Professional statement of Michael D. Ernst

http://homes.cs.washington.edu/~mernst/
August 2013

My research aims to improve programmer productivity; that is, to make software more reliable, more secure, and easier (and more fun!) to produce. I develop theoretical and practical techniques and tools for helping people to create, understand, and modify software systems. To contribute to these goals, I work in the areas of software engineering, programming languages, type theory, security, static and dynamic program analysis, bug prediction, testing, verification, and development environments. My results often stem from cross-fertilization between traditionally separate research areas: experimental vs. theoretical, syntactic vs. semantic, static vs. dynamic, exact vs. inferred.

Much of my research is empirical in nature: I am concerned not only with how people should, but also with how they do, develop and maintain software. Even my more speculative work is motivated by practical problems encountered by programmers (including myself!) who wish to create or maintain reliable software. This desire to cross-fertilize science and engineering makes me more likely to put theory to work in the service of applications than to do theory for its own sake.

This attitude leads me to be an experimentalist: my research ideas are realized in prototypes and are evaluated by use in the lab, by case studies, or by controlled human experiments. I freely share my programs and experimental infrastructure; I believe this is an important part of responsible scientific practice, so that others can reproduce and extend the results and implementations.

Formal methods frequently inspire me, both as powerful techniques and as sources of pain for practicing programmers. My work makes formal methods easier to use and finds new applications for both exact and approximate specifications. For example, dynamic invariant detection is a type of behavior modeling that is an alternative to formal specifications. It was inspired by the desire to perform formal verification — even though formal specifications are usually absent, and are too difficult and expensive to develop for most software systems.

Machine learning is another theme in my research. Programmers embed substantial knowledge in their programs and test suites, and machine learning provides a way of reifying and exploiting that knowledge (e.g., via semantics-based program analysis).

Another of my goals is to bridge the gap between static analysis (which tends to be sound) and dynamic analysis (which tends to be more precise). Much of my work incorporates both static and dynamic components.

I am perhaps best-known for my research in dynamic analysis (one notable example is Daikon), testing (one notable example is Randoop), and type systems (notable examples include immutability and Java pluggable types), but my research encompasses many other topics as well. I next present selected research contributions from the past two years. My webpages give a more complete list of my recent and earlier work.) I have been fortunate to work with talented colleagues, and I am particularly proud of my students’ accomplishments. However, for brevity and simplicity this document uses singular pronouns (“I”, “my”). Finally, I discuss some educational contributions.

Programming language design

My work in programming language design focuses on bringing new capabilities to real languages in a backward-compatible way (which is critical for evaluation and adoption), and building implementations to assess their utility. This pragmatic approach promises to aid both today’s and tomorrow’s programmers.

I created the first practical framework for pluggable type systems in Java, the Checker Framework [PAC+08, DDE+11]. This tool enables programmers to define type qualifiers and enforce their semantics at compile time [PE07]. Google is using our system on over 20 of their projects. Oracle is so excited about this research that it is adding support for type qualifier to the Java 8 language. I designed and implemented the syntax [Ern11]. I am the first non-Sun/Oracle employee to be the specification lead for a Java language change. As another benefit, the changes make Java a better platform for experimentation in language design.
We have written dozens of type-checkers and found hundreds of programmer-verified bugs in real programs. One recent example is a system for preventing errors related to regular expressions [SDE12]. Another example is for immutability [ZPL+10, PÖZE13], where our approach is concise (low annotation burden) yet powerful (it type-checks the JDK without code changes, supports creation of immutable circular data structures, and supports both reference and object immutability). My work on rely-guarantee references [GEG13] pushes the envelope regarding what properties can be statically guaranteed about side effects and aliasing.

My research in type inference complements my type-checking work. I created a type-inference system for ownership [DEM11] that enables a user to express preferences among multiple legal typings, and another that generalizes multiple previous ownership algorithms [HDME12]. I have created multiple static and dynamic approaches to inferring immutability (e.g., [HMDE12], and many of these systems combine aspects of immutability and ownership. Another inference project is a flow-sensitive, exception-aware static analysis of field initialization [SE11]. I am in the process of generalizing these inference algorithms, much as the Checker Framework generalized previous work on pluggable type-checking and made it easier to create and evaluate new type systems.

I created a type system that combines the best features of static and dynamic typing [BCE11]. Unlike competing proposals, there is no need for the developer to differentiate between statically-typed and dynamically-typed parts of the program, and static type-checking provides a guarantee of no dynamic type errors. Nonetheless, the program can be run at any time, regardless of static type-checking errors, and errors are deferred as late as possible so that inaccurate static types do not impede execution and experimentation.

Testing

My early work in testing focused on test generation, particularly by exploiting impoverished test suites. Users seem willing to write small test suites or to supply a few sample executions, but they are reluctant or unable to write more comprehensive or more focused test suites. My research automated these and other testing tasks, usually by combining static and dynamic techniques [ZSBE11].

