of them will not have a broad enough world-view to really appreciate the breadth of system architecture concerns. While drawing upon students with experience is very helpful, it also dramatically limits the supply of students to consider for admission. A potentially viable strategy is to include a practicum requirement with carefully selected projects as part of a graduate program.

Professional MS degree. A professional (terminal) MS degree offers a unique opportunity to deploy technology while generating industry contacts and financial support. Professional MS degree programs are a proven way to create state-of-the-practice experts who can infuse new ideas into significant-sized organizations. An approach that has worked fairly well is that taken by the Carnegie Mellon Master of Software Engineering program, which attracts students to an intensive, one-year on-campus degree program with a significant practicum component. In many cases companies pay for students to attend, making it a revenue-generating program overall. (But, of course, careful attention must be paid to avoid such a program unduly sucking resources away from undergraduate education and core research activities, as well as to provide appropriate faculty incentives). The practicum component can not only help students, but also serve as a pilot activity for building long-term industry research relationships.

Undergraduate education. All engineering graduates of first-tier universities should be competent in system architecture skills. There are many systems of low to middling complexity that we should be able to get right, but don't. Often this is due to a lack of system architecture skills in typical engineers. Creating and supporting a way to teach these skills to undergraduate students might eliminate many unnecessary system design and operational failures. Such an approach also enriches the talent pool to feed other steps of the program, and once established creates a way to funnel mature research results into common practice.

System architect core competencies

In addition to appropriate domain skills, a good system architect must have six essential core competencies. (The contents of any such list are of course a matter of personal opinion – so this forms an inclusive starting point rather than a definitive ending point. In particular, Rechtin's work forms an excellent source of ideas in this area as well.) An architect:

1. Must have an appreciation for the full breadth of **complexity in the lifecycle** of a system. This can be learned via significant experience, or taught. Pure classroom teaching is probably insufficient to motivate students as to the importance of many factors without some sort of practicum or simulated experience (a combination of case studies and exercises involving real-world exposure). An architect must at a basic level be aware of the issues and know standard techniques for performing life cycle activities from requirement definition through system retirement. Additionally, at a basic level a system architect must have an appreciation for the strengths and limitations of different types of component technology used in systems, from mechanical components to software, including the influences of people as designers, operators, and maintainers. At an advanced level, an architect must be able to perform conscious tradeoffs with respect to the many process and product factors that affect a system.
2. Must have experience/knowledge of the strengths and weaknesses of **various architectural approaches** as well as ways to combine them. There are an astonishing number of architectural approaches, with at least one distinct approach per technical discipline (a controls or robotics engineer has a default architectural style much different from a hardware or software engineer – we have identified many such approaches to date). An architect must at a basic level know the various approaches and the situations in which they are most applicable, possibly by referring to a collection of appropriate architectural design patterns. At an advanced level, an architect must be able to blend

different architectural styles and be able to overlay multiple concurrent architectural views onto a single system.

3. Must have the ability to **cope with significant amounts of complexity** while creating or modifying an architecture. Ultimately, it is widely held that the best architectures are the product of a single mind. Thus, the limit of architectural complexity is bound by the human ability to grasp complexity. At a basic level, an architect must be proficient at using available tools and techniques to help deal with complexity (abstraction, interface design, CAD/synthesis tools, team structure). At an advanced level, an architect must be able to apply research results in creating improved system decompositions, exploit ways to improve the collective thought of small teams, and focus on essential complexity while discarding accidental complexity.
4. Must be able to **articulate the essence of a particular system architecture** to others. At a basic level, a system architect must be able to communicate the architecture and architectural principles for a particular system to a design team to ensure that architectural coherence is maintained. At an advanced level, system architects must be able to both market an architecture and provide personal leadership to the design team. Advanced architects must ensure that any up-front costs for a good architecture are justified, and that an appropriately skilled and motivated team is assembled to create, deploy, and sustain the resultant system.
5. Must be able to deal with **systems that are evolved/composed** rather than designed from scratch. Even though designers have used components for a long time, it is only recently that we have begun to recognize the immensity of the problems presented by composed systems. Traditionally we have attempted to sidestep this issue by ensuring that we start with well-characterized components (standard components in the computer hardware realm, people sent through Boot Camp in the military personnel realm, *etc.*). But the explosive growth of the Internet and the PC software industry are forcing us to face the reality that we will have to create systems out of components that are far less than perfect, and that may not even be well understood. At a basic level, system architects must be cognizant of the fact that they will use poorly understood and imperfect components, and look for the best available information to measure the risks involved in doing so. At an advanced level, system architects should look for ways to create robust interfaces that encapsulate problems within components as much as possible.
6. Must have **good "taste"** to create a "clean" architecture that scales well and ages gracefully in the face of unforeseeable changes. At a certain level, system architecture is still very much art rather than science. Architects should be aware of this, and if nothing else be exposed to case studies that illustrate central issues and suggested approaches.

