Wes Weimer - Research Statement

The world is increasingly dependent on software, yet software remains buggy. For every three dollars spent on software sales, one dollar is spent on software bugs. Annually, program errors cost more than one half of one percent of the United States GDP. Many recent research efforts to improve software reliability have focused either on finding or proving the absence of defects statically, at the source code level, or on monitoring software dynamically, while it is running, to prevent mistakes.

My main research interest lies in advancing software quality by combining static and dynamic approaches. I am particularly concerned with automatic or minimally-guided techniques that can scale and be applied easily to large, existing programs. For the purposes of scalability and utility I have been willing to sacrifice soundness and completeness. In such cases I use experiments to provide additional validation and also use the underlying theory to characterize to what extent precision has been lost. Finally, finding bugs is insufficient, and I aim to help programmers address defects and understand error reports. I am also interested in designing languages and language features to help prevent errors.

Run-Time Error Handling Mistakes

My doctoral research on mistakes made when handling run-time errors touches on all three aspects. Handling run-time errors is important for reliability. Up to half of a mature program may be devoted to error handling. Unfortunately, unexpected errors do not occur on demand and thus do not lend themselves to traditional testing. I have found that programmers often make mistakes in their error handling -- they often forget to release locks or restore invariants while cleaning up after an exceptional situation. Such mistakes often occur because it is difficult to reason about all control-flow paths associated with run-time errors.

Both purely static and purely dynamic techniques for software quality have drawbacks, but I combine the two to get useful information while sidestepping some traditional pitfalls. First, I proposed a static analysis that locates likely error handling mistakes. Static techniques would ideally be sound, complete, scalable, and give easy-to-interpret results, but in reality a technique will not have all of those properties. My analysis scales well to large programs and gives easily-understood error reports, but lacks precision. The static analysis is complemented by a novel dynamic language feature, the compensation stack, for remembering obligations in the face of run-time errors. Obligations like database locks and file handles are tracked at run-time, giving a solution that can handle cases the static analysis is unable to reason about.

The foundations for this work include dataflow analysis and path-sensitive program analysis, as well as database concepts like compensating transactions and linear sagas. While the static analysis is imprecise, its lack of precision can be well-characterized by simple rules, and the compensation stacks can be shown to discharge their remembered obligations in an expected order. The analysis and the language feature both scale well and are easy to use -- using them I have found over eight hundred in four million lines of code and improved the reliability of a medium sized program in a few hours.

Static and dynamic approaches both need a notion of what the "right" behavior is in order to point out when something is going wrong. Such specifications or policies must normally be constructed manually, creating a high barrier to entry for software quality approaches. In order to help address this problem in general, and based on my observations of what goes wrong in run-time error handling mistakes, I devised a novel specification mining algorithm that inspects a program and produces candidate specifications. These mined specifications can be used by any tool (including my work on run-time errors) to find bugs in that particular program. Compared to previous mining approaches my technique has a high signal-to-noise ratio and produces policies that can be rapidly accepted or rejected without complicated debugging. As a result my system continues to scale when specification mining is used, and the mined specifications have found 430 additional bugs in one million lines of code.

CCured

My earlier work on CCured and its type system and type inference system also fits into this pattern. At its heart, CCured brings type safety and memory safety to C programs. Purely dynamic approaches to this problem, like the well-known Purify tool, are typically sound, complete, and easy to apply but have an unacceptably high run-time overhead. Completely static approaches have low run-time overheads and must in general either falsely reject many safe programs or falsely allow many unsafe programs. CCured combines these two techniques by dividing the pointers in a program into various kinds, some of which are generally checked statically and some of which are generally checked dynamically. It turns out that even in a language like C, many pointers are simple and safe and thus need few run-time checks. Only a few complicated pointers require run-time bounds or run-time type checks. CCured thus has very low overhead because most of the safety guarantees can be proved up front by a strong static type system rather than with run-time monitoring.