My more recent focus has been on helping programmers to understand their tests. In one project, I scaled up test generation techniques so that they apply to programs rather than just libraries, and so that the resulting test cases are maintainable (easy to understand and to adapt to changes in the program) [REP+11]. The technical ideas include controlling side effects, eliminating redundancy, and exploiting values from the program. In another project, I created a technique that assists debugging a failed test [ZZE11]. The FailureDoc tool isolates the parts of a failed test that are most relevant to the failure, which helps programmers to understand and fix the failure. It creates and analyzes an abstract object profile, and and it works even when test-reduction techniques such as Delta Debugging do not.

I have also been helping end users to understand and fix errors. Sometimes, incorrect software behavior results not from a program bug, but from an incorrect configuration option supplied by the user. The ConfDiagnoser tool suggests a configuration option to change so that the software behaves as desired [ZE13]. ConfDiagnoser uses static analysis, dynamic profiling, and statistical analysis to link the undesired behavior to specific configuration options. It does not require a testing oracle, and it can diagnose both crashing and non-crashing errors. Another common problem is broken user workflows: when a UI changes, a user must perform different UI actions to achieve the same task. The FlowFixer tool uses dynamic profiling, static analysis, random testing, and statistical correlation to suggest a replacement UI action that fixes a broken workflow [ZLE13].

Invariant detection

In previous work, I introduced the idea of dynamic invariant detection: performing machine learning over program executions in order to mine (likely) specifications. This new approach, embodied in the widely-used Daikon tool, brought together the formal methods and dynamic analysis communities and spawned a new community. It also won multiple awards (most recently the 2013 ACM SIGSOFT Impact Paper Award for the most influential work done at least 10 years earlier) and was commercialized by at least 3 companies.
Whereas Daikon focused on data structure invariants, my more recent work has focused on temporal properties. The Synoptic tool \cite{BBS+11} reads the logs that systems already write, and it produces a finite-state-machine model of the system. Its models are more accurate and useful to programmers than previous systems because it mines properties from the original logs and then produces the most concise and general model that satisfies those properties. It is efficient because instead of creating and manipulating a huge model until it is sufficiently general, it creates and manipulates a tiny model until it is sufficiently expressive.

Subsequent work \cite{BBA+13} showed how to declaratively express model-inference algorithms such as Synoptic and kTails, which have previously been expressed procedurally. The declarative approach is more principled, easier to understand and modify, and faster. Applications include test generation \cite{BBE+11a}. Ongoing work generalizes Synoptic to distributed systems \cite{BBE+11b}, which requires new formalisms rather than relying on well-understood ones such as FSMs.

**Speculative analysis**

Ordinary program analysis informs a programmer about the program after he has written it. I proposed speculative analysis, which informs a programmer about a program before he writes it \cite{BHEN10}. Speculative analysis works by guessing what actions a programmer might perform, executing those actions in the background, analyzing the resulting program, and communicating the results to the programmer. Information about the consequences of his actions permits a programmer to make better decisions and avoid rework. This research is a natural extension of continuous testing \cite{SE03}, which uses excess cycles on the developer’s workstation to continuously run regression tests in the background. Speculative analysis can also be viewed as an exploration of the limits of program analysis: how can analysis be made most useful to programmers, if computation is free? Nonetheless, the tools are practical.

I created a speculative analysis for detecting and preventing editing conflicts among team members much earlier than they would otherwise be noticed \cite{BHEN11, BMH+12}. The approach proactively merges different developers’ versions of a codebase and unobtrusively displays the results. When no conflicts exist, this permits developers to incorporate other developers’ edits without fear of disrupting their work.

Another application of speculative analysis is for IDE “quick fix” recommendations \cite{MBH+12}. Most IDEs let a developer modify his or her program, but do not indicate the consequences of those changes. The Quick Fix Scout tool shows the consequences of Eclipse’s Quick Fix recommendations, in terms of compilation errors introduced or eliminated. It focuses developer attention on the most useful Quick Fixes and prevents wasted time exploring dead ends.

Speculative analysis has led to a general approach for converting offline, batch analyses into continuous ones that operate on the latest version of the codebase without interfering with the developer \cite{MBEN13}.

**Crowdsourcing**

I have applied crowdsourcing to solve verification problems in two different ways. The first approach uses completely unskilled workers who know nothing about programming, by converting a program verification problem into a game \cite{DDE+12}. The game embodies the same logical constraints as the verification problem, but it exploits human physical intuition. Creating the game requires novel program analysis. This idea gave rise to DARPA’s CSFV program, which is funding half a dozen research teams to pursue the idea. The second approach uses low-skilled workers who understand programming but are unable to perform a verification task independently \cite{SE10, SE12}. A novel user interface decomposes a verification task into manageable subtasks and chooses an order to guide users through those subtasks. The workers are given less autonomy, but they become more productive.
Concurrency

I developed a type system that guarantees that a program will not deadlock [GEG12]. This system is more flexible than previous systems — that is, it accepts safe code that previous approaches prohibit, including fine-grained locking, arbitrary lock acquisition orders, and changing the locking order.

