Statement of Research Interests

Eddie Kohler

Many systems today have poor extensibility, flexibility, correctness, and security, all of which are easier to fix when a system is easy to understand. Understanding a real, complex software system should be as easy as reading a page or two of code. This would require high-level comprehension of the system; however, we design systems with architectures that don't facilitate higher-level comprehension, and the tools we use to program, analyze, test, and debug them don't help either. I want to create new architectures and tools that will facilitate higher-level comprehension and test them on real-world systems. I want to build systems that satisfy tough performance requirements, but are still flexible and understandable. This work will combine my interests in systems, programming languages, and software design with a willingness to rethink the way systems should be built.

As a first approach, I want to treat systems as programming languages, an avenue inspired by my graduate school research. In this model, a new programming language is an integral part of each system. This language specifies system-level properties, rather than the data layout and control flow issues controlled by languages like C and C++. Its constructs correspond directly to high-level properties of the system, which may be different even for two systems in the same domain. (For example, an operating system language might specify how file systems, network connections, and other kernel-level objects interact. Different operating systems would probably have different languages.) A program in the language gives us a readable, high-level description of the system, and we can manipulate this program with language processors to optimize or statically check the system as a whole.

The first of two projects that inspired this approach is Prolac, an object-oriented language for network protocol implementation. Prolac was designed to make protocol implementations readable, modular, extensible, and efficient. It includes advanced language features, appropriate syntax, tight integration with unforgiving environments (Unix kernels), and compiler optimizations. Our Prolac TCP implementation is organized in a new, readable and modular way; for example, each TCP protocol extension is readably located in one small set of subclasses. The compiler optimizes most of this organization away, leaving code that performs comparably to commodity TCPs.

The Prolac language is low-level and detailed, however, which complicates adoption. Most people don't want or need to learn another systems programming language. Therefore, in my next project—the Click modular router—we kept the programming language higher level, specifying the interaction between components rather than the instruction-by-instruction behavior of each component.

Click is an architecture for building software routers from modular components called *elements*. Element definitions are written in C++; the Click programming language specifies how elements should be connected in a particular router. A router's high-level behavior is thus exactly determined by its definition in the Click language. Network administrators can create arbitrary feature combinations by manipulating easy-to-read Click-language files, rather than by hacking kernel source code.

Like formal specifications, the Click language can be used to detect errors statically and to discover global router properties. Unlike formal specifications, the language compiles into working routers, so people can test and modify the way their packets are routed by manipulating Click-language files. Programs analogous to conventional language processors can manipulate these files as well, to optimize them, analyze them, and so forth. We have written several such programs already: a pattern-based optimizer that replaces common element arrangements with faster equivalents; an aligner, which examines a configuration and adds elements to ensure that packet memory is correctly aligned; and a specializer, which creates new, faster elements tailored to how the elements in a configuration are actually being used.

We have implemented the Click architecture on conventional PC hardware with good results. Click routers are flexible—their configurations can be easily read, understood, and changed. Already, performance is good enough for many demanding applications. Furthermore, having the Click language made the core system better by guiding us towards better internal structures and element designs.

Now I want to expand from these lessons into other systems. While my previous work has been in networking, my interest in systems is omnivorous. Networking, operating systems, and distributed applications—particularly over networks of small devices—all seem like fertile ground for this research. Regardless of area, I want to build real systems; I believe you learn from interacting with complexity, and little is more complex than the real world. These new systems probably won't be much like Click or Prolac, because they will be tailored for different problems. Nevertheless, successful system languages may share some basic principles, which I would like to pin down.

This work is fundamentally interdisciplinary in nature. Building a successful system language requires expertise in languages, systems in general, and the particular kind of system being built. I look forward to working in many system areas through close collaboration with peers and, especially, students.

Statement of Teaching Interests — Eddie Kohler

I love learning, and believe it is a responsibility, a privilege, and a joy to teach in return. My love for learning and teaching is broader than any one field, so much of this statement is about teaching in general. So far, I have taught classes in programming languages and systems, and would like in future to teach undergraduates or graduate students in languages, operating systems, networking, and/or elementary courses. My teaching is driven by flexibility, clarity, and the desire to reach all students.

At MIT, I have taught informally, as a lab assistant, and twice as a teaching assistant: for an undergraduate course and a graduate course. My most extensive teaching experience was the graduate course on programming languages. I taught recitation (weekly sections with 25 to 30 students) and lecture once or twice when the professor was out of town; answered students' questions; led quiz reviews; and developed course material, including quizzes and code. I was honored with two awards for the teaching assistantships, one for contributions to students' writing skills and one for general excellence.

My goal in teaching is to reach every student. (Many students reported on evaluations that they "truly believed that [I] cared about their performance in the course.") To reach everyone, a teacher must be flexible and provide different ways to learn the material; then students can choose the ones that work for them. I concentrate my efforts on clear, intuitive explanations, but let students help guide the pace and path of my lectures, and use other techniques whenever they help—from design problems to metaphors to games.

Teaching can also be inspirational, exciting students about computer science and computer systems. Inspiration, I believe, comes mostly from working on well-chosen, difficult, and rewarding problems. Lectures are important for providing intuition, but working on problems is what makes intuition stick; and only solving problems provides the adrenaline rush of inspiration. I put a lot of effort and creativity into creating problems and projects that are meant to inspire.

Of course, the most rewarding problems are research problems, and research and teaching are deeply linked, sharing a concern for the clearest explanation and the simplest solution. Advising graduate students (and undergraduate researchers) requires flexibility and excitement, even more than teaching a lecture. But then the rewards are greater—you end up with a collaborator and a peer.