

Professional statement of Michael D. Ernst

<http://pag.csail.mit.edu/~mernst/>

November 2006

My chief research interest is software engineering, with a focus on programmer productivity. I develop theoretical and practical techniques and tools for helping people to create, understand, and modify software systems. My research activities span the spectrum from programming language design and type theory, to static and dynamic program analysis, to testing, to development environments.

Much of my research is empirical in nature: it concerns both how people *do*, and how they *should*, 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 (though with some exceptions [Ern95a, WE02]).

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 [NE02c, SE04b]. I freely share these artifacts; I believe this is an important part of responsible scientific practice, so that others can reproduce and extend the results and implementations.

Formal methods are a frequent source of inspiration for me. My work reduces the burdens that they place on users and finds new applications for both exact and approximate specifications. As one example, my research on invariant detection (a type of behavior modeling that provides an alternative to formal specifications) was inspired by the desire to perform formal verification—despite the fact that formal specifications are usually absent, and are too difficult and expensive to develop for most software systems.

A related goal is to raise tools from the syntactic to the semantic level, creating a discipline of semantics-based software engineering. I was the first to propose refactorings based on program behavior rather than lexical metrics such as the number of lines in a method [KEGN01]. My work on test suite generation measures coverage based on program semantics rather than lexical properties [HME03].

Machine learning is another theme in my research [EF89, EMW97, DLE03, EP05]. 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). For example, I applied machine learning to the domains of error detection [BE04b] and mode selection [LE04] by use of novel representations of the abstractions of those domains. My work on invariant detection [Ern00] devised new machine learning algorithms for the domain of program executions. Generation of models for test generation [AEK⁺06] takes a temporal tack.

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 and scalable) [Ern03]. Much of my work incorporates both static and dynamic components; examples include my work on component upgrades [ME03, ME04a, ME04b], test factoring [SE04a], verification [NE02b, NE02c, NEG⁺04], test generation [PLEB06, dPM⁺06, AEK⁺06], side effect analysis [BE04a, TE05, AEGK06], and type inference [GPME06], among others.

I next present some recent research contributions. I have been fortunate to work with talented colleagues, but for brevity 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, and building implementations to assess their utility. This pragmatic approach promises to aid both today’s and tomorrow’s programmers.

I devised a type system for expressing and enforcing immutability constraints: an immutable reference cannot be used to modify the abstract state of its referent [BE04a, TE05]. Such a type system increases expressiveness; provides machine-checked documentation; detects and prevents subtle errors; and enables transformations. For a statically type-safe language, the type system guarantees reference immutability. If the language is extended

with immutability downcasts, then run-time checks enforce the reference immutability constraints. The system is implemented in a backward- and forward-compatible way in the Java language and has been demonstrated on programs of up to 120,000 lines of code. I developed a formal model for expressing type soundness and two distinct type inference algorithms [Tsc06, AEGK06].

I designed an extension to Java's annotation system that permits annotations to appear on any use of a type and lifts other restrictions. This makes user-defined type qualifiers practical in Java for the first time, and our tools enable programmers to define their own type qualifiers and enforce their semantics at compile time. Sun accepted the proposal for incorporation into Java 7 [EC06].

Static analysis

My recent static analysis work uses type inference and checking, abstract interpretation (dataflow analysis), and pointer analysis to address parametric polymorphism and deadlock.

I designed sound, precise, and scalable analyses for refactoring Java programs to use parametric polymorphism ("generics"), which improves type safety and expressiveness. One analysis [DKTE04] converts client code to supply type parameters at each use of generic classes. It uses a context-sensitive pointer analysis to determine possible types at allocation sites, and a set-constraint-based analysis (that incorporates guarded, or conditional, constraints) to choose consistent types for allocation and declaration sites. Another analysis [KETF06] addresses the problem of converting library definitions; this is harder, since adding type parameters changes the very constraints that may have led to adding the type parameters. Another difficulty is that many legal but undesirable solutions exist. Ours is the first correct (not to mention practical) solution, and it outperforms human experts and has found better parameterizations for widely-distributed libraries (as confirmed by the libraries' authors).

I devised a flow-sensitive, context-sensitive deadlock detection technique for libraries [WTE05]. Unlike approaches for single programs, it guarantees that no client program can cause deadlocks in the library code. I found exploitable deadlocks in the JDK and elsewhere, and proved other libraries to be free of deadlocks.

Verification

Formal verification remains a labor-intensive activity beyond the scope of most development efforts. My research automated hitherto manual parts of the program verification process.