I developed two distinct approaches to detect the most important and common error that crashes multi-threaded GUI applications: only the GUI thread should access UI objects. One approach formulates the problem as call graph reachability, requires no programmer annotations, handles reflection, and found errors in Android, Eclipse plugin, Swing, and SWT applications [ZLE12]. The other approach formulates the problem as a type system and gives a guarantee of no deadlocks while achieving equivalent precision [GDEG13].

Teaching

My teaching (both in the classroom, and research mentoring) stresses both theory and its practical applications, for two reasons. First, reifying an abstraction often makes it easier to understand and remember: many people most easily grasp the general via the specific. Second, and perhaps more important, grounding concepts motivates them: it shows why we should care about them and how they might be useful. I discuss practical applications as a powerful way of bringing the curriculum alive, and I use them not just to explain old ideas but to lead students to new discoveries that are relevant to them. When students get to explore new material, they become more excited by the material and take it to heart. It is then that teaching is most effective, exciting, and worthwhile. It is also natural to segue from coursework (learning facts that the student does not know) to research (learning facts that no one knows). A common theme in my teaching is letting students try-fail-understand-try-succeed: this illustrates the inadequacy of naive methods, emphasizes the utility of the techniques being taught, and improves confidence and enjoyment.

I applied these ideas at Rice University, where I introduced a project-oriented laboratory in software engineering and redesigned the data structures and algorithms class from scratch. At MIT, I improved 6.170 Laboratory in Software Engineering by making specification and testing an integral part of the curriculum, by using an integrated sequence of problem sets instead of disassociated problems, and by having students correct their programs a month after receiving initial grades. I created the Groupthink specification exercise [Ern06], which teaches how to read and write specifications and how to communicate with teammates. It is popular with students across the School of Engineering because specification is shown as a means to a plausible end, students are given the chance to reflect and improve, and students came out appreciating, not hating, specifications. I also created a month-long programming competition (6.187) that is more realistic than competitions of a few hours with artificial problems and a graduate class in program analysis (6.893/6.883), among other educational and service contributions. The Groupthink module and my other teaching materials have been used at universities across the country.

At UW, I have developed several innovative new classes, in addition to other contributions such as modifying our senior-level software engineering project class to involve real-world clients, and creating new offerings of graduate compilers and software engineering.

I created a junior-level class (CSE 331 Software Design and Implementation) that takes students from knowing how to write a program, to knowing how to design and write a correct program. The class introduces concepts such as abstraction, modularity, specification, object-oriented design, invariants, formal and informal proofs, and program decomposition — topics that had been missing from the curriculum or had been treated in an informal and fragmentary manner. Other faculty report an improvement in students who have taken the new class.

I created a new introductory computing class (CSE 140 Data Programming) aimed at both majors and non-majors. The class is unique in the world, to the best of my knowledge, in that every assignment solves a real-world problem using a real dataset from science, engineering, or business. The class is aimed at service to the university, since every educated professional is aided by an understanding of how to analyze data computationally. It also serves to attract people to computer science who wouldn’t take a CS class or would be turned off
by standard assignments about games, puzzles, and abstract mathematical tasks such as computing the factorial function. Students get to see how computing is relevant in practice to their lives. This focus also affects the class’s choice of topics and the way that it is taught. Within a year, though word of mouth, Pacific Union College and Evergreen College adopted this class in place of their existing introductory programming classes, and it is also being taught at Southern Adventist University. I created this class on my own time and taught it without it counting toward my teaching requirements.

My classes are well-received by students. For each of the 4 classes I taught in the past two years, the Dean of Engineering has commended me for teaching one of the 10 best-rated classes in the college. This is in addition to other such commendations in the past.

I enjoy combining research and education, for I find the activities complementary: practicing either strengthens the other, for both are about simplifying complex phenomena into basic, easy-to-understand underlying concepts. My goals are to deepen students’ understanding of the material, show them how theory is relevant, and excite them about contributing to scientific knowledge. In my 3 graduate classes in software engineering and compilers at UW, more than half of the student projects have turned into published research. In my undergraduate programming classes, I always introduce students to research tools, and many have gone on to do research themselves — and in a few cases their own tools have been used in subsequent offerings. My research tools have been used in dozens of classes around the world — both because they ease software development, relieving students of tedium and demonstrating the value of good methodology, and also because they aid understanding of program analysis and lead to new research ideas.

In addition to outstanding work by my PhD students, I have also nurtured undergraduate work. My MIT M.Eng. (a 5-year undergraduate/Masters degree) students won the departmental M.Eng. thesis award in 4 out of 6 years, and two of my undergraduates won the undergraduate research prize. My UW undergrad students have also received honors, including national fellowships, CRA research awards, etc. My undergraduate and 5th-year-Masters students have produced over 35 refereed publications.
Selected references

For a complete list of publications, see http://homes.cs.washington.edu/~mernst/pubs/.