In order to make CCured apply more easily and directly to existing programs I developed a pointer kind inference algorithm. Given a standard C program and a knowledge of the CCured type system, this algorithm divides the pointers in the program into CCured's pointer kinds such that the resulting program adheres to CCured's type system and contains (loosely) the smallest number of run-time checks. The inference algorithm works based on constraint systems, type systems and unification-based pointer analysis. Since the solution is not unique, the resulting assignment is often manually tweaked, but in general, with very little effort, existing C programs can be made safe with CCured. This project was also my first experience with the importance of providing programmers with multiple ways of understanding or dealing with reported errors. A pointer that requires expensive run-time checks can be viewed as a performance bug. Users of such a system want to know not just that a pointer will be slow, but also why it was inferred to be slow and can be done about it. As a result, CCured's inference algorithm comes with detailed explanations for its behavior and CCured comes with many suggested ways in which users can "fix" such "bugs" by explaining their intent to the system.

Future Work

In the next five years I intend to explore ideas related to my current research on software quality. I really enjoy this sort of work and I look forward to doing more of it. Essentially, I believe that bug finding alone is insufficient and I want to investigate higher-level design issues. I

believe that this area of research needs a change of focus in the problems being addressed more than it needs incremental improvements to existing techniques. We are very good at finding mistakes, but what should we do with them once we find them and how can we ensure that we won't make the same mistakes in the future?

My current research can find error-handling bugs in programs and give programmers the some tools to fix them. Given that bugs can be found, I want to go beyond that and help the user understand and fix the problem. To that end, I am interested in inference algorithms for the automatic placement of compensation stacks or similar error-handling features. I propose to go one step further and have my system automatically propose possible ways in which the program's error handling could be restructured to avoid mistakes. At the very least, this work would provide another concrete way of explaining the results of the analysis -- textually suggesting a way to fix the problem is an effective way of explaining what is wrong. At its best, this work would reduce to practically zero the cost of improving software quality with respect to error handling. With my current research an off-the-shelf program can be mined for important error-handling specifications and specification violations can be found automatically. With a feature placement inference algorithm a proposed strengthening of the program's error handling to fix the violations would merely need to be reviewed and applied. I plan to base such a placement algorithm on region inference algorithms and dependency analysis. The goal is to discharge remembered obligations as early as possible, but not before all parts of the program are done using them.

If static and dynamic techniques are to be integrated into software engineering practices, error reporting must be improved. I am concerned that the error reports generated by software quality tools can often only be understood by tool experts. I have seen programmers reject complete back-traces leading to errors as too complicated or not sufficiently indicative. The problem is only exacerbated by concurrency or heap data structures. Ultimately, finding and reporting bugs will not increase software reliability if programmers are unable to understand the reports well enough to fix the bugs. My belief is that the standard trace-based error report is the wrong abstraction for showing software faults. As a start, I plan to investigate combining such reports with invariant-based explanations (e.g., which invariants or specifications are violated and at what point?), possibly drawing on specification mining techniques, and also explanations based on suggested fixes (e.g., given the program, the bug and the specification, how would the tool have changed the program to fix the bug?).

Finally, from a broader design perspective, I am interested in investigating more novel programming language features for error handling and reliability. Since I have seen that unexpected control-flow paths are a large source of errors, I would like to start by reducing the number of such paths that programmers have to consider. My compensation stacks deal with obligations or code in loosely the same way that region memory management deals with allocated objects. However, some design decisions were made in order to make the idea easy to validate by a graduate student. In the future I want to consider more ambitious design research directions. Many of my ideas for compensation stacks grew out of close collaboration (i.e., attending meetings, retreats and presentations) with the Berkeley-Stanford Recovery-Oriented Computing systems group and with researchers at the Center for Hybrid and Embedded Software Systems. I hope to continue working just as closely with systems researchers and software engineers because I believe that such interaction will lead to well-justified areas of improvement.

Some work has already been done in these three general directions but I see this area of software quality research as generally wide open, quite important, and one in which I would like to make a contribution.