Automated theorem provers require written annotations or specifications, which can be difficult and tedious for humans to provide; many researchers consider it as hard to decide what to prove as to actually prove it. By contrast, dynamic modeling techniques provide a set of candidate annotations, but those properties are potentially unsound. My research combines static and dynamic analyses, guaranteeing soundness of the dynamic analysis and lessening the annotation burden for users of the static analysis [NE01, BCC⁺05]. In experiments, over 90% of the dynamic analysis output was provable, and it supplied over 90% of the annotations that would otherwise be written by a human [NE02b]. A controlled human experiment verified the benefits for programmers [NE02c].

Proof assistants can perform more sophisticated proof steps but perform more limited search. Proof assistants require the user to supply not only the goal, but also the lemmas that lead up to it and each step of the proof. I showed how to obtain lemmas from generalization over program executions, and proof steps (or "tactics") from the structure of the test cases [NEG⁺04]. The technique reduced by over 90% the human interaction required to prove three realistic distributed algorithms, for both the LP and Isabelle theorem provers [NE02a].

Coping with unanticipated environments

Many software failures result from use of software in circumstances for which it was not designed or tested. I have devised several analyses that address the important and practical problem of mitigating such failures.

I developed a technique to assess whether replacing a component of a software system by a purportedly compatible component may change system behavior [ME03]. An automated logical comparison checks that the new component was tested (by the vendor) in all the circumstances in which the old component was run (by the

user). The technique operates before integrating the new component into the system or running system tests, it is user-specific, and it requires no formal specifications. Our implementation identified incompatibilities among (supposedly compatible) versions of the Linux C library, and it approved the upgrades for other programs that were unaffected by the changes [ME04a]. I proved correctness of the upgrade model [ME04b].

I proposed a technique for finding latent code errors, which are not yet manifest as user-visible faults [BE04b]. The technique generates machine learning models of program properties known to result from errors, then applies these models to new code to rank properties that may lead to errors. The technique outperforms previous attempts by using a superior model of program behavior that permits generalizing over disparate programs.

Some programs operate in several modes, such as robot navigation on roads, in buildings, or in open terrain. Multi-mode programs may under-perform in unanticipated environments, even if they operate properly in situations for which they were designed and tested. Program steering [LE04] generalizes from good executions to build a new mode selector that can replace the built-in one when the program is performing poorly. In experiments, program steering substantially improved performance in unanticipated situations.

I automated and scaled up a previously manual technique for run-time enforcement of data structure consistency [DEG⁺06], by automatically inferring the consistency specifications; violations of them are repaired at run time. The resulting system withstood a hostile “Red Team” evaluation in which an outside company attempted to defeat the protection mechanism.

Dynamic analysis

Dynamic analysis is complementary to static analysis: although dynamic analysis is often unsound, it also tends to be more precise, scalable, and applicable to legacy programs. My work applies dynamic analysis, with good effect, to problems that in the past have only been addressed statically.

I designed an algorithm for inferring abstractions in programs that use primitive datatypes [GPME06]. It infers programmer intent from the structure of operations and the interaction of values and variables in the program. Implementations, for both C++ and Java, use a novel multi-level analysis that simultaneously utilizes both source-level and binary-level information. Programmers found the results helpful for program maintenance.

I devised a staged approach to mutability (side effect) analysis [AEGK06], which indicates which variables are used in a read-only manner and which may be modified. My approach combines multiple fast, simple mutability analysis stages (both static and dynamic). All stages are sound, but accommodate optional heuristics that trade off soundness and accuracy. The resulting system outperforms previous analyses (some of which are extremely complex and sophisticated), in terms of both scalability and overall accuracy.

I devised the first technique that automatically proposes refactorings based on semantic, not syntactic, properties. In a case study, a developer agreed with most of the recommendations regarding his code [KEGN01].

Testing

A common theme in my testing research is investigating how to exploit 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 helps to automate such tasks, among other testing tasks. The research spans test generation, test selection, test classification, and test execution.

I devised techniques that guide random testing to generate test inputs that are structurally complex while avoiding ones that are either illegal or redundant. One technique incorporates feedback from executing previously-generated inputs [PE04a, PLEB06]; that is, it integrates the test generation and test execution phases to improve them both. It found serious, previously-unknown errors in widely-deployed commercial applications. Another technique uses inferred models of legal method call sequences to bias test generation toward legal method sequences, increasing coverage and creating data structures beyond the ability of undirected random generation [AEK⁺06]. Our random testing techniques are the first to outperform symbolic and systematic techniques on their own benchmarks [dPM⁺06, PLEB06], and our techniques scale much better.

Test factoring [SE04a] automatically creates fast, focused unit tests from slow system-wide tests. Each new unit test exercises only a subset of the functionality exercised by the system tests (thanks to a capture-replay technique that create mock objects for untested behavior), so it is more amenable to test selection and prioritization. Use of factored unit tests helps to catch errors earlier in a test run, reducing the time until a programmer becomes aware of an error [SAPE05].