System architects must, of necessity, draw upon ideas from many different areas. Technical areas I have dealt with that come close to system architecture include design methodology, software engineering, and system engineering. Beyond that, important case studies can be drawn for success stories involving general system architecture principles. Computer hardware engineering successfully spans a huge dynamic range of scale via standardized abstractions and considerable automated support infrastructure (materials / polygons / transistors / gates / logic blocks / functional units / processors / computers / clusters / the Internet). The military and the telecommunications industry have much insight to offer in terms of life cycle issues. Other areas no doubt abound, but require someone with dual expertise in both a domain and the area of system architecture to harvest ideas for use. My personal area of interest is embedded distributed systems, which in many ways form a useful vehicle for illustrating system architecture principles while providing a rich enough problem space to begin an attack upon larger issues.

Course summary

While not a course specifically on system architecture, the below course summary describes some system architecture content I am currently teaching at Carnegie Mellon University. The course was taught for the first time in Fall 1999 with 30 seniors/MS students, and is planned for Fall 2000 with approximately 50 students.

Course Synopsis:
18-540 Distributed Embedded Systems, Carnegie Mellon University ECE Dept.
Instructor: Phil Koopman

Embedded computers seem to be everywhere, and are increasingly used in applications as diverse as transportation, medical equipment, industrial controls, and consumer products. This course covers how to design and analyze distributed embedded systems, which typically consist of multiple processors on a local area network performing real time control tasks. The topics covered will include issues such as embedded communication protocols, requirements analysis, synchronization, real-time operation, fault tolerance, distributed I/O, design validation, and industrial implementation concerns. The emphasis will be on areas that are specific to embedded distributed systems as opposed to general-purpose networked workstation applications. This course assumes that students already know fundamental topics such as interrupts and basic I/O, as well as architectural concepts such as memory systems and networks that are commonly taught in introduction-level embedded system courses.

Standard course coverage:
The standard course coverage is designed to teach the key skills needed by new engineers entering the embedded system industry, and assumes that students already have a basic skill set in “introductory” embedded systems (where the “introductory” level is the level dealt with by the overwhelming majority of embedded system courses and textbooks).

- Embedded systems in the real world
- Distributed systems and why embedded distributed systems are different
- Distributed time
- Modeling & architecture
- Performance issues
- Embedded communication protocols (including CAN and TTP protocol case studies)
- Protocol building blocks & performance analysis
- Robust system design & fault tolerance techniques
- Distributed I/O (sensors and actuators – not disk drives)
- Distributed real time scheduling
- Validation and certification
- Elevator architecture as a semester-long case study, with examples from other applications too

Advanced coverage:
Only the top one-third or so of the class is really ready to learn advanced system architecture concepts. Thus, the course is taught on two levels at once, with advanced coverage presented but not made into testable material. This advanced material is interspersed as “war stories” told during lectures, as scenarios for homework problems, and as issues that lie in wait just below the surface of assigned problems. This approach has been found to be helpful in motivating students looking for deeper coverage without alienating students who are not yet ready or willing to understand the deeper issues. Specifics include:

- Business issues, including: various models for company profitability and how they affect tradeoffs; the reality of non-technical constraints on solutions.
- Levels of abstraction, including: top-down decomposition + bottom-up synthesis; the desirability of finding orthogonal building blocks; multi-technology design tradeoffs
- Non-functional requirements, including: time to market; cost; safety; “ilities”
- Life-cycle perspective, including: requirements phase through disposal phase; selected real-world issues such as logistics, cross-cultural design issues, and ethics