Automated techniques can generate many test inputs, but without a formal specification, it can be difficult to know whether a particular execution has triggered a program error. I devised a technique for selecting tests that are likely to expose errors — tests whose run-time behavior is maximally different from succeeding runs [PE04a]. The same paper also gives techniques for test input generation and for converting a test input into a test case.

The operational coverage technique [HME03] is analogous to structural code coverage techniques, but it operates in the semantic domain of program properties rather than the syntactic domain of program text. The technique is automated, and it outperforms structural coverage techniques that require substantial human effort. More importantly, it is complementary to structural techniques, so both can be productively applied.

Continuous testing [SE03] uses excess cycles on the developer's workstation to continuously run regression tests in the background. This simple but novel and practical idea provides early notification of errors, which reduces the cost of fixing them. Developers need not remember to run tests, nor wait for them to complete. A controlled human experiment indicated that continuous testing reduces errors [SE04b].

Invariant detection

Invariants, such as those in assert statements or formal specifications, are useful for many purposes. Programmers rarely provide them, because of their high costs and limited benefits. My research addresses both parts of the value proposition: it makes likely invariants easier to obtain, via automated dynamic analysis, and it makes them more useful, by creating analyses and tools that utilize them. Dynamic invariant detection demonstrated that it is possible to automatically capture valuable semantic information from program runs, and it demonstrated that such information can be usefully applied to tasks that had been traditionally performed with sound abstractions. Additionally, dynamic invariant detection opened the way to new analyses for which program models generated dynamically are better-suited than those from static analysis or from humans. Dynamic techniques for determining invariants complement static ones, which lack knowledge of the program execution environment and for which real-world constructs like pointer analysis remain difficult, unsolved technical problems.

Since introducing the concept of dynamic invariant detection [ECGN99, Ern00, ECGN01], I have extended the technique to pointer-directed data structures [EGKN99] and conditional properties (implications) [DDLE02, DLE03], devised techniques that filter out less useful properties and add useful ones [ECGN00], and developed more scalable online algorithms [PE04b].

This work is being commercialized by 3 companies (Agitar, Determina, and Scrutiny) and is used internally by many more. It has inspired over a dozen additional implementations of invariant detection. I distribute and support the Daikon invariant detector [EPG⁺06]. Around 100 publications describe research that depends on running Daikon as part of its methodology. Yet more research has used the Daikon source code in case studies.

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. Discussing practical applications is a powerful way of bringing the curriculum alive, and all the more so when it leads the way to new discoveries rather than merely explaining old ideas. 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 learning facts that one does not yet know to learning facts that no one yet knows.

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. Likewise, I made several changes to MIT's 6.170 Laboratory in Software Engineering. I made specification and testing an integral part of the curriculum, composed realistic problems that were hard to solve without using these techniques, graded students on the quality of their programs, and used an integrated sequence of problem sets instead of disassociated problems. Students came to learn that specification and testing are powerful techniques for building programs, not tedious exercises to be done for their own sake in the classroom and with little relation to software engineering. I similarly motivated documentation and correctness by having students return to their programs a month after writing them; this also gave them a chance to return to a problem with better skills and to observe their own progress. I took a similar approach in creating one of the most popular modules of the UPOP (Undergraduate Practice Opportunities Program) IAP (January wintersession) activity [EC05]: specification was a means to a plausible end, students were given the chance to reflect and improve, and students came out appreciating, not hating, it. This UPOP module and my other teaching materials have been used at universities across the country. A common theme in my teaching is letting students *try-fail-understand-succeed*: this illustrates the inadequacy of naive methods, emphasizes the utility of the techniques being taught, and improves confidence and enjoyment. I also created another IAP class — 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 contributions.

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. I aimed some of my research at educational venues [Ern95a] (even though it was subsequently invited for submission to Theoretical Computer Science). My research tools have been used in dozens of classes inside and outside MIT — both because they ease software development, relieving students of tedium, and because they aid understanding of program analysis and lead to new research ideas.

My 12 M.Eng. (a 5-year undergraduate degree) students have produced 16 refereed papers. In the past 5 years, my students won three departmental M.Eng. thesis awards; a fourth project won the undergraduate research prize. My PhD students have also won many awards and fellowships.

Selected references

For a complete list of publications, see <http://pag.csail.mit.edu/~mernst/pubs/>.

- [AE05] Shay Artzi and Michael D. Ernst. Using predicate fields in a highly flexible industrial control system. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 319–330, San Diego, CA, USA, October 18–20, 2005.
- [AEGK06] Shay Artzi, Michael D. Ernst, David Glasser, and Adam Kiezun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 18, 2006.
- [AEK⁺06] Shay Artzi, Michael D. Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, USA, October 23, 2006.
- [BCC⁺05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
- [BE04a] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.
- [BE04b] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.
- [DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002. Draft. <http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps>.
- [DEG⁺06] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 233–243, Portland, ME, USA, July 18–20, 2006.
- [DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 15–34, Vancouver, BC, Canada, October 26–28, 2004.
- [DLE03] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 21, 2003.
- [dPM⁺06] Marcelo d'Amorim, Carlos Pacheco, Darko Marinov, Tao Xie, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 2006: Proceedings of the 21st Annual International Conference on Automated Software Engineering*, pages 59–68, Tokyo, Japan, September 20–22, 2006.
- [EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.
- [EC05] Michael D. Ernst and John Chapin. The Groupthink specification exercise. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 617–618, St. Louis, MO, USA, May 18–20, 2005.
- [EC06] Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. <http://jcp.org/en/jsr/detail?id=308>, October 17, 2006.
- [ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.
- [ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.
- [ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

- [EF89] Michael D. Ernst and Bruce E. Flinchbaugh. Image/map correspondence using curve matching. In *AAAI Spring Symposium on Robot Navigation*, Stanford, CA, March 28–30, 1989. Also published as Texas Instruments Technical Report CSC-SIUL-89-12.
- [EGKN99] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington Department of Computer Science and Engineering, Seattle, WA, November 16, 1999. Revised March 17, 2000.
- [EKC98] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.
- [ELP06] Michael D. Ernst, Raimondas Lencevicius, and Jeff H. Perkins. Detection of web service substitutability and composability. In *International Workshop on Web Services — Modeling and Testing*, pages 123–135, Palermo, Italy, June 9, 2006.
- [EMW97] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Aichi, Japan, August 23–29, 1997.
- [EP05] Michael D. Ernst and Jeff H. Perkins. Learning from executions: Dynamic analysis for software engineering and program understanding, November 7, 2005. Tutorial at 21st Annual International Conference on Automated Software Engineering.
- [EPG⁺06] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.
- [Ern94a] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.
- [Ern94b] Michael D. Ernst. Serializing parallel programs by removing redundant computation. Technical Report MIT/LCS/TR-638, MIT Laboratory for Computer Science, Cambridge, MA, August 21, 1994.
- [Ern95a] Michael D. Ernst. Playing Konane mathematically: A combinatorial game-theoretic analysis. *UMAP Journal*, 16(2):95–121, Spring 1995.
- [Ern95b] Michael D. Ernst. Slicing pointers and procedures (abstract). Technical Report MSR-TR-95-23, Microsoft Research, Redmond, WA, January 13, 1995.
- [Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.
- [Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.
- [GPME06] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 18–20, 2006.
- [HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.
- [KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.
- [KETF06] Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing Java classes. Technical Report MIT-CSAIL-TR-2006-061, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, September 5, 2006.
- [LE04] Lee Lin and Michael D. Ernst. Improving adaptability via program steering. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 206–216, Boston, MA, USA, July 12–14, 2004.
- [ME03] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE 2003: Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 287–296, Helsinki, Finland, September 3–5, 2003.
- [ME04a] Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, pages 440–464, Oslo, Norway, June 16–18, 2004.

- [ME04b] Stephen McCamant and Michael D. Ernst. Formalizing lightweight verification of software component composition. In *SAVCBS 2004: Specification and Verification of Component-Based Systems*, pages 47–54, Newport Beach, CA, USA, October 31–November 1, 2004.
- [NDE⁺01] David Notkin, Marc Donner, Michael D. Ernst, Michael Gorlick, and E. James Whitehead, Jr. Panel: Perspectives on software engineering. In *ICSE 2001, Proceedings of the 23rd International Conference on Software Engineering*, pages 699–702, Montreal, Canada, May 16–18, 2001.
- [NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.
- [NE02a] Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Laboratory for Computer Science, Cambridge, MA, May 25, 2002.
- [NE02b] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.
- [NE02c] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.
- [NEG⁺04] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.
- [PE04a] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. Technical Report 968, MIT Laboratory for Computer Science, Cambridge, MA, October 2004.
- [PE04b] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, Newport Beach, CA, USA, November 2–4, 2004.
- [PLEB06] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Redmond, WA, September 2006.
- [SAPE05] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.
- [SE03] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, November 17–20, 2003.
- [SE04a] David Saff and Michael D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, Washington, DC, USA, June 7–8, 2004.
- [SE04b] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 76–85, Boston, MA, USA, July 12–14, 2004.
- [TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.
- [Tsc06] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2006.
- [WCES94] Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, Portland, OR, January 1994.
- [WE02] Elizabeth L. Wilmer and Michael D. Ernst. Graphs induced by Gray codes. *Discrete Mathematics*, 257:585–598, November 28, 2002.
- [WTE05] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.